Heavyweight extensions to the UML metamodel to describe the C3 architectural style

Jorge Enrique Pérez-Martínez
Universidad Rey Juan Carlos, Spain
j.perez@escet.urjc.es

Abstract

UML is widely accepted as the standard for representing the various software artifacts generated by a development process. For this reason, there have been attempts to use this language to represent the software architecture of systems as well. Unfortunately, these attempts have ended in the same representations (boxes and lines) already criticized by the software architecture community.

In this work we propose an extension to the UML metamodel that is able to represent the syntactics and semantics of the C3 architectural style. This style is derived from C2. The modifications to define C3 are described in section 4. This proposal is innovative regarding UML extensions for software architectures, since previous proposals where based on light extensions to the UML metamodel, while we propose a heavyweight extension of the metamodel. On the other hand, this proposal is less ambitious than previous proposals, since we do not want to represent in UML any architectural style, but only one: C3.

Keywords: UML, metamodel, C3 style, component, connector, role, port

1. Introduction

UML [17] has become the standard for representing the software products obtained in the various activities (like requirement acquisition, requirement analysis, system design, or system deployment) of a software development process. For this reason, it is not surprising that there have been attempts to use UML to represent the software architecture of an application. However, the language is not designed to represent syntactically and semanticaly the elements of a software architecture. The attempts to instantiate the constructors defined in the UML metamodel or to extend UML by using stereotypes to represent these elements has driven to the same representations (boxes and lines) that have been widely criticized by the software architecture community. Consequently, the only solution is to extend the UML metamodel.

However, the extension of the UML metamodel implies the modification of the language, which means a deviation from the standard. This has been one of the reasons used in the literature to extend UML with stereotypes or by specifying profiles for the area of interest.

In this work, we propose the extension of the UML metamodel by providing it with elements that, once they are instanciated, can represent the C3 architectural style. A question that arises at this point is why not using Architecture Description Languages (ADLs) to describe the application software architecture, therefore avoiding the change to the UML metamodel. One possible answer to this question can be found in [19]: "The currently available architectural description languages (ADLs) have not spread in industry mainly because they are not generic enough, are not standardized and are poorly supported by tools. UML is a standard, but its current semantics fails to meet the criteria stated above: it is weak at describing interfaces, the abstractions it provides are not univocal and it provides little support for modelling architecturally significant information". Additionaly, the ADLs are not integrated in any development process (like the Unified Sofftware Development Process [8]), while UML is. Hence, representing the application architecture with UML allows the integration of this representation with the rest of software artifacts.

The rest of the paper is organized as follows. In Section 2 we describe the two possible strategies to extend UML, as specified by the Object Management Group (OMG). In Section 3 we present several attempts to extend UML for representing software architecture. In Section 4 we describe the main elements that appear in the description of the C3 architectural style (a variation of the C2 style). In Section 5 we characterize these elements as UML metaclasses. Finaly, Section 6 presents conclusions and future lines of research.

2. Strategies to extend UML

The OMG [17] specifies two strategies to extend UML. The first one uses profiles, also sometimes *called lightweight built-in extension mechanisms*. The most important profile element is the stereotype. Stereotyping is a pure extension mechanism. The model elements marked with a stereotype have the same structure (attributes, associations, operations) defined by the metamodel element that describes them, plus the constraints and tagged values added by the stereotype to that metamodel element. However, with stereotypes we can not change the semantics of the metamodel elements (at most, we can refine it), change its structure, nor create new elements of that metamodel.

The second strategy is a *heavyweight extensibility mechanism* as defined by the specification of Meta Object Facility (MOF) [15]. In this strategy the goal is to extend the UML metamodel by explicitly adding new metaclasses and other metaconstructors. The difference between lightweight and heavyweight extension comes from the existence of restrictions on the way the UML profiles can extend the UML metamodel. These restrictions impose that any extension defined for an UML profile must be purely additive, i.e., the extensions can not conflict with the standard semantics. These restrictions do not apply to the MOF context, which can define any metamodel.

3. Related work

In this section we present several works that have used UML to represent software architectures. These works follow one of the following strategies to represent architectural elements: (1) they use the UML elements as defined by the language; (2) they use "light" extensions of UML; or (3) they use "heavy" extensions of UML.

The study of Garlan and Kompanek [3] is an excelent analysis of the possibilities of UML to represent the structural aspects of a software architecture. Following ACME [4], these authors identify the following structural aspects: components, ports, connectors, roles, systems, representations, bindings, properties, types, and styles. For their analysis they used the strategies (1) and (2) from the previous paragraph. In their study, the authors conclude that UML must be extended before it can represent software architectures.

In [9, pp. 513-514], Kandé and Strohmeier state that: "However, as a general-purpose language, the UML does not directly provide constructs related to software architecture modeling, such as configurations, connectors, and styles." For this reason, they extend UML by incorporating some of the key abstractions of the ADLs, like components, connectors, and configurations. They propose a profile for software architecture using lightweight and heavyweight extensions. These authors use the heavyweight extension mechanism to incorporate their particular interpretation of the "architectural viewpoint" concept [7] to the UML metamodel, and the lightweight extension mechanism to specify connectors, components, architectural patterns, and configurations. We believe, like Garlan and Kompanek [3], that it is not appropriate neither to stereotype a subsystem to represent one component nor to stereotype the collaboration constructor to represent a connector. Furthermore, we strongly believe that these architectural elements must be represented in the UML metamodel as first-class entities, and not as stereotypes.

Selic [21] discusses several options for the modeling of the component architectural concept, as defined by ACME, considering UML subsystems, components, and classifiers. The author concludes that the three UML elements can be used complementarily. He suggests a modification to the UML metamodel that consists of removing the generalization relation between subsystem and classifier, and replacing it by an association representing the link between a subsystem and the classifiers that realize it.

Medvidovic, Rosenblum, Redmiles, and Robbins [12] evaluate UML's ability to represent software architectures following the strategies (1) and (2) stated above. Regarding the former strategy, they conclude that the modelling capabilities of UML as it is do not fully satisfy the requirements to describe the structure of software architecture. The reasons are that (a) UML does not provide special constructors to model architecture elements and (b) the rules of a given architectural style are directly reflected in its ADL, while with UML we have to apply these rules mentaly to emulate the structural constraints. Moreover, the authors doubt of UML's ability to correctly model aspects related with the components' dynamic behaviour or with their interactions. Regarding the latter strategy to extend UML (light extension with stereotypes), the authors study how UML can support the constructors present in the ADLs: C2, Wright, and Rapide. With respect to C2, the authors conclude with a set of deficiencies of UML that prevent it to explicitly represent some architectural aspects. For instance, UML allows to specify the messages received by a class, but it does not allow to specify those sent by the class.

We can not finish this section without referring to the works of OMG in this area. OMG has launched four Request For Proposal (RFP) for UML 2.0: Infrastructure, Superstructure, OCL, and Diagram Interchange. Out of these four RFP, we will discuss the first two, since they are more closely related with software architecture. From UML 2.0 Infrastructure [13] we can highlight that UML 2.0 will define a kernel of the language and new extension mechanisms. The question is whether elements to describe software architectures will be included in this kernel and , in case they are not, whether we will be able to define an architecture description language with the new extension mechanisms. Regarding UML 2.0 Superstructure [14], we are particularly interested in the following statement (Page 25): "However, the ability to model architectures is a common requirement for most software domains and, consequently, should be part of the core modeling capabilities of UML rather than being limited to a profile." Hence, we presume that UML 2.0 will define elements to describe software architectures. However, and at the moment, some of the proposals sent for revision [16][24] do not deal with this aspect significantly.

Other related works can be found in [1], [2], [5], [6], [10], [18], [19], [20], and [23].

4. The C3 architectural style

Shaw and Garlan [22] define an architectural style as a description of component types accompanied by a pattern of execution control and/or data transfer. In this context, C3 is an architectural style derived from the C2 style [11]. The modifications introduced in the C2 style to obtain C3 are the following:

- C3, unlike C2, does not predetermines the kind of inheritance of the components. It lets the components to choose it.
- In C2 the internal structure of a component is based on four elements: internal object, wrapper, domain translator, and dialog & constraints. This structure is specially designed to deal with applications with a strong component of graphical user interface. As stated by Medvidovic [11], this structure does not restrict the composition properties of the architecture. In our style C3, we do not force the structure of the components, hence allowing to work with components like those of CORBA (CCM), DCOM (COM), or JavaBeans.
- We augment the definition of the C2 component interface elements, providing means for each element to declare, on top of a direction, a name, a set of parameters, and a possible result, preconditions and postconditions.

5. A proposal of heavyweight extension to UML to describe the C3 architectural style

To extend the metamodel we have followed two rules:

- We do not remove any existing metaconstructor nor modify their syntax or semantic.
- The new metaconstructors must have as few relations as possible with the metaconstructors already defined, i.e., they must be self-contained (as much as possible).

The objective behind these rules is to simplify the implementation of this extension in tools that already support the current UML metamodel 1.4.

To describe our proposed extensions to the UML metamodel we will use the same technique used by OMG to describe the metamodel. We will describe them from three viewpoints, (1) Abstract Syntax, (2) Static Semantics, and (3) Dynamic Semantics.

We will introduce the extensions to the UML metamodel to represent the structural aspect of the C3 architectural style in a new package which we call C3Description, located in the package Foundation (see Figure 1). There is a dependency between C3Description and Core because the former uses constructors defined in the latter. There is a dependency between C3Description and Data Types because the former uses types defined in the latter.

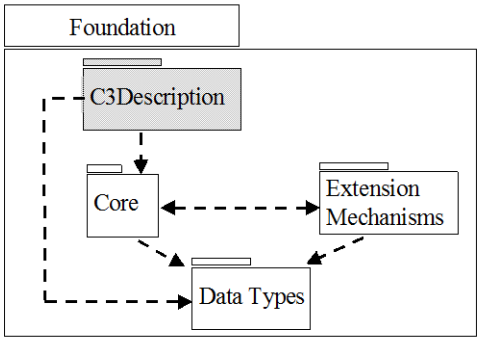

Figure 1. Relationships between the C3Description package and the rest of packages in Foundation.

5.1 Abstract Syntax

The abstract syntax for the package C3Description is shown in Figure 2. Although not shown in the figure, the new constructors are added to the metamodel as subclasses of ModelElement (which defines the name metaattribute) which is itself a subclass of Element (the root metaclass). As can be seen in Figure 2, we use the constructors Constraint, Attribute, and Parameter, defined in the package Core, and the types Boolean and ProcedureExpression, defined in the package Data Types.

5.1.1 Architecture

This element represents a container of the construction blocks that can appear in a C3 architecture. In the metamodel it is stated that an architecture is formed by two or more components and one or more connectors. As attribute an architecture has name (inherited from ModelElement).

5.1.2 Component

In a component, the top and bottom domains are represented with two ports: one with value top in the attribute domain and one with value bottom in that attribute. A component is an active element in the sense that it has its own control flow(s). A component has a state, shown in the state variables described by the constructor Attribute. A component can declare an invariant, which is supported by the metaclass Constraint, defined in the package Core. In Figure 2 the relation between Component and Constraint is made explicit for the sake of clarity, since Component inherits this relation from ModelElement. A component can be composed, in the sense that it can contain an architecture. As attributes it has name (inherited from ModelElement) and isActive, which indicates that the component has its own control flow(s).

5.1.3 Connector

A connector has one or more interaction points, characterized with the constructor Role. A connector is an active element, in the sense that it has its own control flow(s). A connector can be composed, in the sense that it can contain an architecture. A connector supports a filtering policy. As attributes it has name (inherited from ModelElement) and isActive, which indicates that the connector has its own control flow(s).

5.1.4 InterfaceElement

This constructor represents an operation involved in the interaction of a component with its environment. As attributes it has name (inherited from ModelElement) and direction, which is an enumerated type with values {prov, req} indicating whether the component supports the operation or requires it to be provided by the environment, respectively. An interface element can have a precondition and/or a postcondition associated with it. These are established over the state variables of the component and/or the parameters. Moreover, an interface element declaration can have parameters in the role of arguments as well as in the role of results, as defined in the package Core of UML.

5.1.5 Filter

This constructor represents the filtering policy of the connector to which it is associated. As attributes it has name (inherited from ModelElement) and f, which is an expression that represents the filtering policy.

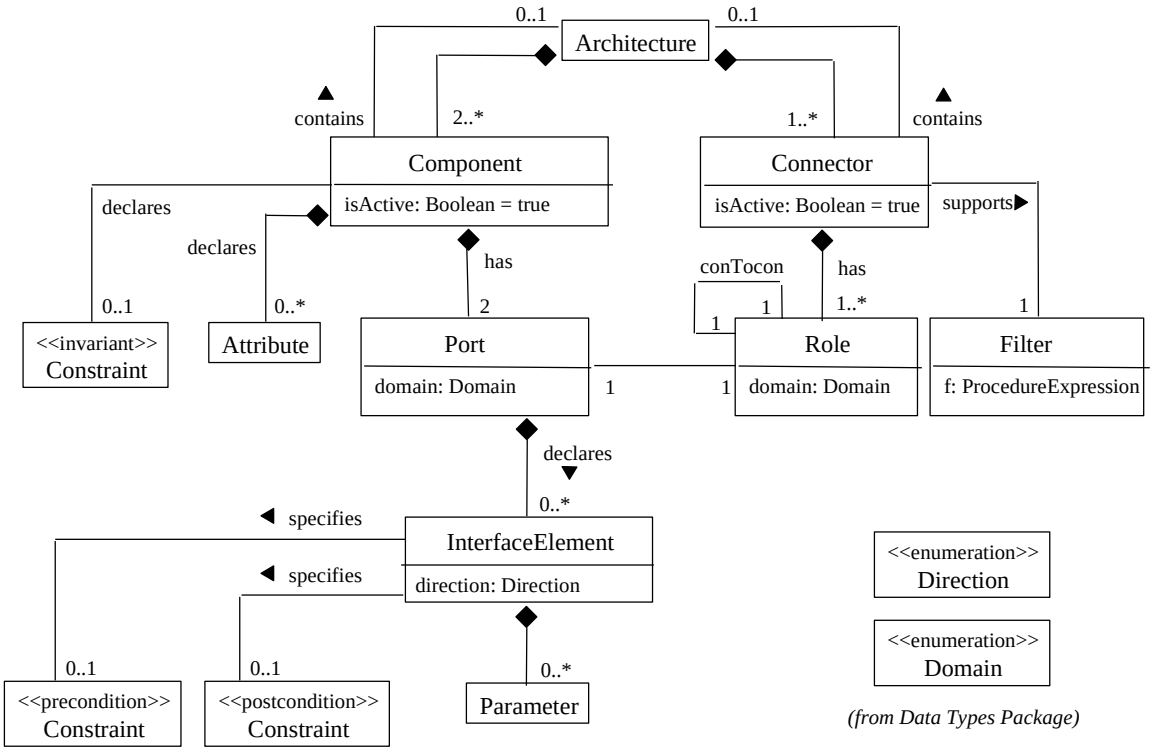

Figure 2. Abstract syntax for the C3Description package.

5.1.6 Port

A port represents a point of interaction of a component with its environment. It has a domain that specifies whether the port represents the top or the bottom domain of its component. A port can be connected to a role of a connector. It declares a set of elements that forms the component interface in its top and bottom domains. As attributes it has name (inherited from ModelElement) and domain, which is an enumerated type with values {top, bottom}.

5.1.7 Role

A role represents a point of interaction of a connector with its environment. It has a domain that specifies whether the role belongs to the top or the bottom domain of the connector in which it is defined. A role can be connected to a port of a component or to another role in some other connector. As attributes it has name (inherited from ModelElement) and domain.

5.2 Well-formedness rules

Due to space restrictions, we only present the well-formedness rules for connectors. They are shown in Figure 3.

5.3 Dynamic Semantics

In this section we detail the dynamic semantics of the elements component and connector of the package C3Description.

5.3.1 Component

As we said above, a component has a top domain and a bottom domain. The top domain specifies the set of notifications to which the component responds and the set of requests issued by the component. The bottom domain specifies the set of notifications issued by the component and the set of requests to which the component responds. If we assume that between a component and a connector there is a total communication in the sense of [11], then the following properties can be established:

- The set of requests issued by a component must be a subset of the operations offered by the components placed above it.
- The set of notifications to which a component can respond must be a subset of the notifications that the components above it can issue.
- The set of notifications issued by a component must be a subset of the notifications that the components placed below it can respond.

```
[1] A connector has one or more roles.
context Connector inv oneOrMoreRoles: self.role -> size() ≥ 1

[2] A connector supports only one filtering policy.
context Connector inv aFilter: self.filter -> size() = 1

[3] The roles of a connector can not be connected among themselves.
context Connector inv noAutoconexión:
     self.role -> forAll (r1, r2 | r1 <> r2 implies r1.role <> r2)

[4] A connector can be a simple or a composed element.
context Connector inv composition:
     self.architecture -> size () = 0 xor self.architecture -> size () = 1

[5] The top interface of a connector is the union of the top interfaces of the components and
connectors connected to its bottom roles.
context Connector inv topInterface: topInterface (self)

being:

topInterface (c: Connector): Set;
topInterface (c) =
     let bottomRoles : Set = self.role -> select (r | r.domain = Domain::bottom) in
     bottomRoles -> iterate (r:Role ; acc:Set = Set{} |
          if r.port-> size() = 1 then acc -> union (r.port.interfaceElement)
          else
               if r.role -> size() = 1 then acc -> union (topInterface (r.role.connector))
               else
               endif
          endif)

[6] The bottom interface of a connector is the union of the bottom interfaces of the components
and connectors connected to its top roles.
context Connector inv bottomInterface: bottomInterface (self)

being:

bottomInterface (c: Connector): Set;
bottomInterface (c) =
     let topRoles : Set = self.role -> select (r | r.domain = Domain::top) in
     topRoles -> iterate (r:Role ; acc:Set = Set{} |
          if r.port-> size() = 1 then acc -> union (r.port.interfaceElement)
          else
               if r.role -> size() = 1 then acc -> union (bottomInterface (r.role.connector))
               else
               endif
          endif)

 [7] A connector can have zero or more connectors connected in its top domain.
context Connector inv conTopCon:
let topRoles: Set = self.role -> select (r| r.domain = Domain::top) in
(topRoles -> select (r| r.role -> size() = 1))-> size() ≥ 0

[8] A connector can have zero or more connectors connected in its bottom domain.
context Connector inv conBottomCon:
let bottomRoles: Set = self.role -> select (r| r.domain = Domain::bottom) in
(bottomRoles -> select (r| r.role -> size() = 1))-> size() ≥ 0
```

Figure 3. Well-Formedness Rules (connectors).

- The set of requests to which a component responds must be a subset of the requests issued by the components below it.

5.3.2 Connector

The primary function of a connector is to conduct the traffic of messages between components. As a secondary function, a connector supports a filtering policy. In C3, two policies are defined (inherited from C2):

- Message filtering, each message is sent only to those components that can understand it and respond to it.
- Message sink, the connector ignores each message sent to it.

6. Conclusions and Future Work

In this work we have described extensions to the UML metamodel to represent software architectures. We have underlined the inability of the language to represent all the aspects of a software architecture and hence concluded the need to extend the metamodel. Regarding this point, this work proposes an extension to the UML metamodel to describe the C3 architectural style, centered in its main elements: components and connectors. The description of this extension has followed the same structure used by OMG to describe the metamodel: abstract syntax, well-formedness rules, and dynamic semantics.

In this work we have described structural aspects of the C3 architectural style. We are currently studying the introduction of dynamic and configuration aspects of C3 to the proposed extension. The integration of these aspects requires the representation in the metamodel of the concept of "viewpoint" as defined in [7]. Once the complete description of C3 is in the metamodel, we plan to integrate the description in a tool like Rational Rose™. We believe this integration provides several benefits. On one hand, we would be putting together academia, that uses ADLs, and industry, that uses UML. On the other hand, this integration allows us to study the potential transitions from the products generated during the requirement analysis to the software architecture of the system, and from the software architecture to the low-level design. In the midterm, we plan to use the experience obtained in the creation of this prototype for C3 to generalize this work for other architectural styles.

References

[1] Abi-Antoun, M. and Medvidovic, N. (1999). Enabling the refinement of a software architecture into a design. *In Proceeding of The Second International Conference on The Unified Modeling Language (UML'99)*. CO, USA: Springer-Verlag.

[2] Egyed, A. and Medvidovic, N. (2001). Consistent architectural refinement and evolution using the Unified Modeling Language. *In Proc. of the 1st Workshop on Describing Software Architecture with UML, co-located with ICSE 2001*. Toronto, Canada, pp. 83-87.

[3] Garlan, D. and Kompanek, A.J. (2000). Reconciling the needs of architectural description with object-modeling notation. *UML 2000 – The Unified Modeling Language: Advancing the Standard. Third International Conference*. York, UK: Springer-Verlag.

[4] Garlan, D., Monroe, R. and Wile, D. (2000). Acme: Architectural description of component-based systems. *Foundations of Component-Based Systems*, Cambridge University Press.

[5] Gomaa, H. and Wijesekera (2001). The role of UML, OCL and ADLs in software architecture. *In Proceedings of the Workshop on Describing Software Architecture with UML, 23rd International Conference on Software Engineering*, Toronto, Canada.

[6] Hofmeister, C., Nord, R.L. and Soni, D. (1999). Describing software architecture with UML. *In Proc. of the First Working IFIP Conf. on Software Architecture*. San Antonio, TX: IEEE.

[7] IEEE (2000). IEEE Recommended practice for architectural description of software-intensive systems.

[8] Jacobson, I., Booch, G. and Rumbaugh, J. (1999). *The unified software development process*. Massachusetts: Addison-Wesley.

[9] Kandé, M. M. and Strohmeier, A. (2000). Towards a UML profile for software architecture descriptions. *UML 2000 – The Unified Modeling Language: Advancing the Standard. Third International Conference*. York, UK: Springer-Verlag.

[10] Lüer, C. and Rosenblum, D.S. (2001). UML component diagrams and software architecture- experiences from the WREN project. *In Proceedings of the Workshop on Describing Software Architecture with UML, 23rd International Conference on Software Engineering*, Toronto, Canada.

[11] Medvidovic, N. (1999). Architecture-based specification-time software evolution. (Doctoral Dissertation, University of California, Irvine, 1999).

[12] Medvidovic, N., Rosenblum, D.S., Redmiles, D.F. and Robbins, J.E. (2002). Modeling software architectures in the unified modeling language. *ACM Transactions on Software Engineering and Methodology*, 11 (1), 2-57.

[13] OMG (2000). Request for proposal: UML 2.0 infrastructure RFP.

[14] OMG (2000). Request for proposal: UML 2.0 superstructure RFP.

[15] OMG (2001). Meta Object Facility (MOF) specification (version 1.3.1).

[16] OMG (2001). OMG Unified Modeling Language specification (initial submission). Version 2.03, interim superstructure submission. Financial Systems Architects, MEGA International, Mercury Computer Systems, TogetherSoft, Hitachi.

[17] OMG (2001). Unified Modeling Language specification (version 1.4).

[18] Rausch, A. (2001). Towards a software architecture specification language based on UML and OCL. *In Proceedings of the Workshop on Describing Software Architecture with UML, 23rd International Conference on Software Engineering*, Toronto, Canada.

[19] Riva, C., Xu, J. and Maccari, A. (2001). Architecting and reverse architecting in UML. *In Proceedings of the Workshop on Describing Software Architecture with UML, 23rd International Conference on Software Engineering*, Toronto, Canada.

[20] Robbins, J.E., Medvidovic, N., Redmiles, D.F. and Rosenblum, D. (1998). Integrating architecture description languages with a standard design method. *In Proceedings of the International Conference on Software Engineering* (pp. 209-218). Kyoto, Japan: IEEE.

[21] Selic, B. (2001). On modeling architectural structures with UML. *In Proceedings of the Workshop on Describing Software Architecture with UML, 23rd International Conference on Software Engineering*, Toronto, Canada.

[22] Shaw, M. and Garlan, D. (1996). *Software architecture. Perspectives on an emerging discipline*. N.J., USA: Prentice-Hall.

[23] Störrle, H. (2001). Turning UML-subsystems into architectural units. *In Proceedings of the Workshop on Describing Software Architecture with UML, 23rd International Conference on Software Engineering*, Toronto, Canada.

[24] U2 (2001). 2U submission to UML 2 RFP, initial submission to superstructure. Adaptative, Data Access, Kinetium, Softlab, Siemens.