On the Composition and Extension of Software Components

Carlos Canal Ernesto Pimentel José M. Troya
Dept. Lenguajes y Ciencias de la Computación
University of Málaga
Campus de Teatinos, 29071 Málaga, Spain
Email: {canal,ernesto,troya}@lcc.uma.es

Abstract

Architectural specifications of software systems show them as a collection of interrelated components, and constitute what has been called the Software Architecture level of software design. It is at this level where the description and verification of structural properties of the system are naturally addressed. Besides, the use of explicit descriptions of the architecture of software systems enhances system comprehension and promotes software reuse. Several notations and languages for architectural specification have been recently proposed. However some important aspects of composition, extension and reuse have not been properly addressed, and deserve further research. These include language aspects, such as derivation of components and architectures using mechanisms of inheritance, polymorphism and parameterization, and also verification aspects, like analysis of compatibility among system components. Our approach tries to address some of these open problems by combining the use of formal methods, particularly process algebras, with concepts coming from the object-oriented domain.

**Keywords: Software Architecture, formal methods, π-calculus, compatibility,
inheritance of behavior**

**Workshop Goals: sharing knowledge, discussing with other groups, obtaining
conclusions: state of the art and future trends**

1 Background

The authors of this paper are members of the Software Technology Group (GISUM) at the University of Málaga, where they have participated in a number of research projects related to Software Engineering for open and heterogeneous distributed systems [DRT97a], and to the application of formalisms to several fields, such as the integration of programming paradigms (object-orientation, concurrency, functional and logic models) [PT93, DPT94, MP97], and the specification and validation of communication protocols [MT96]. Recently, our interest shifted from object-orientation to component-orientation, and the application of the latter to different levels of the development process. Some of the currently most active research directions in the group involve Software Architecture [CPT97], Coordination Models [DRT97b], Component-Based systems [VT97] and Component-Oriented Application Frameworks [FT97].

2 Position

2.1 Software Architecture

The term Software Architecture (SA) has been recently adopted referring to the level of software design in which the system is represented as a collection of computational and data elements, or components, interconnected in certain way. From this point of view, we can consider SA as the level where the architecture and structural properties of software systems are described. SA focuses in those aspects of design and development which cannot be suitably treated inside the components that form the system [SG95]. Among them are included those which derive from the structure of the system, i.e. from the way in which its different components are combined.

The significance of explicit architectural specifications of software systems is twofold. First, they raise the level of abstraction, facilitating the description and comprehension of complex systems. Second, they increase reuse of both architectures and components. However, effective reuse of a certain architecture often requires that some of its components can be removed, replaced, and reconfigured without perturbing other parts of the application [NM95].

Although object-orientation can be applied to all levels of software design, in SA the more general term *component-oriented* is preferred, allowing to consider not only objects but architectures, interaction mechanisms and design patterns as first-class concepts of a software architecture [Nie95]. However, most concepts coming from the object-oriented paradigm can be almost directly applied to SA. In particular, we are interested the application of inheritance, parameterization and polymorphism to the specification of software architectures.

2.2 Architectural Description Languages and Tools

Despite of the importance of architectural aspects in software development, their descriptions have been traditionally limited to the use of certain idioms [SG94], such as *client/server architecture*, *layered architecture*, etc. These textual indications were generally accompanied with informal box-and-line diagrams. However, these descriptions lack of a precise meaning, which limit their utility to a great extent [AAG93].

Only in the 90's appeared the first Architectural Description Languages (ADLs). ADLs address the need for expressive notation in architectural design. They try to provide precise descriptions of the *glue* for combining components into larger systems. However, most of the work is to be done yet [Gar95]. While the proposed notations seem useful for the description of complex software systems, most of them are not formally based (see the section on Comparison), which prevents the analysis and proof of the properties of the systems and architectures described. In addition, several significant issues, such as parameterization or inheritance, are not usually addressed.

In order to avoid some of these limitations, our interests focus on the application of formal methods to SA. Formal specifications have a precise meaning derived from the semantics of the notation used and they admit several forms of reasoning, providing a formal basis for ADLs and allowing the development of verification tools.

2.3 Process Algebras

The success on the application of formal methods to SA depends on the ability in finding models and formalisms adequate to this level of the development process. To this effect, process algebras are widely accepted for the specification of software systems, in particular for communication protocols and distributed systems. The systems so specified can be checked for equivalence, deadlock freedom, and other interesting properties.

In particular, we propose the use of the π-calculus [MPW92] for the specification of software architectures. The π-calculus is a simple and powerful process algebra which can express directly *mobility* [EN86], making easier the specification of dynamic systems. Mobility is achieved in the π-calculus by passing channel names as arguments of messages. Since the names received can be used as channels for future transmissions, this allows an easy and effective reconfiguration of the system.

The calculus permits the specification of both *local* and *global choices*. Local choices indicate that a process may decide, not depending on its context, to commit to a certain transition or not, and model nondeterministic behavior. Global choices indicate that a process only commits to a transition after an agreement with another process performing the complementary action (either by local or global choice), and model deterministic behavior. Thus, using the π-calculus, we are able to express whether a process can initiate a certain communication (and then it *requires* that its environment could follow its decisions), or it just *offers* a certain behavior to its environment.

The formal basis of the π-calculus permits the analysis of the specifications for bisimilarity, deadlock and other interesting properties, and also the development of automated verification tools [Vic94]. However, the π-calculus is a low level notation, which makes difficult its direct application to the specification of large systems. Hence, a higher level notation is required. Formal specifications in π-calculus can be incorporated into the description of components by extending one of the existing ADLs.

2.4 Composability

One of the properties that we may analyze in a software architecture is *composability*, which we could define as the capability of the system of being composed by combining its components as indicated in its architecture. Composability can be checked by determining whether the components of the system are *compatible* or not. Furthermore, in order to enhance reusability, we should be able to check if a certain existing component can be used in a new system where a similar function is required. Again the intuitive notion of *compatibility* arises.

Compatibility could be determined by composing in parallel the specifications in π-calculus of system components. Then, the resulting system would be analyzed for deadlock. However, this would be impractical for complex systems, as it requires the analysis of all the interaction traces of large specifications. Instead of that, we propose the use of explicit interface specifications, or *roles*, for each connection or *attachment* between components of the system, indicating the behavior of those components as seen from outside. Then, each attachment between roles is checked locally for compatibility. This reduces the complexity of the analysis to a great extent.

Thus, a software component will be specified by set of roles, which describe its behavior in relation to the other components it is attached to. Each role describe the protocol that the component follows with respect to a certain attachment. Our notion of role derive from the *Hiding* operator ($/L$) defined in [Mil89] for CCS, and roles may be considered as partial specifications of the interface of a component. As we usually want to attach roles that match only partially, equivalence checking, using for instance the bisimilarity relations established for the π-calculus, is not well suited for our purposes. Thus, we have defined a relation of protocol compatibility in the context of π-calculus. This relation ensures that two components, represented by a pair of roles, will be able to interact without deadlock until they reach a well-defined final state. In this context, the analysis of the composability of a software architecture is reduced to local analysis of compatibility. Compatibility ensures that any software system built according to the specifications of the architecture will not produce a deadlock caused by the interaction in any attachment between its components.

A formal proof of the properties of compatibility is out of the scope of this paper, but it can be found in [CPT97]. Compatibility analysis can be easily automated in a similar way to the characterization devised by Sangiorgi for the bisimilarity relations [San93].

2.5 Extensibility

In order to promote effective reuse of both components and architectures a mechanism of redefinition and extension of roles and components is required. In the object-oriented paradigm reuse is achieved by inheritance and polymorphism. Inheritance refers to a relation among object classes by which a heir class inherits the properties (methods and attributes) of its parent classes, while it can extend them by adding its own properties. The inherited properties may be redefined, usually under certain restrictions, but roughly speaking, we may say that the interface of the heir class includes those of its parents. Inheritance is a natural condition for polymorphism, as it ensures that the derived classes will have at least the same properties than their parents. Thus, a relation of inheritance would be also of use for specifications of software components.

However, in our context the interface of a component is defined not only by the signature of its properties (i. e. the signature of its roles), but this interface also includes the behavioral patterns described in the roles. Thus, redefinition of behavior is restricted by several conditions, which define what we have called a relation of inheritance among roles. These conditions ensure that role compatibility is closed under inheritance, in the sense that if two roles are found compatible, any derived role related to one of them by inheritance will be also compatible. Role inheritance define the restrictions which allow polymorphism of behavior, and promotes both incremental specification and reusability.

Again, we refer to [CPT97] for a formal definition of role inheritance and its properties.

```
component Producer-Consumer {
  interface empty;
  composition
   p :  Producer;
   c :  Consumer;
   b :  Buffer;
  attachments
   p.Writer(in,wquit) <> b.Input(in,wquit);
   c.Reader(out,rquit) <> b.Output(out,rquit);
  }
```

```
component Producer {
  interface role Writer(w,q) = ···;
  }

component Consumer {
  interface role Reader(r,q) = ···;
  }
component Buffer {
  interface
   role Input(i,iq) = ···;
   role Output(o,oq) = ···;
  }
```

Figure 1: Architecture of a Producer-Consumer system

2.6 Extensible architectures or frameworks

Role inheritance can be easily extended to components. A child component inherits its roles from its parents, while redefinition is restricted by the conditions defined for role inheritance. Since role inheritance ensures the maintenance of compatibility, we can replace a component in an architecture with any other component which inherit from the former, maintaining deadlock freedom. Notice that if we found that a certain component `B` inherits from another component `A`, we can safely replace `A` with `B` in any system that contains it with no need of checking the compatibility of the modified attachments. This gives place to a mechanism of architecture instantiation, by which a software architecture can be considered as a generic *framework* [NM95] which can be partially instantiated and reused as many times as needed. Component frameworks derive from the idea of design patterns, and they represent the highest level of reusability in software development: not only source code and single components, but also architectural design is reused in applications built on top of the framework [Pre96]. Since our specifications can be verified for compatibility, this promotes both software reusability and quality.

2.7 Example

We will show the implications of our approach by means of a well-known example. Consider the architecture of a typical `Producer-Consumer` system. This architecture is built from three interconnected components: a `Producer`, a `Consumer`, and a temporal store that we can describe as a `Buffer`. Suppose that the `Producer` generates several items and sends them to the `Buffer`, using an operation `in`, until it decides to quit, which is notified by sending a predefined event `wquit`. On the other hand, the `Consumer` retrieves items from the `Buffer`, using and operation `out`, and performs some computations with them. The `Consumer` ends when it has got all the items generated by the `Producer`, which is notified by the `Buffer` with an event `rquit`.

From the description above, we can derive the specification of the interface of the components. Events `in` and `wquit` form the interface between the `Producer` and the `Buffer`, while `out` and `rquit` form the interface between the `Buffer` and the `Consumer`. Figure 1 (left) shows what could be the specification of our system in a typical ADL. The `Producer-Consumer` system is a composite which contains three sub-components, each one instance of a certain component class. These subcomponents (to be more exact the roles that represent their interfaces) are attached to each other, making the previously described interfaces.

The correctness of the attachments can be determined by static analysis, checking the conformance of the names of the operations and the type of the parameters, but mere type checking doesn't ensure that the system will not crash during execution. For instance, the `Consumer` can invoke an `out` operation when the `Buffer` is empty and the `Producer` has quitted the system. This behavior will lead to a deadlock or a run-time error, depending on the actual implementation of the operations. Thus, a correct performance of the system requires that the components involved follow a certain protocol.

In our example, the `Producer` may invoke `in` an undefined number of times, but it must finish sending an `wquit` event. This behavior must be specified by the role `Writer`. On the other hand, the `Buffer` is defined by two roles. Role `Input` must indicate that the `Buffer` is able to accept either `in` or `wquit` at any time, but always ending in `wquit` after a sequence of `in` operations, while role `Output` must specify that the `Buffer` will perform `out` operations only when it is not empty, and that a `rquit` event will be sent when the `Buffer` empties after the `Producer` has quitted. Since the `Consumer` doesn't know whether these conditions have occurred or not, this will be specified as an nondeterministic behavior of the `Buffer`, using local choices. Finally, the `Consumer` must perform `out` operations until it receives a `rquit` event, this being described in role `Reader` as a deterministic behavior, using global choices. Figure 1 (right) shows the specification of system sub-components and their roles.

Role protocols can be specified in the π-calculus (but they are not included here to avoid detailed explanations about the syntax and semantics of the calculus), and we could analyze the compatibility of the attachments `Input` <> `Writer` and `Output` <> `Reader`, finding out that the architecture is composable. Thus, an instance of this architecture, such as

```
prodcon : Producer-Consumer;
```

will be deadlock-free.

However, not every `Producer-Consumer` system follows exactly these protocols. Let's consider a certain `Producer'` which generates exactly four items before quitting. Let role `Writer'` describe this behavior. Using the relation of role inheritance, we can verify that `Writer'` inherits from `Writer`, and thus `Producer'` inherits from `Producer`. Hence,

```
prodcon' : Producer-Consumer(p : Producer');
```

an instance of `Producer-Consumer` in which the `Producer` has been replaced with an instance of `Producer'`, will be also deadlock-free.

Similarly, we could extend our `Buffer` with an operation `flush`, that empties the `Buffer`, and add this new behavior to the `Input` role. Once again, we can verify that the new `Buffer'` inherits from `Buffer`. Thus, `Buffer'` can be used in any system containing a `Buffer`, as in

```
prodcon'' : Producer-Consumer(p: Producer'; b : Buffer');
```

with no need of checking the compatibility of the extended role `Input'` with the roles representing the particular `Producer`s in these systems.

In conclusion, a single proof of role inheritance allow the replacement of a component in any system belonging to a family of related (but not identical) systems. Liveness properties of the architecture are ensured by static checking even in a scenario of dynamic, i.e. at run-time, binding.

3 Comparison

In the last few years several proposals related to the specification of software architectures have been presented. However, most of them are not formally based, which prevent any kind of analysis of the systems so specified. All these proposals are compositional, in the sense that they consider a software system as a composition of several more elemental units. However, they differ in which are considered the elemental units for the composition of software systems. Some of them, like Darwin [MK96], consider that systems are built from components, while others, like Wright [AG94b] or UniCon [S^+95], also includes connectors as first-order elements of the notation. In [AG94b], this distinction leads to the use of *ports* for components interfaces, and *roles* for connectors. This causes an asymmetric interpretation of compatibility which would lead to different relations of inheritance for ports and roles.

Besides, the distinction of components and connectors at language level does not scale very well. If we think of composite structures, formed by several components and connectors, the distinction vanishes, as usually these structures share characteristics of components (computation, data storage) and connectors (they serve to interconnect other components). The composition of components and connectors would lead to hybrid composites with free ports and roles which could not be classified either as components or as connectors. In order to maintain regularity and simplicity, we do not distinguish at language level between these categories, and all system components are specified the same, as a set of roles which represent their interface.

If we focus on formal methods, several papers [AG92, AG94a, IW95] have already proposed the use of CSP, Z or the Chemical Abstract Machine for architecture specification. In [MEK95] the π-calculus is used for defining the semantics of Darwin, where direct expression of mobility in the calculus is used to endow this language with lazy instantiation and direct dynamic instantiation mechanisms. However, the modeling of the interactions among components is not considered, and type checking is reduced to name equivalence. Aspects of type inheritance or component specialization are not considered either.

With regard to compatibility, our proposal follows the ideas developed in [AG94a], which uses CSP to determine compatibility of ports and roles. However, as it is stated in [MEK95], CCS or CSP do not seem appropriate for the description of evolving or dynamic structures. Our proposal tries to solve this problem by using the π-calculus. Direct expression of mobility in the calculus makes easier the architectural specifications for dynamic systems. Furthermore, [AG94a] does not address inheritance or specialization.

In [YS97], finite-state diagrams are used for the specification of what they call *protocols*, and relations of compatibility and protocol subtyping are also provided. Our approach differs from theirs in several relevant characteristics. Some of them derive from the richer expressiveness of process algebras with respect to state diagrams, as indicated in [MS95]. First of all, the use of the π-calculus allows the specification of dynamic systems simply by sending channel names as arguments of messages, while their proposal refers only to static architectures. Second, using the π-calculus, every message is send or received through a certain channel. Scope rules permit the restriction of channels to a set of processes, resulting in specifications more robust, modular, and also closer to software implementations, while in [YS97] channels are not considered, and every message is sent to a common *pool*, from which it could be retrieved by any component in the system; this being easily error-prone.

Third, the π-calculus distinguish between global and local choices, allowing us to express scenarios in which a component may decide locally to commit to a certain transition or not (asynchronously), or in which the components involved in a communication globally agree in the commitment. However, their approach only takes into account synchronous global choices.

References

[AAG93] G. Abowd, R. Allen, and D. Garlan. Using style to understand descriptions of software architecture. In *Proc. SIGSOFT'93: Foundations of Software Engineering*, December 1993.

[AG92] R. Allen and D. Garlan. A formal approach to software architectures. In *Proc. IFIP'92*, September 1992.

[AG94a] R. Allen and D. Garlan. Formal connectors. Technical Report CMU-CS-94-115, Carnegie-Mellon University, March 1994. Available at <ftp://reports.adm.cs.cmu.edu>.

[AG94b] R. Allen and D. Garlan. Formalizing architectural connection. In *Proc. ICSE'94*, pages 71–80, Sorrento (Italy), May 1994.

[CPT97] C. Canal, E. Pimentel, and J.M. Troya. A formal definition of compatibility and inheritance for software architectures. Technical report, Dept. de Lenguajes y Ciencias de la Computación, Universidad de Málaga, June 1997. A preliminary version was presented at JIS'96, Seville, November 1996. Available at <http://www.fie.us.es/pub/lsi/jids>.

[DPT94] M. Diaz, E. Pimentel, and J.M. Troya. Drol: A distributed and real-time object-oriented logic environment. *Computer Journal*, 37(5), 1994.

[DRT97a] M. Diaz, B. Rubio, and J.M. Troya. Distributed programming with a logic channel-based coordination model. *Computer Journal*, 1997. (to appear).

[DRT97b] M. Diaz, B. Rubio, and J.M. Troya. The tuple channel coordination model. In *Proc. ICSE'97 workshop on Software Engineering for Parallel and Distributed Systems*, pages 95–106, Boston, May 1997.

[EN86] U. Engberg and M. Nielsen. A calculus of communicating systems with label-passing. Technical Report DAIMI PB-208, Computer Science Dept., University of Aarhus, 1986.

[FT97] L. Fuentes and J.M. Troya. A component-oriented architecture to design multimedia services on a distributed platform. In *Proc. of WWCA'97 Conference, Tsukuba (Japan)*, number 1274 in LNCS, pages 90–105. Springer Verlag, 1997.

[Gar95] D. Garlan. Research directions in software architecture. *ACM Computing Surveys*, 27(2):257–261, June 1995.

[IW95] P. Inverardi and A. L. Wolf. Formal specification and analysis of software architectures using the chemical abstract machine model. *IEEE Transactions on Software Engineering*, 21(4):373–386, April 1995.

[MEK95] J. Magee, S. Eisenbach, and J. Kramer. Modeling darwin in the π-calculus. In K.P. Birman, F. Mattern, and A. Schiper, editors, *Theory and Practice in Distributed Systems*, number 938 in LNCS, pages 133–152. Springer Verlag, 1995.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[MK96] J. Magee and J. Kramer. Dynamic structure in software architectures. In *Proc. of ACM Foundations of Software Engineering*, pages 3–14, San Francisco, October 1996.

[MP97] J.M. Molina and E. Pimentel. Modularity in functional-logic programming. In *Proc. ICLP'97*, Leuven (Belgium), July 1997. The MIT Press.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Journal of Information and Computation*, 100:1–77, 1992.

[MS95] F. Moller and S. A. Smolka. On the computational complexity of bisimulation. *ACM Computing Surveys*, 27(2):287–289, June 1995.

[MT96] P. Merino and J.M. Troya. Evp: Integration of fdts for the analysis and verification of communication protocols. In *Computer-Aided Verification CAV'96*, number 1102 in LNCS, pages 406–410. Sringer Verlag, 1996.

[Nie95] O. Nierstrasz. Requirements for a composition language. In *Proc. of ECOOP'94 workshop on Models and Languages for Coordination of Parallelism and Distribution*, number 924 in LNCS, pages 147–161. Springer Verlag, 1995.

[NM95] O. Nierstrasz and T.D. Meijler. Research directions in software composition. *ACM Computing Surveys*, 27(2):262–262, June 1995.

[Pre96] W. Pree. *Framework Patterns.* SIGS Publications, 1996.

[PT93] E. Pimentel and J.M. Troya. Compositionality issues of concurrent object-oriented logic languages. In *Proc. PARLE'93*, number 694 in LNCS, pages 529–540. Springer Verlag, 1993.

[S$^+$95] M. Shaw et al. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4):314–335, April 1995.

[San93] D. Sangiorgi. A theory of bisimulation for the π-calculus. Technical Report ECS-LFCS-93-270, University of Edinburgh, June 1993. Available at <http://www.dcs.ed.ac.uk/publications/lfcsreps>.

[SG95] M. Shaw and D. Garlan. Formulations and formalisms in software architecture. In J. van Leeuwen, editor, *Computer Science Today*, number 1000 in LNCS, pages 307–323. Springer Verlag, 1995.

[SG94] M. Shaw and D. Garlan. Characteristics of higher-level languages for software architecture. Technical Report CMU-CS-94-210, Carnegie-Mellon University, December 94. Available at <ftp://reports.adm.cs.cmu.edu>.

[Vic94] B. Victor. A verification tool for the polyadic π-calculus. Master's thesis, Department of Computer Systems, Uppsala University (Sweden), May 1994. Available as report DoCS 94/50 at <http://www.docs.uu.se/docs/reports>.

[VT97] A. Vallecillo and J.M. Troya. On the addition of properties to components. In *Proc. of ECOOP'97 workshop on Component-Oriented Programming*, LNCS. Springer Verlag, 1997. (in press).

[YS97] D. M. Yellin and R. E. Strom. Protocol specifications and components adaptors. *ACM Trans. on Programming Languages and Systems*, 19(2):292–333, March 1997.

Biographies

Carlos Canal is Assistant Lecturer of Software Engineering at the University of Málaga (Spain). His research interests deal with Software Architecture, in particular composition and coordination issues, and formal methods. He received a M.Sc. degree in Computer Science from the University of Málaga in 1993, and he is currently working on his Ph.D.

Ernesto Pimentel is Associate Professor at the University of Málaga (Spain). He is currently working in the integration of declarative programming paradigms, and the expressiveness of models of concurrency. His research experience is focused on the application of formal methods and mechanisms for concurrency to the object-oriented paradigm. He received a Ph.D. in Computer Science from the University of Málaga in 1993.

José M. Troya received his M.Sc. and Ph.D. degrees from the University Complutense of Madrid (Spain) in 1975 and 1980 respectively. From 1980 to 1988 he was an Associate Professor in that University, and since 1988 he has been Professor and Head of the Department of Computer Science at the University of Málaga. He has worked on parallel algorithms for optimization problems using dynamic programming and branch-and-bound techniques, and on concurrent logic programming. His current research interests are in coordination languages, software architectures and programming environments for heterogeneous and distributed systems.