A Social Semantic Infrastructure for Decentralised Systems Based on Specification-Carrying Code and Trust[1]

Giovanna Di Marzo Serugendo and Michel Deriaz

Abstract. Decentralised systems made of autonomous devices and software are gaining more and more interest. These autonomous elements usually do not know each other in advance and act without any central control. They thus form a society of devices and software, and as such need: *basic interaction mechanisms* for understanding each other and a *social infrastructure* supporting interactions taking place in an uncertain environment. In an effort to go beyond pre-established communication schema and to cope with uncertainty, this paper proposes an interaction mechanism based exclusively: on semantic information expressed using specifications, and on a social infrastructure relying on trust and reputation.

1 Introduction

The growing diffusion of personal devices connected to Internet is promoting the development of pervasive and wireless applications, as well as those that are to be deployed on a Grid or on a P2P network. A key characteristic of these applications is their self-organised and decentralised nature, i.e., they are made of autonomous software entities which do not know each other in advance and act without any central control. These software entities need advanced means of communication: for understanding each other, to gather and share knowledge, information and experience among each other, and to ensure their own security (data integrity, confidentiality, authentication, access control). Therefore, such a technology needs a social infrastructure supporting, in an intertwined way: mutual understanding, knowledge sharing and security support.

This paper proposes to combine a meta-ontology framework with a dynamic trust-based management system, in order to produce a social semantic middleware supporting the diffusion of semantic information among interoperable software.

The proposed infrastructure relies on the notion of Specification-Carrying Code (SCC) as a basis for mutual understanding, acting as a meta-ontology. Each autonomous software entity incorporates more information than its operational behaviour, and publishes more data than its signature. The idea is to provide separately, for each entity, a functional part implementing its behaviour - the

[1] A full version of this paper appeared under: G. Di Marzo Serugendo, M. Deriaz: "A Social Semantic Infrastructure for Decentralised Systems Based on Specification-Carrying Code and Trust". In Proceedings of the Socially-Inspired Computing Workshop. pp. 143-152. D. Hales and B. Edmonds (Eds). 2005. http://cui.unige.ch/~dimarzo/papers/sic05.pdf

traditional program code; and an abstract description of the entity's functional behaviour - a semantic behavioural description under the form of formal specification. In order to cope with the uncertainty about the environment, and peer entities, individual entities maintain as well local trust values about other entities and share trust and reputation information among them.

Such an interaction mechanism is useful for large scale systems (world-wide, or with high density), where a centralised control is not possible, and for which a human administration must be completed by a self-management of the software. Domains of applications of such an interaction mechanism include P2P, Grid computing systems, as well as emerging domains such as Autonomic Computing, or Ambient Intelligence. Section 2 presents the principles of the Specification-Carrying Code paradigm and the associated Service Oriented Architecture. Section 3 then explains how trust-based management systems can be combined with SCC in order to produce a social semantic infrastructure supporting autonomous decentralised software.

2 Specification-Carrying Code

At the basis of any social life, we find communication capabilities. Communication is grounded on common understanding of the information that is transmitted along communication media. Societies of devices and software need interactions based on a common understanding, i.e. relying on a common semantics. Current practice usually relies on pre-established common meanings, such as shared APIs or shared ontologies. We foresee that future programming practice will consist in programming components and "pushing" them into an execution environment which will support their interactions. Therefore, future components will be developed so as to share a minimal design time common understanding.

The idea advocated in this paper is that interactions should be based on a minimal common basis, merely *concepts*. Pragmatically, for artificial entities to understand each other, those concepts have to be expressed in some language. Therefore, the minimal common basis consists in a common *specification language* used for expressing the concepts. Concepts can then be expressed with different words, and with different properties, but equivalent concepts should share equivalent properties. Thus, there is no need to share identical expression of concepts (either through APIs, ontologies, or identical specifications). However, it is necessary to have a run-time tool able to process those specifications and to determine which of them refer to the same concept.

In practice, in addition to their code, entities carry a specification of the functional (as well as non-functional capabilities) they offer to the community. The specification is expressed using a (possibly formal) specification language, for instance a higher-order logical language defining a theory comprised of: functions, axioms and theorems. The specification acts as a meta-ontology and describes semantically the functional and non-functional behaviour of the entity. We call this paradigm *Specification-Carrying Code (SCC)*. In our current model, a service-oriented architecture supports the paradigm. Before interacting with a service providing entity, a requesting entity may check (through run-time proof checking) some of its own

theorem on the submitted theory. Vice-versa, before accepting to deliver a service, a service providing entity may check the correctness of the requesting entity. This allows an entity to interact with another entity only if it can check that the way the other entity intends to work corresponds to what is expected. The important thing to note here is that entities do not share any common API related to the offered/requested service. Indeed, since entities do not know in advance (at design time) with which entities they will interact, the specification language acts as the minimal common basis among the entities. The lack of APIs implies in turn that input/output parameters can only be of very simple types.

The Specification Carrying Code paradigm is supported by a service-oriented architecture, where autonomous entities register specifications of available services, and request services by the means of specifications. The SCC paradigm supports two basic primitives: a service providing entity *registers* its specification to some run-time middleware that stores the specification in some repository. An entity requesting a service specifies this service through a specification, and asks the run-time middleware to *execute* a service corresponding to the specification.

Once it receives an execute request the run-time infrastructure activates a model checker that determines which of the registered services is actually able to satisfy the request (on the basis of its registered specification). The theorem checker establishes the list of all services whose semantics corresponds to the request. Depending on the implementations, the run-time infrastructure may either chose (non-deterministically) one service, activate it and give back the result (if any) to the requesting entity; or pass the information to the requesting entity which will directly contact the service provider. In the first case, the communication is anonymous, while in the second case it is not. Depending on the situations, both cases are valuable.

Depending on the chosen specification language, the specification may vary from a series of keywords together with some input/output parameters description, to a highly expressive formal specification consisting of a signature and additional axioms and theorems characterising the behaviour of the operators specified in the signature. Services matching requests are not necessarily specified in the same textual manner. The theorem checker ensures that they have the same semantics. The more expressive is the specification language, the more it allows to get rid of shared conventions or keywords.

2.1 A Semantic Service-Oriented Architecture

We have realised two different implementations of the service-oriented architecture supporting the SCC paradigm.

The first implementation has been realised for specifications expressing: signatures of available operators whose parameters are Java primitive types; and quality of service required. Both operators name and quality of service are described using keywords. The resulting environment, a middleware called LuckyJ, allows server programs to deposit a specification of their own behaviour or of a requested behaviour at run-time. In the LuckyJ environment activation of services occurs anonymously and asynchronously. The service providing entity and the service requesting entity never enter in contact, communication is ensure by the LuckyJ middleware

exclusively. The requesting entity is not blocked waiting for a service to be activated. Experiments have been conducted for dynamic evolution of code, where the services can be upgraded during execution without halting or provoking an error in the client program. This is an important feature of decentralised applications since the application transparently self-adapts to new (or updated) services introduced into the environment. The LuckyJ environment only allows the description of basic specification relying on ontology (keywords) shared among all the participating services [5]. Even though LuckyJ allows purely syntactic specifications, it nevertheless proved the viability of the approach under the form of a service-oriented architecture, and its usefulness for dynamic evolution of code.

In order to remove the need for interacting entities to rely on pre-defined keywords, a second implementation of the above architecture has been realised. This architecture allows entities to carry specifications expressed using different kinds of specification language, and is modular enough to allow easy integration of new specification languages [2]. This architecture supports simple primitives for an entity to register its specifications or to request a service, and for the environment to execute the corresponding requested code once it has been found.

The current prototype supports specifications written either in Prolog, or as regular expressions. However it cannot check together specifications written in two different languages. In the case of Prolog, the middleware calls SWI Prolog tool to decide about the conformance of two specifications, in the case of regular expressions we have implemented a tool that checks two regular expressions, and is able to transform them into Java code.

3 Combining SCC and Trust-Based Systems

Human beings exchange different kinds of *semantic* information for different types of purposes: to understand each other, to share knowledge about someone or something else, to take decisions, to learn more, etc. Despite people share the same understanding regarding information, this information remain local, incomplete and uncertain, leading people to rely on trust to actually take decisions. A common example is provided by the trust put into banking establishments, acting as largely trusted third parties for credit card based interactions.

It is similar for artificial entities that are situated into uncertain environments and that have to interact with unknown entities. Specifications help understanding. However nothing prevents a malicious entity to not follow its specification. In order to fully verify this point, the specification should be accompanied by a proof asserting that the code actually satisfies the specification. Unfortunately, even if a formal proof ensures that the code is not malicious and that it follows its specification, the same code can be, due to bad operational conditions, unable to perform the intended service. Therefore, instead of relying on formal (rigid) proofs, we have preferred to consider a trust-based mechanism that allows run-time adaptation to peers behaviour.

The model we intend to build thus considers the following two above aspects of human behaviour: (a) communication through semantic information (SCC); and (b)

ability to take decisions despite uncertainty based on the notion of trust and risk evaluation.

3.1 Trust-Based Systems

Trust-based systems or reputation systems take their inspiration from human behaviour. Uncertainty and partial knowledge are a key characteristic of the natural world. Despite this uncertainty human beings make choices, take decisions, learn by experience, and adapt their behaviour. We present here a research work from which we will take inspiration to extend our current architecture.

SECURE Trust System. The European funded SECURE project has established an operational model for trust-based access control. Systems considered by the SECURE project are composed of a set of entities that interact with each other. These entities are autonomous components able to take decisions and initiatives, and are meaningful to trust or distrust. Such entities are called *principals*. Principals are for instance portable digital assistants (PDAs) acting on behalf of a human being, or personal computers, printers, mobile phones, etc. They interact by asking and satisfying services to each other.

In a system based on the human notion of trust [1], principals maintain local *trust values* about other principals. A principal, who receives a request for collaboration from another principal, decides or not to actually interact with that principal on the basis of the current trust value it has on that principal for that particular action, and on the risk it may imply of performing it. If the trust value is too low, or the associated risk too high, a principal may reject the request. A PDA requiring an access to a pool of printers may see its access denied if it is not sufficiently trusted by the printers. For instance, it is known that this PDA sends corrupted files to the printers.

After each interaction, participants update the trust value they have in the partner, based on the evaluated outcome (good or bad) of the interaction. A successful interaction will raise the trust value the principal had in its partner, while an unsuccessful interaction will lower that trust value. Outcomes of interactions are called *direct observations*. After interacting with a printer, a PDA observes the result of the printing. If it is as expected, for instance double-sided, and the document is completely printed, the PDA will adjust the trust value on that particular printer accordingly.

A principal may also ask or receive *recommendations* (in the form of trust values) about other principals. These recommendations are evaluated (they depend on the trust in the recommender), and serve as *indirect observations* for updating current trust values. As for direct observations, recommendations may either raise or lower the current trust value. We call *evidence* both direct and indirect observations. Some PDAs may experience frequent paper jams, on a given printer. They will update (in this case lower) their trust value in that printer, and advertise the others, by sending them their new trust value. The PDA that receives this recommendation will take it into account, and decide if it uses that printer or not [6].

Thus, trust *evolves* with time as a result of evidence, and allows to adapt the behaviour of principals consequently.

3.2 Towards a Social Semantic Service Oriented Architecture

Derived from the SECURE trust-based access model, we describe here trust-based interactions rules grounded on semantic information exchange and global emergent reputation:
Request for collaboration and exchange of specifications. A principal A receives a request for collaboration from another principal B. A and B exchange their respective capabilities under the form of a specification expressed in the specification language. They learn each other about their respective provided services.
Decision to interact. Based on the received specification, A and B respectively evaluate if the services provided by the other fulfill its needs (checking of properties expected to be satisfied by the partner).

The decision then depends on the evaluation of the specification, past direct observations of interactions with B (if any), previously received recommendations about B from other entities, current trust value A has about B, and the risk incurred by the interaction. A may also decide to ask score managers about the reputation of B.
Trust Update. If A decides to interact with B, it will observe the outcome of the interaction, evaluates it (positive or negative), and updates accordingly the local trust value it maintains about B.
Recommendations. Besides collaboration requests, A may receive a recommendation from B under the form of specification defining the degree of trust the recommender has on a subject C. Recommendations are evaluated with respect to trust in the recommender, and make the trust A has in the subject C evolve (increase or decrease).

The model defines then a homogeneous framework which serves for expressing and checking semantic information of different kinds: functional behaviour, non-functional behaviour, observations, and recommendations.

4 Open Issues

We identify the following issues that need to be addressed before this paradigm can be fully exploited in everyday life:
Content Description. There is currently no agreed structure, format and consequently no standard for stating the content of functional description, non-functional descriptions, and policies. The richer the information, the more powerful interactions can be envisaged, but the more complex the languages and related tools become and the more power consuming the computation becomes.
Modelling Languages. Languages for expressing the information vary from simple keywords, to more structured ontologies, to algebraic specification, to different kinds of logics (temporal logics, descriptive logics, higher-order logics, etc.), and even to category theory. The more expressive the language, the more powerful the management of the models can be, but the more difficult the corresponding automated tools are.
Models Processing Tools. In addition to the language for expressing the specification, it is necessary to have run-time efficient tools for processing them, either specification checkers or theorem provers.

Run-time infrastructure. Implementing run-time infrastructure supporting both the processing of formal specifications and the corresponding code execution may turn to be a complex task, despite existing proof-of-concepts for such infrastructures.

5 Conclusion

The model proposed here follows the separation into individual capabilities and social organisation mentioned by Minsky [4]. The exchange of functional and non-functional capabilities in our model corresponds to the diffusion of knowledge about the capabilities of individual principals. The use of trust and the exchange of recommendations adds a social layer on top of the interaction mechanism. Typical applications that can benefit from this technology include wireless cellular network routing, ambient intelligence systems, autonomic computing systems [3], or access control systems.

Acknowledgement

This work was partly supported by the EU funded SECURE project (IST-2001-32486), and by the Swiss NSF grant 200020-105476/1.

References

[1] V. Cahill and al. Using trust for secure collaboration in uncertain environments. *IEEE Pervasive Computing Magazine, special issue Dealing with Uncertainty*, 2(3):52-61, 2003.

[2] M. Deriaz and G. Di Marzo Serugendo. Semantic service oriented architecture. Technical report, Centre Universitaire d'Informatique, University of Geneva, Switzerland, 2004.

[3] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41-50, January 2003.

[4] M. Minsky. *La Société de l'Esprit*. InterEditions, 1988.

[5] M. Oriol and G. Di Marzo Serugendo. A disconnected service architecture for unanticipated run-time evolution of code. *IEE Proceedings-Software, Special Issue on Unanticipated Software Evolution*, 2004.

[6] S. Terzis, W. Wagealla, C. English, P. Nixon, and A. McGettrick. Deliverable 2.1: Preliminary trust formation model. Technical report, SECURE Project Deliverable, 2004.