Defining Open Software Architectures for Customized Remote Execution of Web Agents

Valérie Issarny and Titos Saridakis
IRISA / INRIA, *Campus de Beaulieu, 35042 Rennes Cédex,* FRANCE
({issarny,saridaki}@irisa.fr*)*

Abstract. Agent-based solutions promise to ameliorate Web services, by promoting the modular construction of Web servers, relieving the network from transferring useless data, supporting user mobility, etc. However, existing Web servers do not favor the hosting of agents. This paper proposes a description of agent behavior in terms of its requirements regarding resource utilization (e.g. memory, and disk space), functional services (e.g. system calls), and non-functional properties (e.g. degree of replication, and access control). When formally expressed, these requirements can be used in an automated decision process, which is based on software specification matching techniques. Upon the acceptance of an agent, the host uses these requirements to construct an environment customized to agent's execution. We discuss the benefits of this approach, and how it can be used to promote existing agent-based solutions in the Web framework.

Keywords: Agent, customized execution, execution properties, mobility, remote execution, software architecture, specification matching.

Abbreviations: ADL – Architecture Description Language

1. Introduction

Since its appearance in the early 90's, Web has been greatly evolved. While it was initially intended for the transfer of hypertext documents, its success and world-wide acceptance led the focus of research interest into other directions like distributed computing, electronic commerce, etc. However, from the perspective of intensive use, Web's architecture was not designed for such employment. Consequently, today Web is confronted with a number of problems ranging from server saturation and heavy network traffic, to access control and verification of client requests. One of the most prominent solutions for coping with those problems is the use of *software agents*, a concept introduced to distributed systems from the field of artificial intelligence [1]. In common sense, an agent can be anyone who acts on behalf or in the interest of somebody else. Consequently, a software agent is a piece of code endowed with capabilities that allow it to perform some task in the place of a person or some other piece of code. In the context of this document we use the term *agent* to signify an autonomous piece of code with mobile characteristics, that can be executed independently from its originator.

Agents permit computations to be executed close to the data, which results in a decrease of both the network bandwidth utilization, and the response time. By relaxing the restrictions of permanent location of computations, they promote distribution of workload among the components of a distributed system. Finally, agents support the construction of modular servers capable of adapting to a large variety of client requests. On the other hand, agents raise major problems on their potential hosts. Such problems are related to host's capabilities to provide all the indispensable primitives for supporting an agent's execution, and to the compatibility of agent's requirements with host's policies. These execution primitives and policies are the *execution properties* that fully describe the agent's behavior in the host environment. Accepting an agent without supporting all required execution properties might not be acceptable for the agent's originator, while accepting an agent whose execution properties are in conflict with host's policies might not be acceptable for the host. Hence, before accepting an agent, the host system should be informed of agent's properties, analyze them and decide whether or not the given agent can be properly hosted.

This paper presents a framework, in which a host decides on the acceptance of an agent based on the explicit description of agent's execution properties. Based on means provided by the software architecture field, we describe an agent's execution properties as an open architecture, which abstractly but unambiguously characterizes the host environment expected by the agent. Using this architectural description as the agent's specification, the host is able to detect conflicts with its own policies and to decide whether or not to accept the agent. In case of agent acceptance, the host also has the information to customized the execution environment to meet agent's requirements.

The remainder of the paper is structured as follows: next section gives an overview of the results in the field of software architecture, which we use as a base ground for our proposal. Section 3 addresses the description of open software architectures so as to characterize Web agents with respect to the execution properties that are expected from the host environment. The instantiation of a host environment customized to a given agent, is further addressed in Section 4. Related work is discussed in Section 5, and we conclude in Section 6 by summarizing our contribution.

2. Software Architecture

Research in the software architecture domain aims at reducing costs of developing complex software systems [18, 21]. Towards that goal, formal notations are being provided to describe software architectures, replacing their usual informal description in terms of box-and-line diagrams. These notations are generically referred to as Architecture Description Languages (ADLs). An ADL allows the developer to describe the gross organization of a system in terms of coarse-grained architectural elements, abstracting away their implementation details. Prominent elements of a software architecture are subdivided into the following categories:

- Components, that characterize a unit of computation or a data store.
- Connectors, that characterize a unit of interaction.
- Configurations, that describe a specific, possibly generic, software architecture through the composition of a set of components via connectors.

Existing ADLs differ depending on the software architecture aspects to which they are targeted. We identify at least two research directions, which define different ADLs: *(i)* the architecture analysis, which provides the formal specification of an architecture's behavior (e.g. [14, 10]), and *(ii)* the architecture implementation, which delivers the implementation of an application its architectural description (e.g. [20, 8]). Both research directions instigate the design and implementation of CASE tools, based on technologies such as model checking, theorem proving, and type checking [12].

2.1. Example

To illustrate the use of ADLs, we take as an example a primitive Distributed File System (DFS). The DFS is composed of a client interacting with a (possibly distributed) file server for performing file accesses. We further assume that the interaction protocol between components is RPC-like. Using a simplified version of the ADL used in the Aster project[1], the DFS architecture is given as a configuration made of the client and file server components, which are bound together using a connector describing an RPC protocol. Figure 1 contains the corresponding declarations.

[1] http:/www.irisa.fr/solidor/work/aster.html

```
component client =
  port
    client (typeFormat format);
    server (typeFormat format);
  functional
    typeDsc open (typeStr fn);
    typeInt close (typeDsc fd);
    typeBool read(typeDsc fd,
                  typeBuff buf);
    typeBool write(typeDsc fd,
                   typeBuff buf);
    typeKey key();
    typeStat state();
  interaction
    open, close, read, write: client;
    key, state: server;
  non-functional
    key: Authentication;
    state: CheckPoint;

component fileServer =
  port
    server (typeFormat format);
  functional
    typeDsc open (typeStr fn);
    typeInt close (typeDsc fd);
    typeBool read (typeDsc fd,
                   typeBuff buf);
    typeBool write (typeDsc fd,
                    typeBuff buf);
  interaction
    open, close, read, write: server;
  non-functional
    read, write: FailureAtomicity;
```

```
connector RPC =
  role
    client (typeFormat format);
    server (typeFormat format);
  port
    client (typeFormat format);
  functional
    typeKey key();
    typeStat state();
  interaction
    RPC;
    key, state: client;
  non-functional
    Secure, Reliable;

configuration DFS =
  components
    C: client;
    FS: fileServer;
  connectors
    com: RPC;
  binding
    functional
      C.open: FS.open; C.close: FS.close;
      C.read: FS.read; C.write: FS.write;
      com.key: C.key;
      com.state: C.state;
    interaction
      C.client: com.client;
      com.server: FS.server;
```

Figure 1. The architectural description of the DFS.

The architectural elements described in Figure 1 define the gross organization of the DFS in an abstract manner, and give the associated execution properties. More precisely, the execution properties of the DFS are subdivided into:

- *Functional properties*, which define the operations that are provided and called by the architectural elements. The fact that a functional property is either provided or called by the element declaring it, is given by the semantics of the port through which the corresponding interaction is performed. For instance, the *client* component calls the *open()* operation of the *fileServer* component.
- *Non-functional properties*, which characterize the resource management policies that are provided by the architectural elements. For instance, the *state()* and *key()* operations of the *client* component provide the *CheckPoint* and *Authentication* properties respectively; the *RPC* connector provides *Reliable* and *Secure* communications for all the interactions using it.
- *Interaction properties*, which characterize the communication protocols that are used for performing the interactions among components. In the DFS example, interactions are achieved using an RPC protocol.

Until this point, we have considered the declaration of execution properties in terms of operation signatures and "names" associated to the declared operations (e.g. *key: Authentication*). The type-checking process typically employed in distributed programming environments to verify the correctness of bindings among declared operations is based on pattern matching. However, using pattern matching techniques to verify the satisfaction of the execution properties requirements placed on bindings relies on an informal description of the execution properties. Obviously, we cannot rely on pattern matching performed on "names" describing the execution properties (e.g. *Authentication, RPC*, etc), to verify the correctness of bindings; a slightly different interpretation of the same "name" by each of the interacting sides may cause communication problems difficult to track down and resolve. To overcome this problem, we associate to a "name" a set of formal specifications that serve as its definition. This allows both clean interface declarations and execution properties requirements resolution based on specification matching. The next subsection gives an overview of work done in the area of formally specifying execution properties. For the sake of conciseness, we do not give examples of formal specifications in the following. The interested reader may refer to [7] for more details on this topic.

2.2. Formal specifications of execution properties

Formal specification of functional properties amounts to specifying the behavior of the operations required and provided by the architectural elements. the behavior of operations is specified in terms of *pre*- and *post*-conditions using Hoare's logic (e.g. [17, 23]). In addition to the straightforward benefit of formally specifying functional properties for verifying the correctness of component interconnection, it further favors software reuse and evolution. A component may be retrieved from a component database using the component specification. The correctness of component substitution within a configuration can be checked with respect to the specifications of involved components. The aforementioned verifications lie in the definition of relations over specifications. These relations define, in terms of specification matching, correctness conditions for software interconnection, re-use, and substitution [17, 13, 23].

A non-functional property characterizes a resource management policy that is implemented by the underlying execution platform. Similar to functional properties, non-functional ones are specified in terms of first order logic (e.g. [9]). These specifications refer to operations that are not explicitly stated in the configuration description of a software system, but which are provided by the execution platform or the middleware in a way transparent to the software system. This allows us to specify any non-functional property in terms of a unique predicate instead of pre- and post-conditions. Practically, the formal specification of non-functional properties provide for the verification, with respect to the declared non-functional properties, of the correctness regarding the bindings among components. In addition, it enables the systematic customization of middleware with respect to properties required by the architectural elements (e.g. [8, 24]). Briefly stated, middleware components providing non-functional properties are retrieved in a systematic way through specification matching between required and provided properties. The retrieved components are integrated with the application components using base connectors.

Formal specification of interaction properties has been examined in [2], where a CSP-like notation is introduced for the formal specification of the behaviors of components and connectors with respect to their communication patterns. This allows the correctness verification of a configuration with respect to the communication protocols that are used, using the notion of refinement given in CSP. In that framework, a component description embeds a set of *port* processes that are the component interaction points, and a *coordination* process defining the coordination among ports. The behavior of a component is described by the parallel composition of port and coordination processes. Similarly, a connector is defined in terms of a set of *role* processes, which realize communications among components, and a coordination process that specifies the coordination among roles. The behavior of a connector is described by the parallel composition of role and coordination processes.

3. Web Agent in an Open Software Architecture

A Web agent defines a software component, which interacts with components from the host environment. Each of these interactions is characterized by the functional, non-functional, and interaction properties associated to it. Thus, a software agent can be specified inside an *open software architecture*, which defines a configuration made of the *agent component* and the set of *open components* and connectors that must be provided by the host environment. Then, a host may safely accept and execute an agent if the former's capabilities cover the latter's requirements agent's requirements. The agent's requirements on the host environment can be precisely defined by providing the following information for the elements of the agent's open architecture.

- The agent component defines:
  - the functional properties it expects from the host environment together with the associated interaction properties, and
  - the functional, interaction, and non-functional properties it provides.
- Each connector characterizes the interaction and non-functional properties that are to be made available by the host environment for communication among the agent and the host's components.
- Each open component characterizes a software component with which the agent is willing to interact.

In other words, the agent component abstractly defines the agent's behavior by exposing how it interfaces with the host. The open components and connectors provide a description of the execution properties that should be provided by the host environment.

agent component X =
  – *Same as* client *in* Figure 1 –

open component hostFileServer =
  – *Same as* fileServer *in* Figure 1 –

connector service =
  – *Same as* RPC *in* Figure 1 –

configuration agent-X =
  components agent: X; FS: hostFileServer;
  connectors Svce: service;
  binding – *Same as* DFS **binding** *in* Figure 1 –

Figure 2. Description of an agent accessing a file server in the host environment.

3.1. Example

For illustration, let us give the definition of an open architecture for an agent that accesses a file server in the host system. Figure 2 gives a description of the resulting architecture, which is close to the DFS architecture discussed in Subsection 2.1 except from the *client* component that now corresponds to the agent. Given this description of the open architecture, the host can safely accept and execute the agent, if the host is able to instantiate the **open** *hostFileServer* component and the *service* connector, without violating its own security policies.

3.2. Interpreting agent specifications

So far we have been arguing that architectural descriptions like the one given in Figure 2 provide sufficient information about an agent's requirements and a host's guarantees. Based on this information, one can decide whether an agent conforms with some host's policies, and whether a host can support all the requirements of some agent. To make practical use of this approach, we need to associate agents and hosts with interfaces that describe their execution properties. In addition, hosts must be provided with a framework that allows them to interpret such interfaces. This framework should support the analysis of agent's execution properties, and the reasoning on their combination with host's policies (e.g. see [3] for a study on the combination of properties describing security policies).

Ideally, all three types of execution properties (i.e. functional, non-functional, and interaction properties) should be described using formal specifications. However, common practice has shown that informal specifications provide sufficient guarantees for correct reasoning on functional properties, like in the case of OMG's *Object Transaction Service* (see chapter 16 in [6]). In a similar manner, informal specification has been shown sufficient for interaction properties, due to the well-known and widely accepted interpretations of common communication protocols. In contrast to the above, formal specification is necessary for non-functional properties, since there does not exist a precise common understanding of what they represent. Consider for example the case of an RPC system: designers have a common interpretation of the client-server interaction, but the interpretation of the associated non-functional properties, like the *at-most-once* failure semantics often differ.

Given the architectural description that serves as the specification of an agent, the analysis and reasoning that should be performed by a host prior to the agent's acceptance, are formally defined as follows:

$$
\begin{array}{ll}
\forall C_a \in \mathcal{O}(\text{Agent}): & \forall I_a \in \mathcal{I}(\text{Agent}): \\
\quad \exists C_h \in \mathcal{C}(\text{Host, Agent}) \mid & \quad \exists I_h \in \mathcal{I}(\text{Host, Agent}) \mid \\
\quad\quad \text{MatchFunc}(C_a, C_h) \wedge & \quad\quad \text{MatchFunc}(I_a, I_h) \wedge \\
\quad\quad \text{MatchNonFunc}(C_a, C_h) \wedge & \quad\quad \text{MatchNonFunc}(I_a, I_h) \wedge \\
\quad\quad \text{MatchInter}(C_a, C_h) \wedge & \quad\quad \text{MatchInter}(I_a, I_h) \\
\quad\quad \text{MatchPorts}(P(C_a), P(C_h)) &
\end{array}
$$

The functions and the symbols used in the above expressions, are defined as follows:

- $\mathcal{O}$(Agent) and $\mathcal{C}$(Host, Agent) denote respectively the open components declared by Agent, and the components available by Host to Agent, based on trust issues (i.e. the origin of the agent and the associated level of trust).

- $\mathcal{I}$(Agent) and $\mathcal{I}$(Host, Agent) denote respectively the connectors declared by Agent, and the connectors available by Host to Agent, based on trust issues.

- $P(C)$ denotes the set of ports defined in C.

- MatchPorts(P, P') evaluates to true if the set of ports P (which are requested by the agent) is a subset of the set of ports P' (which are provided by the host). This function can be implemented using pattern matching techniques.

- MatchFunc(C, C') evaluates to true if the functional properties of C' match those of C. This function can be implemented using pattern matching, if functional properties are specified in terms of operation signatures. Otherwise, if functional properties are formally specified, this function can rely on a theorem prover.

- MatchNonFunc(C, C') evaluates to true if the non-functional properties declared by C' match those declared by C. This function should be implemented using a theorem prover, since we have argued that specification matching of non-functional properties is mandatory.

- MatchInter(I, I') evaluates to true if for each interaction property in I there is at least one interaction property in I' that matches it. Similarly to the functional properties, this function can be implemented using pattern matching. However, for greater robustness and flexibility formal specifications of interaction properties should be employed. For instance, this may be achieved using a CSP-based process algebra as proposed in [2]. Then, interaction properties match if the processes declared in I' refine those in I. Let us remark here that the matching function may be automated using a tool like FDR [5].

4. Instantiating Customized Hosts for Web Agents

Given an agent's specification, we have seen how the host environment can analyze the agent's requirements and reason on their compatibility with host's policies. In this section we report on the work we have carried out on using the outcome of this analysis to instantiate an execution environment customized to the agent's needs. The instantiation process is based on searching and retrieving host's architectural elements that match the open components and connectors declared by an agent. In addition to the execution properties that have been mentioned thus far, we explicitly take under consideration the usage of host resources requested by an agent, so as to guarantee that the agent will be able to execute to completion [19]. For that purpose, we include an additional clause, named **resource**, in the configuration description, in which the resources required for the agent's execution are declared. Figure 3 gives an example of an explicit declaration of agent requirements concerning host's resources.

configuration agent-X =
  components
    agent: X;
    FS: hostFileServer;
  connectors
    Svce: service;
  binding
    – *Same as* DFS **binding** *in* Figure 1 –
  resource
    MEMORY = 15KB;
    DISK = 40KB

Figure 3. Explicit declaration of agent's requirements regarding host's resources.

4.1. Customizing the host environment

Upon the reception of an agent specification, the host first evaluates whether it is able to execute the agent based on the agent's originator and on available architectural elements and resources. If so, the host notifies the agent's originator, which then sends the agent's code. If the host does not accept the agent, the reason of rejection is sent to agent's originator. Possible reasons include: insufficient level of trust for performing the requested operations, unavailability of architectural elements, and unavailability of some resources. The agent's originator may then revise its initial requirements by modifying the open software architecture constraints, and make a new request to the host.

Considering the agent's specification, instead of treating it as a hostile entity facilitates its acceptance by the host. However, until now, we have not considered issues related to the safety of the host, which in the Web community is considered much more important than the acceptance of an agent. By accepting an agent relying on the declared properties, we risk to accept an agent that actually exhibits a different behavior than the one it declares. In that case, the agent may cause damages to the execution environment. Obviously, for our approach to be viable, situations like the above should never raise. To assure this, the execution environment for a given agent is built by the host according to the agent's specification, which describe the exact interaction points with the host and their properties. Except from the allocated resources (i.e. private memory and disk space) which the agent can access without any restrictions, the only way for an agent to access the host is to pass through the declared bindings conforming to the associated execution properties. Hence, the execution environment is safe for the host since it does not allow the agent to perform any actions other than those declared in its interface.

Although the host guarantees the correct execution of an agent whose behavior conforms with the one declared in its interface, no guarantees exist for the execution of agents whose behaviors deviate from the declared ones. In some cases the customized environment may provide "stronger" execution properties than those requested by an agent. This may occur under two conditions: *(i)* if the host does not possess a component providing exactly the property requested by the agent, but it does possess a component providing a "stronger" property, and *(ii)* if the agent is still accepted by the host when the stronger property replaces the originally requested one in the agent's specification. Such cases include the allocation of a bigger portion of resources than the one required, the support for 32-bit encryption keys while only 16-bit keys were requested, the use of a component that implements failure atomicity and retry-on-error while only the first was requested, etc. Hence, an agent that requested 15.5KB of memory but actually uses 16KB may finally execute to completion, although this is not a priori guaranteed.

4.2. Prototype implementation

To experiment the practicality of our approach, we implemented a prototype for agents written in the Java programming language. The prototype is a client-server system, where clients contact the server through CGI to request the remote execution of one of their agents. The client-server interaction is decomposed as follows:

- The client first sends the agent's specification to the server, and waits for approval or rejection notification from the server.

- Upon reception of an agent's specification, the server checks whether it is able to host the agent based on available architectural elements and resources, as presented in the previous subsection. Architectural elements available on the host are stored in a software repository, which is organized so as to be able to identify the subset of elements that can safely be made available to an agent with respect to its originator. In the current prototype, we distinguish between two kinds of agents: those originated from the same Intranet to which the server belongs, and those originate outside this Intranet. Availability of architectural elements simply relies on pattern matching for functional and interaction properties. On the other hand, checks regarding non-functional properties are done using specification matching and relies on the tool we developed for middleware customization [8]. If the server can safely execute the agent, it computes a unique key for it and reserves the requested resources; the key is sent to the client together with the acceptance notification. If the agent cannot be run, the reason for rejection is notified to the client.

- Once the client receives the notification message from the server, it checks whether the agent has been accepted or not. In the former case, it sends the agent's code to the server together with the associated key. In the latter case, the client re-issues the requests later on, if the reason of rejection is the temporary resource unavailability. The client may issue a new agent specification if the reason of rejection is some other execution property.

- Upon the reception of the agent's code, the server checks the agent's identity using the associated key. Once the agent is authenticated, it runs in the customized execution environment, which only allows the agent to behave in the way that was described in its specification. In addition, the customized execution environment provides a private address space for the agent's execution, in order to confine the consequences of an agent's crash on the failed agent alone. To assure that an agent respects the execution properties it has declared in its interface, we have used the *SecurityManager* Java class to build the *AgentSecurityManager*. The *AgentSecurityManager* surveys the execution of an agent in terms of file system and network accesses, and resource consumption, and causes an agent to abort if the agent attempts some unauthorized action.

The prototype is sufficient for evaluating the practicality of our approach, but it needs to be enhanced from the standpoint of performance and scalability of the host instantiation process. The software repository of architectural elements available on the host is actually composed of a small set of elements, enabling sequential search. Further work is needed so as to experiment with a large software repository. In addition, our prototype implementation would obviously benefit from enhanced products based on the Java technology. Products like SUN's Jini[2] may provide for substantial improvements to our prototype.

[2] http://java.sun.com/products/jini

5. Related Work

The implicit host-agent interaction model that underlies our approach is similar to the conventional Web agent interaction model, where the agent describes a process and the host provides an execution environment (e.g. *HTTP-based Mobile Agents* [11]). Yet, in our case the execution environment is prepared by the host according to the agent's execution properties, which implies that the execution environment is explicitly defined by the agent, and that an agent and its execution environment are strongly coupled. From this standpoint our Web agents resemble *Mobile Ambients* [4], which define the mobile code and its bounded execution environment as a single entity that can move across Web's administrative domains. However, a mobile ambient is a modeling entity used as the structural element of a system model described in the ambient calculus, whereas our Web agent is a set of execution properties describing the behavior of a piece of software. The conceptual similarity of mobile ambients and our Web agents suggests that the ambient calculus can be employed to provide a formal model of the host-agent interactions described in this paper.

At the practical level, our prototype deals with the safety problems stemming from inappropriate resource use by mapping an agent to an individual process. Although this approach assures that problems like memory address violation will affect only the execution of the agent that caused them, it restricts severely the number of agents that can execute concurrently in a given host. A different approach is suggested by *Proof-Carrying Code*, or PCC for short, which has been used to support safe execution of mobile agents [15]. In PCC agents carry a proof that they conform to the host's policies and the host is capable of verifying the validity of this proof. Hence, agents can be safely mapped to threads which share their address space with other agents or even host threads. In this sense, PCC provides an elegant alternative to the heavy hosting scheme used by our prototype, for the extra cost of building a proof at the agent's originator and verifying its validity at the host. However, PCC does not provide any support for nonfunctional execution properties, nor support of any type for the agent's requirements from the host. Hence, it is not directly comparable to our approach. Rather, it should be considered as a alternative approach for resource management in prototype.

In general, our approach for hosting Web agents does not aim at replacing existing environments supporting mobile agents. Instead, it aims at providing a unifying way for describing agents' behavior in terms of their functional and nonfunctional execution properties, and considering their hosting based on that behavior. In our framework, issues related to code mobility and its remote execution in foreign hosts can be uniformly expressed as execution properties. Execution properties integrate agent characteristics regarding both mobility support primitives (e.g. *attach, move,* and *clone* [11]), and non-functional requirements (e.g. *authentication, availability, secrecy*, and *integrity* [22]). Hence, approaches like *Mobile Assistant Programming* [16] that provides the underlying framework for creating agents and moving them to different hosts, can gain significant benefits from the presented approach, in terms of flexible agent hosting.

6. Conclusion

In this paper, we have proposed a framework for reasoning on the acceptance of agents and for constructing execution environments customized to agents' requirements. The proposed framework conveniently adapts existing technology from the fields of distributed systems and formal specifications to the needs of Web agents. Our approach is based on the description of open software architectures characterizing the execution properties provided by agents and expected from the host environment. The proposed framework suggests that hosting an agent consists of verifying the compatibility of the agent's requirements with the host's policies, and then customizing the host environment to meet the approved requirements. Our proposal is easy to use in the simple case (i.e. the interface declaration for agents that do not have any non-functional requirements is equally easy to the declaration of a CORBA interface), while it supports the declaration of complex execution properties without sacrificing the functionality, performance, or safety of the host (i.e. the customized execution environment guarantees both agent's requirements and host's constraints).

Besides the benefits for the agent, the presented approach provides support for modular constructions of host environments, which results in flexible and scalable Web servers. The customization of the agent's execution environment permits the use of resource management algorithms which allow the host to concurrently serve more than one agent without significant impacts on its performance. Moreover, modular constructions allow the Web servers to apply different hosting policies, according to various criteria based on agent characteristics. As a consequence, a number of other research issues can be addressed in the Web agent framework, including how a host can use its current state for deciding to accept an agent, how to advertise, dispose of, and cost its resources, what combinations of non-functional properties should the host accept, how to assure fair treatment of agents with equivalent requirements, etc.

References

1. ACM Press. Intelligent Agents. *Communications of the ACM*, 37(7), July 1994.
2. R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.
3. C. Bidan and V. Issarny. Dealing with Multi-Policy Security in Large Open Distributed Systems. In *Proceedings of the 5th European Symposium on Research in Computer Security*, pages 51–66. LNCS 1485, Springer-Verlag, September 1998.
4. L. Cardelli and A. D. Gordon. Mobile Ambients. In *Proceedings of the 1st International Conference on the Foundations of Software Science and Computational Structures*, pages 140–155. LNCS 1378, Springer-Verlag, MArch 1998.
5. FSE Document. Failures Divergence Refinement: User Manual and Tutorial, 1.3 edition. Technical report, Formal Systems (Europe) Ltd., Oxford, England, 1993.
6. OMG Document. CORBAservices: Common Object Services Specification. Technical report, Object Management Group, November 1997. Available at http://www.omg.org/corba/csindx.htm.
7. V. Issarny. Configuration-Based Programming Systems. In *Proceedings of SOFSEM'97: Theory and Practice of Informatics*, pages 183–200, November 1997.
8. V. Issarny, C. Bidan, and T. Saridakis. Achieving Middleware Customization in a Configuration-based Development Environment: Experience with the Aster Prototype. In *Proceedings of the Fourth International Conference on Configurable Distributed Systems*, pages 275–283, May 1998.
9. V. Issarny, C. Bidan, and T. Saridakis. Characterizing Coordination Architectures According to Their Non-Functional Execution Properties. In *Proceedings of the Thirty First Hawaii International Conference on System Sciences*, pages 275–285, January 1998.
10. J. Kramer and J. Magee. Exposing the Skeleton in the Coordination Closet. In *Proceedings of COORDINATION'97: Coordination Languages and Models*, pages 18–31. LNCS 1282, Springer-Verlag, September 1997.
11. A. Lingnau, O. Drobnik, and P. Dömel. An HTTP-based Infrastructure for Mobile Agents. In *Proceedings of the 4th International World Wide Web Conference*, pages 461–472, December 1995.
12. N. Medvidovic and R. Taylor. A Framework for Classifying and Comparing Architecture Description Languages. In *Proceedings of the Joint 6th European Software Engineering Conference - 5th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 60–76, September 1997.
13. R. Mili, R. Mittermeir, and A. Mili. Storing and Retrieving Software Components: A Refinement Based System. *IEEE Transactions on Software Engineering*, 23(7):445–460, July 1997.

14. M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, 21(4):356–372, April 1995.
15. G. C. Necula and P. Lee. Safe, Untrusted Agents using Proof-Carrying Code. In *Mobile Agents and Security.* LNCS 1419, Springer-Verlag, December 1998.
16. S. Perret and A. Duda. Mobile Assistant Programming for Efficient Information Access on the WWW. In *Proceedings of the Fifth International World Wide Web Conference*, pages 1373–1384, May 1996.
17. D. E. Perry. The Inscape Environment. In *Proceedings of the 11th International Conference on Software Engineering*, pages 2–12, May 1989.
18. D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.
19. T. Saridakis, V. Issarny, and C. Bidan. Customized Remote Execution of Web Agents. In *Proceedings of the 31st Hawaii International Conference on System Science*, pages 614–620, January 1998.
20. M. Shaw, R. DeLine, D. Kelin, T. Ross, D. Young, and G. Zelesnik. Abstraction for Software Architectures and Tools to Support Them. *IEEE Transaction on Software Engineering*, 21(4):314–335, April 1995.
21. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1996.
22. J. W. Stamos and D. K. Gifford. Remote Evaluation. *ACM Transaction on Programming Languages and Systems*, 12(4):537–565, October 1990.
23. A. M. Zaremski and J. M. Wing. Specification Matching of Software Components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, October 1997.
24. A. Zarras and V. Issarny. A Framework for Systematic Synthesis of Transactional Middleware. In *Proceedings of Middleware98 – IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 257–274, September 1998.