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

Giovanna Di Marzo Serugendo*
*Centre Universitaire d'Informatique
University of Geneva
24, rue Général-Dufour
CH-1211 Geneva 4
Giovanna.Dimarzo@cui.unige.ch

Michel Deriaz†
†Centre Universitaire d'Informatique
University of Geneva
24, rue Général-Dufour
CH-1211 Geneva 4
Michel.Deriaz@cui.unige.ch

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 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. Finally, Section 4 describes some related works.

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. In the case of social insects, pheromone deposited by ants in their habitat is correctly understood depending on whether it refers to food, or to the nest. In the case of human beings, words of the language refer to well understood concepts. Similarly, 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: communication through shared APIs, usually already shared at design time and which are uniquely syntactic expressions of signatures; communication through shared ontologies allowing run-time adequacy but requiring sharing of keywords. 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.

Pushing the idea at its extreme, even different specification languages could be used simultaneously by different entities to communicate provided there are translators from one language to the other. But this is beyond the scope of this paper.

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.

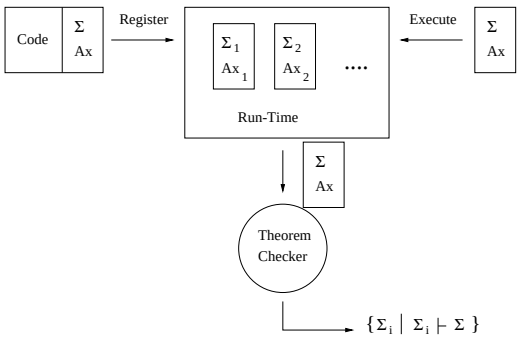

Figure 1: SCC Principle

Figure 1 shows two basic primitives of the SCC paradigm: 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

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. We have realised two different implementations of this service-oriented architecture.

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 (Oriol and Di Marzo Serugendo, 2004). 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 (Deriaz and Di Marzo Serugendo, 2004). 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. We foresee the integration of additional specification languages, such as Higher-Order Logic (HOL) and Isabelle theorem checker, JENA, and the Common Simple Logic (CSL).

These languages have different expressive powers: regular expressions are a powerful tool for describing syntactic expressions, and do not support expression of semantic properties. Prolog and HOL are logical languages allowing rich expressivity for describing properties. However, it can rapidly become impracticable to describe usual things such as printing, or complex lists. Therefore, we are investigating languages allowing both logical expressivity and some ontological concepts, such as Jena or CSP.

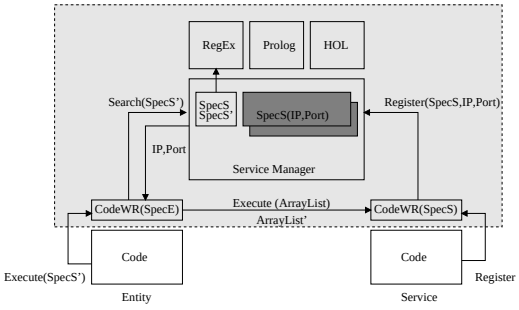

Figure 2: Semantic Service-Oriented Architecture

Figure 2 shows the implemented semantic service-oriented architecture. A *Code* wishing to provide a service or requesting a service is first encapsulated into a wrapper *CodeWR*, which is responsible to handle the specification *SpecS* corresponding to the behaviour of *Code*, and to handle the two basic primi-

tives *Register* and *Execute*. The advantage of using such a wrapper is that with very minor modifications any existing server code can become a specification-carrying code.

A run-time environment, called *Service Manager*, stores specifications of registered services, and activates the corresponding Theorem Checkers once a request has been submitted. In case a service corresponding to the request has been found, the wrapper of the requesting entity then receives the necessary information (IP address and Port number) for contacting directly the service.

The *Code* is not aware that there has been a direct call to a service; the wrapper has transparently managed the whole request. If we consider the wrapper being part of the middleware, communication is anonymous, as in our previous implementation.

Additional information related to programming services and requests can be found in Deriaz and Di Marzo Serugendo (2004).

2.2 Example

2.2.1 Regular Expressions

A specification file is a XML file divided into subsections. Each subsection corresponds to a particular language. Each subsection has to be self- contained: it describes completely a service or a request. A specification file is structured as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<specs>
  <regex active="true">
    ...
  </regex>
  <prolog active="false">
    ...
  </prolog>
</specs>
```

Once it has received an entity request, the service manager tries to match the request specification with the service specification for all languages that are active. In the above example, we see that two languages are defined (regex and prolog) but only one is active (regex). It means that only regular expressions will be taken into consideration. XML allows us to define a different structure for each language. For example in the case of regex, we have four tags: `<name>` which denotes the name of the service, `<params>` which describes the expected parameters, `<result>` which defines the structure of the result, and `<comment>`, which contains optionally additional information.

The following is an example of a sorting *service publication* defined by the regular expression:

```
<specs>
 <description active="true">
   <content> Sorting Service</content>
<</description>
<regex active="true">
   <name>(?i)\w*sort\w*</name>
   <params>String\*</params>
   <result>String*</result>
 </regex>
</specs>
```

The regular expression describing the name `((?i)\w*sort\w*)` accepts all the words that contain the word sort, like quicksort, sorting, or sort. `(?i)` sets the matching case insensitive. The parameters are expressed be the `String\*` regular expression, which means that we expect a list of 0, 1 or more Strings. If we would expect exactly three Strings (for example), we would write String String String. The result tag indicates that this service returns a list of Strings as well. Note that it is of course only a trivial example; the power of regular expressions allows us to express a service name much more precisely.

A service request then is expressed in the following manner:

```
<specs>
  <description active="true">
    <content>Sorting Request</content>
  </description>
  <regex active="true">
    <name>sort</name>
    <params>String*</params>
    <result>String\*</result>
  </regex>
</specs>
```

New tags can be added in the future. Another language can have a completely different structure. These two last points justify the use of such an extensible language as XML.

2.2.2 Prolog

Below is service, able to reverse lists, expressed in Prolog. This service defines first the `append` operator which is necessary to define the reverse operator `rev`. Appending any list `L` to the empty list `[]` returns `L` (line 9). Appending any list `L2` to a non-empty list `[H|T]` (Head and Tail) returns a list with the same head `H` and with `L2` appended to `T` (lines 10, 11).

The `rev` operator is then defined: reversing the empty list, returns the empty list (line 13); and reversing a non-empty list `[H|T]` returns a list `R` obtained by recursively applying `rev` on the tail of the list and appending the head at the end (lines 14, 15).

```
1 <specs>
   <description active="true">
3   <content> Reverse Service</content>
  </description>
5  <regex active="false">
  </regex>
7 <prolog active="true">
    <content>
9    append([],L,L).
     append([H|T],L2,[H|L3]) :-
11       append(T,L2,L3).

13   rev([],[]).
     rev([H|T],R) :-
15       rev(T,RevT), append(RevT,[H],R).
    </content>
17 </prolog>
 </specs>
```

The specification request simply describes the axioms expected to be satisfied by a reverse operator here called `revlist` (lines 9, 10), as well as the property that reversing two times a list returns the original list (line 11).

```
1 <specs>
   <description active="true">
3    <content> Reverse Request </content>
  </description>
5 <regex active="false">
  </regex>
7 <prolog active="true">
    <content>
9    revlist([],[]), revlist([A|B],R),
     revlist(B,RevB), append(RevB,[A],R),
11   revlist([A|B],R) , revlist(R,[A|B]).
    </content>
13 </prolog>
  </specs>
```

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

The semantic information is expressed using a specification language conveying the semantic part of the specification. Run-time checked properties assess semantic meaning. Local context information is also provided under this form. This is useful for mobile devices or mobile code.

As said before, even if properties have been checked, the underlying code can be malicious, or for some reason it cannot follow its specification. Therefore, in addition to the notion of specification (formally describing the basis of interactions), a trust-based model is used for sharing knowledge among entities. This allows run-time adaptation to current behaviour, based on direct observations, and recommendations. The same semantic framework serves for expressing recommendations, or diffusing observations (theories can be dynamically built and modified). In addition, the propagation of properties or theorems integrates well into the trust framework, since sending a theorem is one form of recommendation. Entities exchange information conveying different types of meaning: functionality, non-functional aspects, quality of service, current state; events (recommendations, security attacks, observations), etc.

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 two research works from which we will take inspiration to extend our current architecture: an operational model for trust-based control, and a trust calculation algorithm that allows to calculate a global (emergent) reputation from locally maintained trust values.

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 (Cahill and al., 2003), 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 (Terzis et al., 2004).

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

EigenTrust. EigenTrust (Kamvar et al., 2003) is a reputation system for P2P networks in which every peer rates the peers from whose they download files. It is an interesting solution to the problem of maintaining in a totally decentralised manner local trust values that globally converge to an emergent reputation value. These values are stored in a local trust vector. Starting from theses local trust values, the distributed EigenTrust algorithm computes a global trust vector, representing the global reputation of each peer. Each peer computes this vector and the authors proved that the computation will always converge to the same global trust vector. Simulations of systems, based on this trust mechanism, show that the number of inauthentic files downloaded by honest peers still significantly decreases even if up to 70% of the peers collude in order to subvert the system.

The idea is that the global reputation of one peer depends on what other peers think about it, according to the successfulness of former transactions, on what friends think about it, on what the friends of friends think about it, and so on; if the chain is long enough, the result of the computation converges to the global trust value.

A set of peers, called score managers, is assigned to each peer. A score manager is responsible to store the global trust value, i.e. the emergent reputation value, of its daughter peer. To determine the score managers of a specific peer, a client peer will apply different distributed hash functions on the peer's identity. All honest score managers of a specific peer will then give the same global trust value.

3.2 Towards a Social Semantic Service Oriented Architecture

In order to incorporate a social layer into our current semantic architecture, we are planning: to extend our current interaction model in order to incorporate trust information; and to adapt the EigenTrust algorithm from file sharing to services requests.

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 fulfil 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.

- *Reputation Update.* Once local trust values have been updated, the EigenTrust algorithm is then started and the new value of the global reputation is computed.

- *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.

3.3 Discussion

Using EigenTrust in our architecture will allow users to ask services only to reputable peers and exclude malicious peers. Starting from the current EigenTrust algorithm, we intend to address the following issues:

Two-ways rating. In its current form EigenTrust allows one-way rating only. In the systems we consider, we need a two-ways rating. Indeed, like in eBay, where both buyer and seller rate each other, we want that service providers and clients rate each other after every transaction. On the client side it is obvious that we want to know which services are reputable and which are malicious ones. On the service side it is also interesting to avoid malicious clients that try to make denial of services attacks or that try to corrupt the service by sending bad parameters.

Privilege good principals. In order to encourage principals to provide good services, we suggest privileging those with a high reputation. In case of a network overload, a reputable service will serve only reputable clients. In fact, the more a principal becomes reputable, the more it will deal with high-trusted peers.

Different trust values. The EigenTrust algorithm defines only one trust value for each peer. The authors claim that a peer that provides good files will also be good in providing trust values for other peers. In the case of our architecture, we prefer to compute different trust value: one for each available service, one for the behaviour of a principal when it acts as a client, and one indicating its reliability for trust computation of other peers.

Reputation Update. The EigenTrust algorithm implies that reputation values are all calculated together, since trust values are all closely linked and dependent of each other. However, we could consider a more flexible algorithm, still inspired by EigenTrust, that as well converges to the global emergent reputation, but not necessarily in one shot. The reputation value would converge slowly but the whole algorithm would not affect the efficiency of the system.

Distributed Architecture. Our SOA architecture is currently centralised. The Service Manager acts as a server that connects client entities with services. It is similar to Napster; clients ask the server for a specific file, and the server responds with the address of the peer that contains it. The main difficulty that we will have to face to obtain a completely decentralised architecture is the problem of peer discovery. Where should a peer connect in order to find a service? In many well-known P2P file sharing systems, like in Kazaa, the peers that have a high-speed connection are automatically designed as super-nodes. A super-node is a peer like another, but which adds a directory service. All other peers connect to the closest super-node in order to locate a specific file. If the

super-node does not have it, it transmits the request to another super-node.

In our future distributed architecture, the centralised service manager will disappear. The directory functionality provided by the service manager will become a service like another. Every peer can therefore act as a service manager.

3.4 Example

The following small example shows how a group of computers can share a pool of printers through our envisioned infrastructure. Before interacting with each other computers and printers exchange their respective functional as well as non-functional capabilities, e.g. a printer claims that it is a postscript double-sided printer, and a computer asks to print a PDF file. After having interacted with a printer, the computer stores the observation related to its experience with the printer (works as expected, only one side, no impression at all, etc.). Depending on the outcome of the interaction, or if it has been requested to do so, the computer may want to share its knowledge with some of the other computers. It will then inform the others that the printer is not actually double-sided, but only single sided, or that the printer went out of toner, and is no longer available, or that one of the printers is faulty and has a random behaviour.

This example shows that: printers and computers can exchange information about their respective functional and non-functional behaviour; computers can exchange information among themselves about the printers and other computers state or actual capabilities (independently of their claimed functionality); the shared knowledge allows computers to efficiently use the remaining set of working printers (adaptation, resource management), as well as to correctly inform the user about the nearest well functioning printer. This example shows as well the validity of information. The faulty printer has a random behaviour, this is a long term valid information (information is not very accurate, but not volatile). However, if the printer has been able to print two minutes ago, we can almost be sure that it will be able to print in the next couple of minutes, but not necessarily later (information is accurate but highly volatile). This example raises also the question of the accuracy of shared information. In the case of the printer, it claims that it can print, but actually it cannot. In the case of a computer, it can claim that the printer is out of order, but it may lie. In both cases, sharing knowledge about printers or other computers helps circumvent the problem, and adapt the individual as well as the collective behaviour to the environment.

4 State of the Art

Specification-Carrying Software. The notion of specification-carrying software is being investigate since several years at the Kestrel institute (Pavlovic, 2000; Anlauff et al., 2002). This idea has been proposed initially for software engineering concerns, essentially for: ensuring correct composition of software and realising correct evolution of software. Algebraic specifications and categorical diagrams are used for expressing the functionality, while coalgebraic transition systems are used to define the operational behaviour of components. The visions of this team include as well run-time generation of code from the specifications. Compared to these works, this paper proposes a version where the behaviour of a component is not fully specified in all its operational details, but sufficiently in order to be used for correct self-assembly of software at run-time.

Meta-Ontologies. Meta-ontologies are algebra allowing definition of type theories, operations, and axioms. From that perspective, category theory (Johnson and Dampney, 2001), higher-order logics that define terms, operators, axioms, and provable or checkable theorems are meta-ontologies.

Current semantic Web services simply use information, expressed or communicated through languages such as RDF or OWL, as linking glue. However, the exchanged information is not yet used to allow full interoperation, or reactive behaviour to the semantics of information. Middleware addressing both semantic issues and intelligent interoperability are currently an open issue.

Trust-Based Management Systems. Trust management systems deal with security policies, credentials and trust relationships (e.g., issuers of credentials). Most trust-based management systems combine higher-order logic with a proof brought by a requester that is checked at run-time. Those systems are essentially based on delegation, and serve to authenticate and give access control to a requester (Weeks, 2001). Usually the requester brings the proof that a trusted third entity asserts that it is trustable or it can be granted access. Those systems have been designed for static systems, where an untrusted client performs some access control request to some trusted server (Appel and Felten, 1999; Bauer et al., 2001). Similar systems for open distributed environment have also been realised, for instance Li et al. (1999) proposes a delegation logic including negative evidence, and delegation depth, as well as a proof of

compliance for both parties involved in an interaction. The PolicyMaker system is a decentralised trust management systems (Balze et al., 1996) based on proof checking of credentials allowing entities to locally decide whether or not to accept credentials (without relying to a centralised certifying authority).

Tag-Based Models. Tags are markings attached to each entity composing the self-organising application (Hales and Edmonds, 2003). These markings comprise certain information on the entity, for example functionality and behaviour, and are observed by the other entities. In this case the interaction would occur on the basis of the observed tag. This is particularly useful if applied to interacting electronic mobile devices that do not know each other in advance. Whenever they enter the same space, for example a space where they can detect each other and observe the tags, they can decide on whether they can or cannot interact.

Smart labels/Smart Tags. Smart tagging systems are already being deployed for carrying or disseminating data in the fields of healthcare, environment, and user's entertainment. For instance, in the framework of data dissemination among fixed nodes, (Beaufour et al., 2002) propose a delivery mechanism, based on the local exchange of data through smart tags carried by mobile users. Mobile users or mobile devices do not directly exchange smart-tags; they only disseminate data to fixed nodes when they are physically close to each other. Data information vehicled, by smart tags, is expressed as triples indicating the node being the source of the information, the information value, and a time indication corresponding to the information generation. Smart tags maintain, store, and update these information for all visited nodes. A Bluetooth implementation of these Smart Tags has been realised in the framework of a vending machine (Beaufour, 2002). In smart tagging systems, data remain structurally simple, and understandable by human beings, and does not actually serve as a basis for autonomous local decisions.

5 Conclusion

The model proposed here follows the separation into individual capabilities and social organisation mentioned by Minsky (Minsky, 1988). 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 (Ducatel et al., 2001), autonomic computing systems (Kephart and Chess, 2003), or access control systems.

In order to experiment this approach with mobile components, we foresee as well to combine our prototype with a positioning system currently deployed in our department.

Acknowledgements

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

References

M. Anlauff, D. Pavlovic, and D. R. Smith. Composition and refinement of evolving specifications. In *Proceedings of Workshop on Evolutionary Formal Software Development*, 2002.

A. W. Appel and E. W. Felten. Proof-carrying authentication. In *6th ACM Conference on Computer and Communications Security*, 1999.

M. Balze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *IEEE Conference on Security and Privacy*, 1996. URL

L. Bauer, M. A. Schneider, and E. W. Felten. A proof-carrying authorization system. Technical Report TR-638-01, Princeton University Computer Science, 2001.

A. Beaufour. Using Bluetooth-based Smart-Tags for Data Dissemination. In *Pervasive Computing 2002*, 2002.

A. Beaufour, M. Leopold, and P. Bonnet. Smart-tag based data dissemination. In *ACM International Workshop on Wireless Sensor Networks and Applications (WSNA'02)*, 2002.

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.

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

K. Ducatel, M. Bogdanowicz, F. Scapolo, J. Leijten, and J.-C. Burgelman. Scenarios for Ambient Intelligence in 2010. Technical report, Institute for Prospective Technological Studies, 2001.

D. Hales and B. Edmonds. Evolving Social Rationality for MAS using "Tags". In J. S. Rosenschein, T. Sandholm, M. Wooldridge, and M. Yokoo, editors, *Second International Joint Conference on Autonomous Agents and MultiAgent Systems*, pages 495–503. ACM Press, 2003.

M. Johnson and C. N. G. Dampney. On Category Theory as a (meta) Ontology for Information Systems Research. In *International Conference On Formal Ontology In Information Systems (FOIS'01)*, 2001.

S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. The eigentrust algorithm for reputation management in p2p networks. In *WWW2003, May 20-24, 2003, Budapest, Hungary*, 2003.

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

N. Li, J. Feigenbaum, and B. N. Grosof. A logic-based knowledge representation for authorization with delegation. In *12th IEEE Computer Security Foundations Workshop*, 1999.

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

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

D. Pavlovic. Towards semantics of self-adaptive software. In *Self-Adaptive Software: First International Workshop*, volume 1936 of *LNCS*, pages 50–65. Springer-Verlag, 2000.

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

S. Weeks. Understanding trust management systems. In *2001 IEEE Symposium on Security and Privacy*, 2001.