Dynamic Assembly & Integration on Component-based Systems*

Andres Flores[1] and Macario Polo[2]

[1] Departamento de Ciencias de la Computación, Universidad Nacional del Comahue, Buenos Aires 1400, 3400 Neuquén, Argentina. Email: aflores@uncoma.edu.ar
[2] Escuela Superior de Informática, Universidad de Castilla-La Mancha, Paseo de la Universidad 4, Ciudad Real, España. Email: macario.polo@uclm.es

Abstract. Integration implies a primary task on component-based development. From a system architecture, off-the-shelf components are selected to satisfy functionality. A general integration process may include component '*qualification*', '*adaptation*', and '*composition*'. We are currently interested on automation of the qualification phase. Our intent is to 'integrate' into an infrastructure a '*component evaluation process*'. Thus applications could be dynamically created from component assembly and then integrated into the underlying infrastructure. In this paper we also report other approaches that fulfil some of that general phases, and we highlight methods, and techniques that were applied. Since the quality of a product depends on the quality of a process, we look for achieving a sound integration process to achieve reliability on component-based systems.

Keywords. Component-based Software Engineering

1 Introduction

Component-based software development (CBSD) involves a mixed process of both top-down decomposition and bottom-up composition, where the major effort is focused on composition techniques rather than component development [4, 13]. Hence an integration process is the primary task performed by designers as opposite to usually be the tail-end of an implementation effort. A strong basis on reusability is clearly stated as components are ready "off-the-shelf", whether from a commercial source (COTS) or reused from another system. They must be integrated with other components to achieve the required system functionality, though they are used "as they are found" instead of being modified [3].

Component integration is vital to a component selection process, and a major consideration in the decision to acquire or build the components. A general integration process (or reference model) could be conceptualised by certain phases which may be identified as *qualification*, *adaptation* and *composition* of

* This work is partially supported by the CyTED project VII-J-RITOS2, the UNComa project 04/E048, and by the MAS project supported by the Ministerio de Ciencia y Tecnologa (TIC 2003-02737-C02-02).

components [3]. Some approaches have focused on such activities mainly at a development stage by applying manual or semi-automatic techniques [4].

We are interested on automating such phases, and we are currently working upon the qualification phase. Our intent is to 'integrate' into an infrastructure a *component evaluation process* from where components could be analysed based on system requirements and a given software architecture. Basically such process implies to analyse compatibility between a component and certain expected aspects. Not only services' signatures should match properly, but also the component assertions. Further, an important factor that may certify a degree of compatibility is to analyse whether there is a matching on the components underlying contexts. Thus concepts on different aspects of a component are analysed considering the actual meaning which may change from one domain to another [11].

In this paper we report different approaches that afford some of the phases on the general integration process, and highlight methods and techniques that were applied. The purpose is to describe the range of different possibilities which may address the same goals, and identify commonalities which may finally stand as standard practices due to their promising benefits.

Thus our approach as well as others will be confronted against the reference model which is introduced on the following section. On each phase we describe the variety of techniques to be used. Section 3 then presents our approach as well as the others in order to distil both uncovered activities and applied methods or techniques. Finally conclusions are presented in section 4.

2 Reference Model for Integration Process

The engineering of CBSD could be considered to be primarily an assembly and integration process. This suggests a reference model as is presented in [3] for describing the engineering practices involved in assembling component-based systems (CBS). The vertical partitions on the reference model, depicted in Figure 1, describe the central artefact of CBSD - the components - in various states or phases. An overview of the shapes adopted by components across the phases is following presented. Then the phases on the reference model will be discussed by describing the main arising issues as well as useful techniques to be applied.

– *Off-the-shelf* (OTS): have hidden interfaces (by a definition that cover all potential interactions among components and not just an API).
– *Qualified*: have discovered interfaces so that conflicts and overlaps could be identified. This implies a partial discovery: only interfaces relevant to effective component assembly are identified.
– *Adapted*: have been amended to address potential sources of conflict. This may imply a kind of component "wrapping".
– *Assembled*: have been integrated into an architectural infrastructure. This infrastructure will support component assembly and coordination.

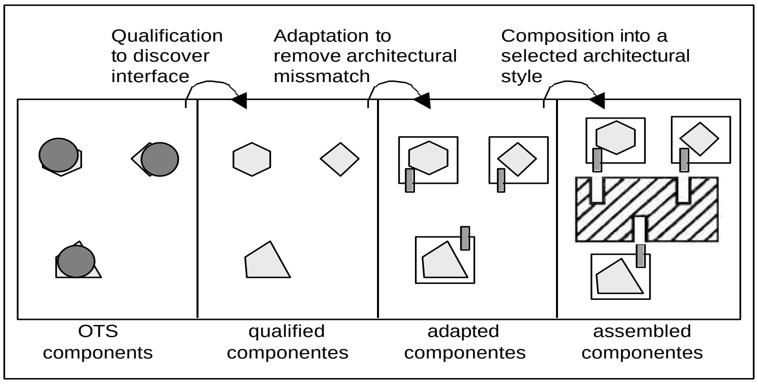
Qualification to discover interface
Adaptation to remove architectural missmatch
Composition into a selected architectural style
OTS components
qualified componentes
adapted componentes
assembled componentes

Fig. 1. Reference Model for Architectural Component Integration

2.1 Qualification

Typically, a component is described in terms of an interface that reflects the component functionality. However, in a much broader interpretation the interface to a component includes far more than the functionality it provides. We need to understand aspects like performance, reliability, usability, and so on [3].

Hence, for most components there are many unknowns, which despite of an attempt to do a discovery through a usual manual evaluation, a number of gaps will remain [10]. Although some work has been done about the qualification of software components, it mostly implies manual or semi-automatic tasks. As there is little agreement on which attributes of a component could be critical, to think on this process being dynamically performed is quite difficult.

Qualification of a component can also extend to a qualification of the development process used to create and maintain a component. Contractors nowadays should perform their practice according to the Capability Maturity Model (CMM) and the current ad-hoc standards [3, 21]. This ensures that the software components they produce have been developed using well-defined practices and procedures.

2.2 Adaptation

Building a system from stand-alone components creates serious conflicts since components must cooperate and architectural mismatch may often occur. The reason is the wide variety of conflicting operating assumptions made by each component [3]. Hence components must be adapted by understanding their purpose and services under the basis of their original context of use. A component that presents similar properties could not fit on the ongoing project due to a context mismatch. The component context provides the means to clear the sense or scope of its information, that is, the actual meaning behind every aspect.

Component adaptation usually involves some form of "wrapping" - locally developed code that provides an encapsulation of the component to mask unwanted and incompatible behaviour. An *insulation* is a special wrapper which provides a unique point of access (a single API) for multiple interfaces (like ODBC for example). Another kind of wrapper is called *instrumentation*, and its purpose is to instrument, debug, or add assertions. They can assist with testing and exploration for side effects [19].

Wrapping is not the only approach to overcoming component incompatibilities. Another approach is through the use of mediators, which informally can be seen as an active agent that coordinates between different interpretations of the same system property. For example, a mediator may contain filters or other agents to convert (translate) between data formats, establish common events, or define common administrative policies (e.g., for security and access control) [3].

2.3 Assembly

Assembled components are integrated through some well-defined infrastructure, which provides the binding that forms a system from disparate components. At the highest abstract level the infrastructure embodies a coordination model that defines how components will interact to perform the required end-user functionality. That is to allow this coordination model to be readily described, validated, and enacted. At a lower level it provides services that will be used by components to interact and to carry out common tasks. The interface to these services must be complete and consistent. At the most practical level the infrastructure is itself a software component that implements the required coordination services. It must be well-written to be easily understood, perform effectively, and be readily updated to new modes of component interaction.

Currently, most active research makes use of messaging systems as infrastructure providers, particularly using object request brokers conforming to those architectures which has been well accepted such as CORBA, DCOM and SUN J2EE. Another important and highly relevant research subject addresses Web Services, where the core of communication - the infrastructures - are developed upon the Internet. The standard architecture is SOA from where some industry approaches have been developed (IBM WebSphere, Microsoft .Net, SUN ONE, etc) [22]. The contribution to CBSD is particularly on distributed systems. In general, reports cite a number of advantages to their use in building CBS as can be seen in section 3 where current approaches will be presented.

3 Current Approaches

This section presents different approaches which implement the general phases from the reference model (Figure 1) presented on section 2. Most of them expose methods that deal with specific mechanisms to uncover particular phases. Following our approach as well as a set of other current approaches are briefly described. For each of them, methods and techniques will be distilled according to the reference model. Thus we may appreciate available implemented possibilities to carry out the same mechanisms and consider their application when a component-based architecture is being developed.

3.1 Our Approach

Based on our interest on Pervasive Computing Environments (PvCEnv) we are currently developing mechanisms to carry out applications assembly at run-time. Most approaches on this particular area are based on externally created applications, which are developed ad-hoc for such environments (as can be seen in [20]). This could make a user feel limited to a constrained range of applications. Users working on PvCEnv expect not only efficacy but also availability and usability on their location independent tasks [8, 12]. Using a wider range of OTS components may help to build a wider range of applications as well [13].

To such intent we are currently focused on supplying a procedure to be able to evaluate components which will be used for assembly of an application. Since a user changes quite often from one operational context to another, requirements must be updated and properly satisfied by selected components. Such components may not be previously evaluated, e.g when they are embedded into a chosen disparate device, so evaluation must be thoroughly done [11, 13].

Components cooperating with others to form another software entity - a new application - outline an interoperability pattern. This implies a particularly complex approach which involves many aspects to be concerned of [6]. Different levels of interoperability have been distinguished [9, 13], e.g. syntactic aspects were undertaken in [17]. Our approach is set on the semantic level since components to be evaluated provide information which must be carefully analysed.

As OTS components are created to be placed on a given context of use, we must assure such context is similar to the one where the component will be inserted. Concepts recognized from component collected information (e.g. from its interface and meta-data) need to be properly interpreted. Hence we propose the use of an Ontology which may give the necessary compatibility under a comparison of the contexts both from the component under evaluation, and the user task and location. An ontology could be placed for each relevant context so to easy the whole evaluation. We intend to add meta-methods to components with detailed information concerning pre and post conditions of both required and evaluated services, as well as invariants of the component. This implies a wrapper mechanism (instrumentation).

Hence we are designing an abstract model of an infrastructure for semantic interoperability, where we intend to 'integrate' a *component evaluation process*. This is accomplished by the application of testing strategies based on contextual information of components and users' tasks. Figure 2 presents such an abstract model of a PvCEnv as a generic view for our solution: an *Infrastructure for Semantic Interoperability* [11]. In fact we are exploring other complementary mechanisms which may enforce this proposal for certification of compatibility.

We also provide a formal basis to our framework by explicitly reflecting the dynamism in the component interaction by a Propositional Linear Temporal Logic [7] based specification. Thus different testing strategies can be more precisely defined and applied. The use of a formal layer as the basis for this work facilitates the definition of a sound procedure to automate testing of semantic interoperability.

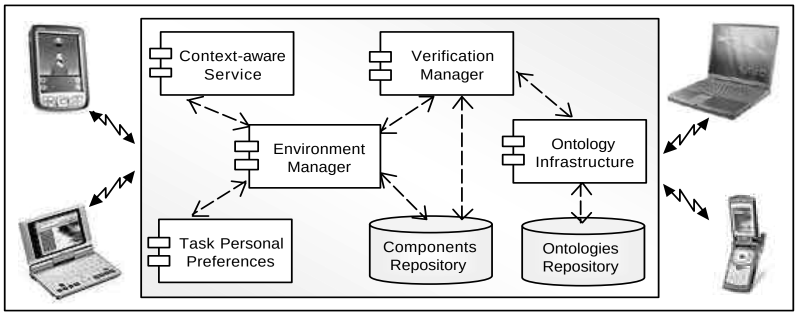

Fig. 2. Infrastructure for Semantic Interoperability

Indeed our intent may seem to be a solution on a regular CBS, though we believe it mostly fits to a PvCEnv as a fair justification. As part of the current project we plan to provide a complete automation of the whole integration process according to the reference model.

3.2 CoCon

In [18] is presented **CoCon** (context-based constraint), an infrastructure that we understand allows a dynamic stability management on a CBS. The approach proposes to integrate into the system architecture a 'Rule Manager to automatically monitor the system's compliance with requirements at runtime. Thus it suits with the paradigm of *Continuous Software Engineering* discussed in [23].

When adapting a CBS to new, altered or deleted requirements, the existing ones should not accidentally be violated. Validation of conformity with requirements is enforced externally. Hence, the system can be transparently adapted via connector refinement without modifying components directly. Thus any OTS or legacy component can be used.

Context is essential to use a component information properly. Meta-data, called *context properties*, is attached to the components as 'yellow sticky notes, which can be seen as a wrapper (instrumentation). A constraint mechanism refers to this meta-data to identify that part of the system where a constraint applies. Only components whose meta-data fits the constraint's 'context condition must satisfy it. Thus a requirement for a cluster of components that share a given context can be automatically protected.

Communication between components is captured by *points of interception* installed in the components communication paths. A proxy [14] mechanism is used, which is rather common on middleware technologies. The proxy is extended by a monitoring functionality – as shown in Figure 3 – and as such can be seen as a *connector* which facilitates requirement protection.

It is clear that this approach fits the adaptation and assembly phases on the reference model, which are achieved by means of active mechanisms incorporated into an underlying middleware or infrastructure. However qualification seems not being considered since component selection was omitted. The reason may be their focus on dynamic verification of validity of requirements.

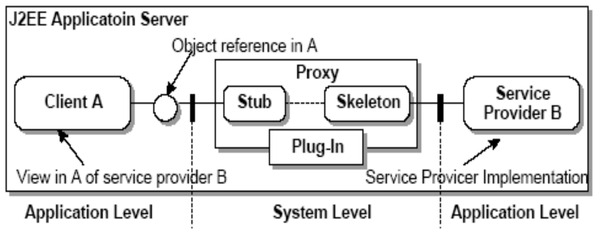

Fig. 3. Proxy Mechanism on CoCon approach

3.3 Hadas

The approach in [2] proposes the framework **Hadas**, which provides dynamic deployment and adaptation of components. It is particularly helpful for distributed applications which must be integrated at runtime as generally occurs with web sites. Hadas provides an adaptability mechanism based on a *Mutable Reflective Component Model*, and thus components support introspection by means of meta-methods which give information about both behaviour and structure of a component. Another approach of such model is the Reflection API in Java 1.1 [15]. This facilitates external modification and self-adaptability.

This allows to wrapper remote sites in order to be treated as components, and also to make easy updating a component as the remote site changes its interface. Thus such wrapper component acts as an agent, which monitors the conditions under which it may execute. Figure 4 depicts the way two remote sites may be composed by the Hadas model. At the bottom lays the infrastructure that makes possible all the mechanisms that can be applied on the integration of distributed applications.

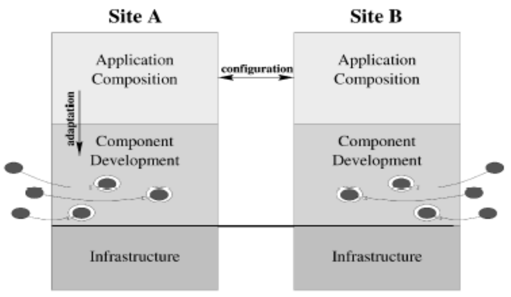

Fig. 4. Hadas Model

A mechanism called *Ambassador*, which is a representative of a component, is used when components (other than sites) are needed on different remote points. Its purpose is to serve as an adaptive remote reference (a Remote Proxy) [14] and also as an interoperability handler to transform data from/to invoker/receiver components (wrapper behaviour). As they possess the same characteristics as components', this allows easy application adaptability. A component and all its representative Ambassadors are similar to the *Observer* pattern [14], whereby the component fills the *Subject* role and the Ambassadors act as *Observers*.

Dynamic composition is based on a set of protocols that are issued between components. The central concept in this model is the Ambassador which is dynamically deployed and can be dynamically tailored according to negotiations and imported protocols. Thus applications can be dynamically composed and reconfigured without affecting components as well as individual components can evolve without affecting the application.

From the point of view of the reference model we can see that this approach provides mechanisms for Adaptation, and Assembly phases. Whether indeed every component can be used (by the wrappering technique), and these components do not need to be modified to fit into the infrastructure, there is no explanation of a technique to select appropriate sites or components.

3.4 BASE

An approach for a PvCEnv is presented in [1], where an infrastructure for adaptable applications is being developed. Currently the micro-broker based middleware BASE has been deployed, and it provides abstractions for applications deployment and communication. It was developed as a minimal platform suitable for small-embedded systems but extensible to make use of abstractions available on resource-rich environments. Objectives are: to support device lookup, service discovery, flexible protocol support and selection, decoupling of application and interoperability communication models, uniform abstraction for device capabilities and services, flexible integration of adaptation mechanisms.

A particular framework will be located on top of BASE, which provides additional mechanisms to enable an application to be dynamically adapted in order to react to changes on availability of services or device capabilities according to the current application execution policy. PCOM is the application model which specifies the architectural building blocks (components) and their interdependencies (contracts between components). At runtime, this specification is mapped to a concrete set of component instances where all mandatory contracts are fulfilled. A contract concept is used to specify required properties for component interaction and also to indicate application configurations. Applications are activated when all of their contracts are fulfilled. Contract enforcement and mechanisms for adaptation are provided by BASE.

On top of PCOM applications can be modelled, and for this a special component (called *Anchor*) is used to specify the set of necessary sub-components. These components depend on each other according to the specified contracts, and the Anchor component provides special contract specifications to complete the sense or scope of such application. Figure 5 shows an example of a health monitoring application which provides information and advices whenever a suitable display is in the vicinity.

An *invocation object* can be used for an application to choose different communication models (RPC, deferred synchronous RPCs, or events via stub objects). This mechanism allows different interoperability protocols become possible. It can be created manually or through a proxy if a service provides a stub object. BASE is responsible for synchronizing the caller and issue invocations, and receives possible replies as well as an invocation.

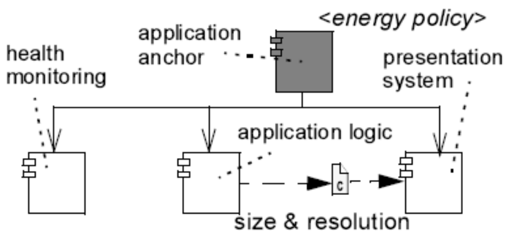

Fig. 5. Application assembly by means of an Anchor - BASE micro-broker

It seems that no other than components developed under the PCOM model can be used in the environment. Thus the *qualification* phase does not apply here. *Adaptation* is achieved at the application level, instead of simpler components, and *assembly* is performed both for an application definition and for communication between different applications (through invocation objects).

3.5 KX Approach

An approach to autonomizing legacy systems is presented in [16], with the purpose to assembling 'autonomic' systems-of-systems from components. The current implementation is called Kinesthetics eXtreme, or KX ('kicks'). KX is being applied experimentally for load balancing and server replacement for Telecom Italia's heterogeneous instant messaging system, and for more complex contingencies in a geographically-based *open information* (e.g., CNN, BBC) analysis system and in use at the US Pacific Command (PACOM).

Inserting adaptation mechanisms into existing application code is difficult, error-prone and hard to reuse or reason about, hence an external adaptation solution was proposed. Upon the objective of automatic reconfiguration and control of the running system was proposed a three-tiered infrastructure. Data is collected from the running system by non-invasive *probes* that report raw data to the higher levels via a *Probe Bus*. Then data is interpreted via a set of *gauges* that maps the probe data into various models of the system. The gauges then report their findings to the *Gauge Bus*.

For both local adaptations and global reconfigurations in the running system, the *Workflakes* decentralized workflow system has been proposed (see Figure 6). Workflakes coordinates the actual reconfigurations workflow through deployment/activation of low-level software *effectors* attached to the target system.

Here are analysed the implications of the interpreted data on the overall system performance and make decisions on whether to: (1) introduce or disable gauges in the interpretation layer; (2) deploy new probes to provide more detailed information to remaining gauges, or turn some off to reduce *noise*; and/or

(3) reconfigure the system itself, perhaps changing the system's structure by introducing new modules or modifying system or component parameters.

Any analysis is based on measurements in terms of available models, which then leads to the feedback phase where runtime modifications to the system are (automatically) carried out. Thus it may help automate most system management functions with little or no human intervention, whereby the adaptation of the running system can be carried out dynamically without incurring any system downtime. Whether the infrastructure is largely independent of the running system, probes and effectors must often be specialized to the implementation technology. In addition, gauges and decision mechanisms must be specialized to the contextual problem.

Probes output data has been structured by an XML Schema as a standard format. A 'Smart Event' Schema includes points of extension with generic application models so that arbitrary gauge technologies can use this information to determine selection of probes. Thus probe descriptions do not need customisation for a given application or its specific models. The initial probe might emit simple Smart Events containing a raw data block, but later processing and analysis stages augment it with additional or higher-level information blocks.

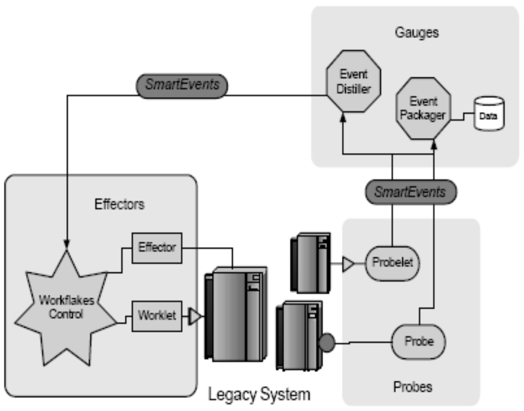

Fig. 6. Dynamic Monitoring of Legacy Systems - KX's implementation

Gauges are intended to gather, filter, aggregate, compute, and/or analyse measurement information about software systems. Particularly, they interpret probe data against various models, to produce higher-level outputs and emit events just as probes. Such events are typically quite abstract, but the Smart Event XML Schema has been defined to support both levels. As with probes, a major concern has been developing standards for gauges to allow interoperability.

The proposed gauges work within a framework called XML-based Universal Event Service (XUES) (Figure 6). XUES involves two major components: the Event Packager transforms raw-data format of legacy probe output into Smart Event compatible event streams. It also packages and logs these events for possible replaying. The Event Distiller can recognize complex temporal event patterns from multiple probe sources, and constructs high-level measurements to reflect the system state. It also produces events to interface with the decision layer and gauge visualizers. It encloses a collection of condition/action rules, where conditions specify the event pattern and actions specify what to do when that pattern is recognized - typically generation of a higher-level event.

As we see in this approach both the *adaptation* and *assembly* phases are thoroughly covered. Since the main intention is to apply this approach on existing legacy systems, no methods for assessment and selection of components are provided. Hence, the approach does not cover the *qualification* phase.

4 Conclusions

Indeed CBSD is one of the most popular approaches in software production. Moreover, component-based technologies simplify functional decomposition of complex systems and support building of re-configurable compositions and tuning of component compositions for the particular context they are used in [10]. However there are several aspects that must be considered on this paradigm. Components may have different characteristics, which must be properly analysed according to different pre-requisites: quality attributes, quality models, different standards, and so on [4, 6].

Since quality of a product depends on quality of a process [21], there is a need to distinguish some stages in the whole process of CBSD. Thus in section 2 was presented a reference model for integration of components. In addition some issues were also discussed concerning their comprising phases, where not only considerations about a component in isolation are relevant, but also the composition with other components or the integration with an underlying infrastructure.

In section 3 we have presented some of the current approaches - including our own work - which reveal the successful application of different methods, mechanisms and techniques to uncover some of the phases of the reference model. Indeed there are some approaches which are mainly focused on component classification and recovery [5], though there is a lack in dynamic evaluation or qualification of components. We understand that this is not a trivial work, though the need of a sound procedure is highly important. When systems must be dynamically adapted from available components an evaluation for functional and non-functional compatibility should be accomplished at a reliable standard.

References

1. C. Becker and G. Schiele. Middleware and Application Adaptation Requirements and Their Support in Pervasive Computing. In *23rd IEEE ICDCSW'03*, pages 98–103, MProvidence, Rhode Island, USA, May 2003.
2. I. Ben-Shaul, O. Holder, and B. Lavva. Dynamic Adaptation and Deployment of Distributed Components In Hadas. *IEEE T-SE*, 27(9):769–787, September 2001.
3. A. Brown and K. Wallnau. Engineering of Component-Based Systems. In *2nd IEEE ICECCS'96*, pages 414–422, Montreal, Canada, October 1996.

4. A. Cechich, M. Piattini, and A. Vallecillo. *Component-based Software Quality: Methods and Techniques*, volume 2693 of *LNCS*. Springer-Verlag, 2003.
5. E. Damiani, M. Fugini, and C. Bellettini. Corrigenda: a hierarchy-aware approach to faceted classification of object-oriented components. *ACM TOSEM*, 8(4), 1999.
6. L. Davis, R. Gamble, and J. Payton. The impact of Component Architectures on Interoperability. *Journal of Systems and Software*, 61(1):31–45, March 2002.
7. B. Berard et.al. *Systems and Software Verification (Model Checking Techniques and Tools)*. Springer-Verlag, 1999.
8. D. Garlan et.al. Software Architecture-based Adaptation for Pervasive Systems. In *ARCS'02*, volume 2299 of *LNCS*, pages 67–82, Karlsruhe, Germany, April 2002.
9. J. Euzenat. Towards a principled approach to Semantic Interoperability. In *Wrkshp on Ontologies and Inf. Sharing, (IJCAI'01)*, pages 19–25, Seattle, US, August 2001.
10. A. Fioukov, E. Eskenazi, D. Hammer, and M. Chaudron. Evaluation of Static Properties for Component-Based Architectures. In *28th IEEE EUROMICRO'02*, pages 33–39, Dortmund, Germany, September 2002.
11. A. Flores, J. C. Augusto, M. Polo, and M. Varea. Towards Context-aware Testing for Semantic Interoperability on PvC Environments. In *IEEE SMC'04, special session: CRIPUC*, The Hague, Netherlands, October 2004. To be published.
12. A. Flores and A. Cechich. Quality Considerations on Ubiquitous Systems. In *II Workshop de Ingeniería de Software (JCC'02)*, Copiapo, Chile, November 2002.
13. A. Flores and M. Polo Usaola. Considerations upon Interoperability on Pervasive Computing Environments. In *6th WICC*, Neuquen, Argentina, May 2004.
14. Gamma, Erich and Helm, Richard and Johnson, Ralph and Vlissides, John. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
15. D. Green. Trail: The Reflection API. 2004. Available at: http://java.sun.com/docs/books/tutorial/reflect/index.html.
16. G. Kaiser, P. Gross, G. Kc, J. Parekh, and G. Valetto. An approach to autonomizing legacy systems. In *SHAMAN'02*, June 2002.
17. O. Koné. An Interoperability Testing Approach to Wireless Application Protocols. *JUCS, Journal of Universal Computer Science*, 9(10):1220–1243, 2003.
18. A. Leicher and F. Bubl. External Requirements Validation for Component-Based Systems. In *14th CAiSE*, LNCS 2348, pp. 404-419, Canada, May 2002. Springer.
19. Oberndorf, P. and Brownsword, L. and Morris, E. and Sledge, C. Workshop on COTS-Based Systems. Technical Report CMU/SEI-97-SR-019, November 1997.
20. A. Ranganathan and R. Campbell. An infrastructure for context-awareness based on first order logic. *Personal and Ubiquitous Computing*, 7:353–364, 2003.
21. J. Torgersson and A. Dorling. Assessing CBD What's the Difference? In *28th IEEE EUROMICRO'02*, pages 332–341, Dortmund, Germany, September 2002.
22. S. Vinoski. Integration with Web Services. *IEEE Internet Comp.*, 7(6):75–77, 2003.
23. H. Weber. Continuous Engineering of Information and Communication Infrastructures. In *FASE'99*, LNCS 1577, pp. 22-29, Germani, March 1999. Springer.