Specification-Carrying Code for Self-Managed Systems

Giovanna Di Marzo Serugendo and Michel Deriaz

Abstract. This paper proposes the notion of Specification-Carrying Code as an interaction mechanism for self-assembly of autonomous decentralised software components. 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 and necessary parameters - a semantic behavioural description under the form of a formal specification. Interactions are exclusively based on the specifications and occur among entities with corresponding specifications. In the case of autonomic computing systems, in addition to functional aspects, the specification may carry a semantic description of non-functional information related to self-management. This paper presents the principles of the Specification-Carrying Code paradigm, the associated Service-Oriented Architecture, and it explains how self-managed systems can benefit from this paradigm.

1 Introduction

Key characteristics of autonomic computing systems are their expected self-organised and decentralised nature. They consist of autonomous software entities which do not necessarily know each other in advance, but act together without any direct central control towards high-level goals defined by some human administrator. These software entities need advanced means of communication for: understanding each other, gathering and sharing knowledge, information and experience among each other, and ensuring self-managing functionalities. Therefore, such a technology needs an infrastructure supporting, in an intertwined way: mutual understanding, knowledge sharing and support for self-managing features.

Current practices usually rely on pre-established common meanings: communication through shared APIs, usually already shared at design time and which are uniquely syntactic expressions of signatures; or communication through shared ontologies allowing run-time adequacy but requiring sharing of keywords. We foresee that future programming practice, especially in the case of autonomic computing systems, 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.

This paper proposes an infrastructure that relies on the notion of Specification-Carrying Code (SCC) as a basis for mutual understanding. In addition to its code, each autonomous software entity carries an abstract description of its functional behaviour which is a semantic behavioural description under the form of a formal specification. Section 2 presents the principles of the Specification-Carrying Code paradigm. Section 3 explains the interest and potentiality for autonomic computing systems. Finally, Section 4 describes some related works.

2 Specification-Carrying Code

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.

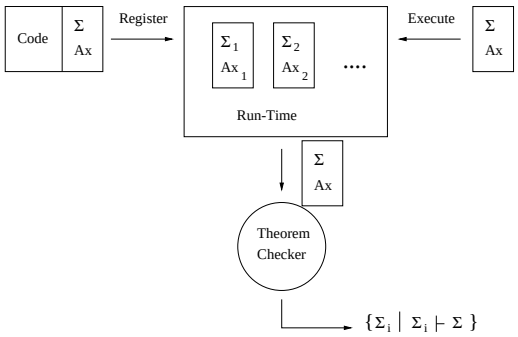

Fig. 1. SCC Principle

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 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, as well as pre- and post- conditions, additional axioms and theorems characterising the behaviour of the operators specified in the signature. Services specifications matching requests specifications 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 required quality of service. 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 ensured 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 self-managed 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 [7]. Even though LuckyJ allows purely syntactical 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 additional specification languages [3]. 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. These languages have different expressive powers: regular expressions are a powerful tool for describing signatures, but do not support expression of semantic properties. Prolog, or Higher-Order Logics (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 the Common Simple Logic (CSL).

2.2 Example

Below is a service expressed in Prolog, stored in a file specService.xml which is able to reverse lists. 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 7). 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 8, 9). The `rev` operator is then defined: reversing the empty list, returns the empty list (line 11); 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 12, 13).

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

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

The specification request, stored in file `specRequest.xml` simply describes the axioms expected to be satisfied by a reverse operator here called `rev` (lines 7, 8), as well as the property that reversing two times a list returns the original list (line 9).

```
1 <specs>
```

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

The following code, in file `ReverseList.java`, consists in a service providing the reverse list functionality:

```
1  import kernel.*;
   import java.util.*;
3
   public class ReverseList extends Service {
5
   public static void main(String[] args) {
7    // register reverse list specification
     new ReverseList().register("localhost",
9                        "specService.xml");
   }
11
   public ArrayList execute(ArrayList list) {
13   Collections.reverseList(list);
     return list;
15 }
   }
```

Class `ReverseList` extends the Service class, available in the kernel package of our architecture. The `main()` method is used to register the service at a service manager available at a localhost. The description of the service is contained in the file `specService.xml`.

Every service extends the Service class and has to redefine the abstract `execute()` method (line 12), which is the only method that will be indirectly invoked by a client. This method ”wraps” the functionality defined in the specification file, here the reverse list functionality (line 13). Parameters are transmitted in an ArrayList and the result is returned in an ArrayList as well.

An entity that wants to use the reverse list service can do it with a single line of code (in a file UseReverseList.java):

```
1  import kernel.*;

3  public class UseReverseList extends Entity {
```

```
5  private void askForReverseList() {

7    // activation of the reverse list service
    result = Entity.execute(
9              SM_ADDRESS, "specRequest.xml",
               parameters);
11 }
   }
```

In the above code, `result` and `parameters` are ArrayLists and SM_ADDRESS is a String defining the service manager address. The entity specification file is available in the same directory and is named `specRequest.xml`. Parameters are added under the form of an `ArrayList`. To execute a request, an entity calls the `execute()` method (line 8) defined in the the Entity class, available in the kernel package of our architecture. Additional information related to programming services and requests can be found in [3].

There is no transfer of API between ReverseList and UseReverseList. Indeed, UseReverseList simply invokes its own static method `execute()` (line 8 of file UseReverseList.java). This method will submit the request present in the specRequest.xml file to a middleware infrastructure called the Service Manager. The Service Manager seamlessly retrieves one service that matches the request. Method `execute()` of ReverseList is activated by UseReverseList with the parameters of UseReverseList. The actual method which does the reverse list activity, here `reverseList()` (line 13 of file `ReverseList.java`) is not called either by UseReverseList or by the Service Manager. The registration and the activation of services are done exclusively through two primitives: `register()` and `execute()` respectively.

2.3 Discussion

Specification matching [6] encompasses *retrieval* of a component from a software library based on its semantics; *reuse* of a component from a software library in order to adapt it to the current needs; *substitution* of a component by another one without affecting the observable behaviour; and *subtyping*. The work presented in this Section is concerned with retrieval of a component whose semantics (registered specification) satisfies a query (specification request). There is no obligation that both specifications are equivalent, it is sufficient that the selected service specification *implies* the specification request.

In our current implementation using Prolog, specifications are registered as Prolog facts and rules, while specification requests are Prolog queries. In the case of regular expressions: the registered specification must match as a regular expression the specification request (but not necessarily the opposite).

3 SCC for Autonomic Computing

Specification-Carrying Code as defined above expresses the functional behaviour of services. In addition to functional aspects, the specification can be extended to incorporate

as well non-functional information, such as quality of service, efficiency, availability, or trust. This section shows the interest of the Specification-Carrying Code paradigm, both in its current form or through an extended version, for self-managed systems. More precisely, it describes how SCC can be useful for realising the four self-management concepts defining the autonomic computing view as explained in [5]. We have actually tested the approach and conducted experiments in the case of self-configuration, and initial tests are under way related to self-protection.

Self-Configuration. "Automated configuration of components and systems follows high-level policies. Rest of system adjusts automatically and seamlessly [5]."

The notion of specification naturally serves to express high-level goals as stated by a (human) administrator. They constitute initial service requests triggering the rest of the system. Similarly, service requests expressing high-level configuration policies may serve for automatic distribution of entities or placement on a Grid of components participating to a scientific calculation. Finally, at a local level, each component of an autonomic computing system may describe, through its specification, its own installation needs (e.g. CPU, Memory, or Network).

For the particular case of automatic and seamless integration of new components, initial experiments [7] have proven useful for dynamic run-time evolution of code. Indeed without stopping any specific entities or the whole system it has been possible to: add in the system and seamlessly use additional features; to seamlessly replace updated entities without the calling entities noticing the replacement (even during a call). Since a specification request is the only element necessary for activating a service, inserting a new functionality then simply consists in registering its corresponding service to the middleware. Replacing an updated entity consists in having both the old and the new entities present at the same time in the system, and if necessary to transfer the state of the old entity to the new one before stopping the updated entity.

Self-Optimisation. "Components and systems continually seek opportunities to improve their own performance and efficiency [5]."

The architecture described in the above sections implies that for each request, the middleware searches for all possible services realising the request. This turns out to be useful for self-optimisation. Indeed, as soon as a service realising a request is available (it simply needs to register itself), it can be selected by the middleware. If the request specifies that the most updated service is required, then the new service will be chosen. It is interesting to note that if the new service itself requires updated services to satisfy its needs, by a cascading effect a large part of the system will then use updated functionalities. Additionally, the specification may express optimisation policies, or describe configuration parameters that can be tuned differently for different contexts.

Self-Healing. "System automatically detects, diagnoses, and repairs localized software and hardware problems [5]."

The Specification-Carrying Code paradigm can be extended to incorporate automatic generation of a code, recognised as erroneous, from its corresponding specification. Alternatively, if a code has been recognised as erroneous by the middleware, its specification is removed from the repository of available services. The whole system then automatically works with the current available services, maybe in a degraded man-

ner, until a new code is inserted into the system and replaces the erroneous one. The notion of specification also serves as a basis for verifying the adequacy of a code, e.g. through proof-carrying code techniques.

Self-Protection. "System automatically defends against malicious attacks or cascading failures. It uses early warning to anticipate and prevent systemwide failures [5]."

A specification may rather naturally describe high-level security policies that have to be realised in the whole system. Moreover, self-regulating schema can also be considered. For instance, combining trust and reputation information with specifications may prove to be an efficient tool for self-protection [4]. In this schema, the specification is extended to incorporate trust and reputation information about a service provider, or about a client. This information, updated at run-time, then serves to accept or deny interaction requests. Checking a specification against its code implies to (formally) prove that the code actually satisfies the specification. However, even if a code is not malicious, i.e. such a proof has been validated, there may be some external or internal condition that nevertheless prevents the code to furnish correctly its service. The use of trust allows to permanently adapting the whole system behaviour to the individual entities behaviour or reponses to services requests.

Discussion. Automated reasoning as proposed in Section 2 mainly concerns functional aspects: a service's specification logically implies a specification request. Formal specifications and automated reasoning solve interoperability problems: there is not need for compatible interfaces or exact declarations and queries. More generally, Section 3 advocates the use of automated reasoning for building autonomic networks: specifications serving to express functional properties, as well as non-functional properties, (re)configuration policies, and interaction protocols.

4 Related Works

Specification-Carrying Software. The notion of specification-carrying software is being investigate since several years at the Kestrel institute [8, 1]. 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 function-based 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.

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, [2] 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 carried 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. 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. The notion of specification-carrying code as expressed in this paper can be seen as a way to tag codes instead of only data.

5 Conclusion

Specification-Carrying Code provides several advantages for run-time execution of decentralised autonomous software in general, and for autonomic computing systems in particular. Among them we can cite interaction and interoperability with unknown entities, seamless integration of new entities and functionalities, possible combination of services, robustness against errors or failures. However, Specification-Carrying Code alone is not sufficient. It needs to be combined, for instance, with additional mechanisms for checking the adequacy of a code with its published specification, for discovering errors, and propagating information about erroneous code, for correlating information and detecting malicious attacks.

6 Acknowledgements

This work is partly supported by Swiss NSF grant 200020-105476/1.

References

1. M. Anlauff, D. Pavlovic, and D. R. Smith. Composition and refinement of evolving specifications. In *Proceedings of Workshop on Evolutionary Formal Software Development*, 2002.
2. 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.
3. M. Deriaz and G. Di Marzo Serugendo. Semantic service oriented architecture. Technical report, Centre Universitaire d'Informatique, University of Geneva, Switzerland, 2004.
4. G. Di Marzo Serugendo and M. Deriaz. A social semantic infrastructure for decentralised systems based on specification-carrying code and trust. In D. Hales and B. Edmonds, editors, *Socially-Inspired Computing*, 2005.
5. J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, January 2003.
6. A. Moormann Zaremski and J. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, October 1997.
7. 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.
8. 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.