SAFARCHIE STUDIO: ArgoUML EXTENSIONS TO BUILD SAFE ARCHITECTURES

Olivier Barais - Laurence Duchien
Université des Sciences et Technologies de Lille
Laboratoire LIFL (UMR 8022), Projet INRIA-FUTURS JACQUARD
Fr 59655 Villeneuve d'Ascq Cedex
{barais,duchien}@lifl.fr

Abstract Nowadays, no standard and universal definition of software architecture was accepted by all the community. Various points of view on different studies bring to several approaches. These approaches focus on only one or two concerns such as component interfaces specification, behavioral analysis or software reconfiguration. This paper argues that, in order to accrue the true benefits of software architecture approaches, one may need to use an architecture centric approach with a global reasoning: From software architecture design to software architecture management to software architecture building, deployment and refinement. However, these different concerns of a software architecture definition must be in consistency. For this reason, we based our approach on architecture types that are points of reference at each step of our reasoning. We offer with SafArchie Studio, a first architecture centric approach based on three-view perspective and driven by the component life cycle.

Keywords: Software Architecture, Tools

1. Introduction

Nowadays, industrial component platforms such as CCM[22], EJB[11] give first means for assembling and deploying components on distributed environments. Academic approaches with ArchJava[2] or Fractal[7] provide more structural elements as composite or connector for precisely defining an abstract architecture and transforming it on deployed and running code. However, facing the difficulties to define correct and safety software architectures, more abstract software architecture models were proposed. They are commonly gathered under the label of ADL for Architecture Description Languages. They come with powerful methods and tools of specification and analysis for high-level designs.

In other hand, since several years, software engineering community promotes model based engineering. New industrial components platforms should allow programmers to easily create distributed software by a composition of components[13] [26]. Nevertheless the re-usability implied by this view seems to be skewed. Platforms are in constant improvement and the re-usability is actually impossible. Therefore, the software engineering community should ab-

stract essential concepts from the platforms. The OMG (Object Management Group) tries to resolve this new challenge by MDA (Model Driven Architecture) [25] approach. In this context, architecture languages become one of building steps mainstays. They define models and tools to build platforms independent, reusable, safe software architecture.

We propose SafArchie, an architecture centric approach for helping architects, designer and developers in the composition of components at each step of the life cycle of applications. Built from a set of models for putting in evidence some properties according to the component life cycle, SafArchie Studio assists the software architecture specification. We based our tool tool on architecture types that we instantiate and deploy on abstract platform. SafArchie integrates results from component frameworks and models based engineering in a powerful ergonomic tool. This tool aims to increase software architecture reasoning in front of component based design and programming on platforms such as CCM, ArchJava, or Fractal.

This paper follows the reasoning about building new software architecture. The first section is an overview of SafArchie approach. It defines the three-view perspective approach leaded by component life cycle. In the second section, we present the design perspective based on architecture types. We propose a reasoning on this design and a new ArgoUML[27] diagram to create this specification. Section 4 defines the deployment perspective, the first physical abstraction model and the mapping to it. The end of section 4 specifies the execution perspective. Some next steps of this work and next features of the SafArchie studio are defined. The section 5 presents related works on software architecture from which we build SafArchie Studio. Finally, the last section gives a conclusion of this work.

2. Overview of SafArchie approach

2.1 A three-view perspective approach

Software architecture defines component instances, their interfaces, their structures, their interactions and the mappings to hardware systems. It represents a static view at a given moment. However, at the run-time, this architecture will be dynamic[4]. For example, component instances and bindings can appear and disappear. In ubiquitous computing domain, run-time environment can also change in accordance with the context.

Defining architecture should consider all facets of the application life. Component approaches provide lots of information in relation with the component life cycle and this information should clarify the architecture. SafArchie works on three-view perspectives depending on the component life cycle: Design perspective, deployment perspective, and run-time perspective. For each perspective, we define a meta-model for clarifying its concepts.

The *design perspective* defines *architecture type* by a set of constraints respected by an architecture. These constraints represent invariants for software architecture such as component interfaces, relations and interactions between components, but also component structure and behavior specifications. From

this architecture type, we check the structural and behavioral conformities between components constraints. Therefore *Design perspective* defines consistent architecture types. By this way, architect guarantees architecture consistency.

The second perspective, called *deployment perspective*, ensures an initial position for a logical architecture on top of a physical architecture. For this perspective we need three models. The first one, the instance model (called *logical architecture*) is a graph of connected component instances. The second one, the *physical architecture*, consists of an abstract representation of physical platforms. Finally the last one specifies the mapping of a logical architecture to an abstract physical architecture.

Finally the last perspective, *run-time perspective*, consists of a supervision of software architecture. This work still in progress builds framework to follow the evolution of a software architecture at run-time.

SafArchie Studio is an extension of ArgoUML tool. We maintain the consistency among the models thanks to a common abstract component model that unifies the common definition and an architecture type model that helps in conformity check at each step of life cycle. We present this component model in the next sub-section.

2.2 SafArchie common abstract component model

Our representation of a component and of relations between components is closely related to ArchJava component model [1]. A component is a grey box defined by its structure and its interfaces. Components can be composed and must be as generic as possible for easily reusing in different contexts. They communicate through ports by provided or required operations. A component is defined by two parts. Firstly, the communication part represents the component interfaces defined by a set of ports. Secondly, the internal part corresponds to the component implementation. In the internal part, only public *attributes* are visible, the component implementation component is hidden. A component can only communicate with its environment through explicitly declared ports.

A *port* is an access point on the component. It represents a logical communication way. Each port defines two sets of operations, *provided* or *required* operations.

An *operation* represents an action of a component. A provided operation can be invoked by a client component. Conversely, the required operations can be processed by a server component. They make explicit dependencies. These dependencies are resolved at the composition time. Operations are defined like methods in Object Oriented Programming (OOP) with a signature with name, parameters, result, and exceptions. An operation can be released by a remote procedure call or a sent message.

Finally, as in ArchJava or Fractal component models, our component model is hierarchical. A component can be a *composite* or a primitive. A composite contains delegation ports, component children and *bindings* between these component children. The ports of a composite are only delegated, i.e. they reference a component child port. The composite reifies the software architecture configuration.

3. Modelling software architecture types

In architecture definition, first steps of architect abstract global structures of the future application by the definition of component assemblies. First, we define concept of architecture type for helping architect in its first design steps. Second, we clarify this architecture by structural and behavioral contracts. Therefore, checking contracts makes architecture safe. Finally, we integrate our approach in ArgoUML.

3.1 Type of architecture definition

As in type definition in programming language, an *architecture type* defines a set of possible values that must be respected by software architectures. Defined constraints deal with component interfaces and identify relations and interactions between these component interfaces. Therefore, an architecture type represents a static view of a software architecture. It also clarifies component structure or behavior specification. From these architecture types, we will check the structural and behavioral compatibilities between components by set of identified invariants.

An architecture type is composed of six main elements: Component type, composite type, bindings, port type, operation, and attribute (see Figure 1).

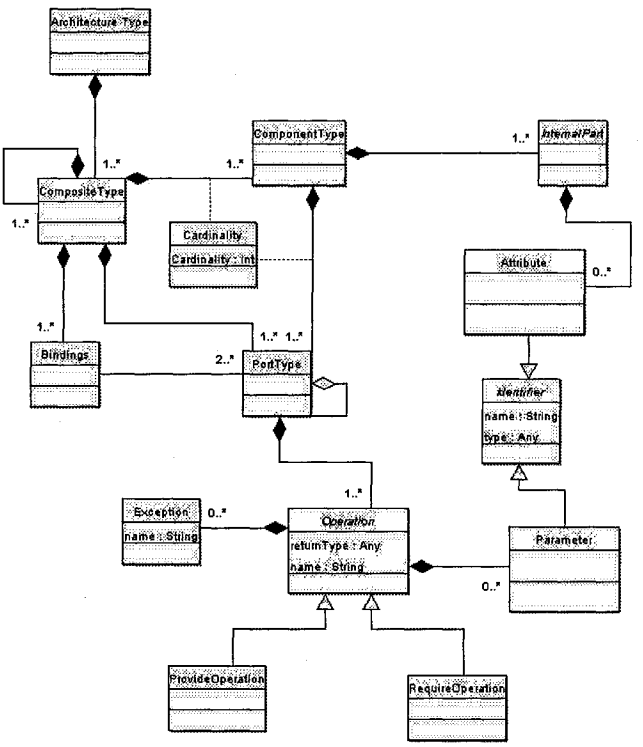

Figure 1. Architecture type meta-model

Designing a *port type* consists of identification of a set of operations that the port should provide or require (see Figure 1). A port type corresponds to a set of operation signatures and their gathering together is guided by the application design. *Component type* defines all port types of the component and the minimum and maximum cardinalities for each one (see Figure 1).

Composite type also identifies all the component types that it should contain and the minimum and maximum cardinalities for each one (see Figure 1). It defines the interactions between these component types through the *binding* concept. Then, a binding identifies a possible interaction between two port types belonging to component types.

By this way, software architecture type is a set of structured constraints in terms of composite type, component type, and port type. Each typed software architecture should respect these constraints.

Example We illustrate our type model with an example describing the architecture type of a replicated calculator. This architecture type has two primitive component types bound into one composite type (see Figure 2). The component type **A** provides one service of calculator through a port type **P**. Each instance of **A** could have only one port **P**. The specification defines the cardinality between the component type **A** and the Port **P**.

The component type **B** is a proxy. It receives some requests and chooses the provider for the service. It has two port types: (**P**,**Q**): the first to communicate with the client, the second to interact with the service provider.

Finally, the composite type **C** has one binding and one delegated port type . The composite environment communicates with component type **B** through this latter. The binding models the communication capabilities between **A** and **B**. The composite could have several instances of **A** but only one instance of **B**.

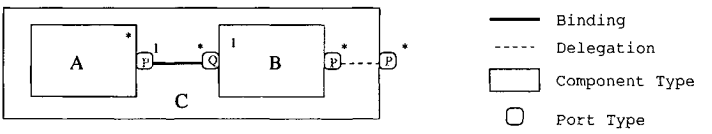

Figure 2. Architecture type example

3.2 Design software architecture by contracts

All the constraints defined by the composite type, the component type and the port type are structural and architectural. With this framework we will assist the components composition. For example, from an architecture type, we could decide the binding of two port types if all the operations provided by the first port type are required by the other and vice-versa. However this type checking often used in Object Oriented Programming has already shown its limits [15]. The compatibility between two operation prototypes could not guarantee their correct use. That is why, in SafArchie, we add a visible part of a black box component in order to simplify the components integration.

```
            provide Boolean withdraw(String clientID, Integer price)
            Context Bank::Auth::withdraw(clientID:String,
price:Integer):Boolean
            pre:  price > 5
            post:  self.accountValue > 0
```

Figure 3. Pre and post condition example

These specification enrichments are commonly used in *Design by Contract* software development techniques. They ensure high-quality software architecture. They guarantee every component of a system lives up to its expectations [20]. In SafArchie architecture type meta-model, we identify two contracts. The first contract is a set of constraints on the operation, called *assertion contracts*. It adds *pre* and *post* conditions in an operation to guarantee its correct using context. The second contract, called *behavioral contracts*, specifies the external behavior of a component type.

3.2.1 Assertion contracts.

In some case, an operation could be invoked and processed only if its context is valid. The context of a component is described from attribute values, parameter values but also architectural constraints like the presence or the lack of a binding. An assertion is a boolean expression about the state of a software system. In a valid software system, all assertions evaluate to true.

Architect can define two classes of assertion. First, *preconditions* define the context that must hold before an operation can be invoked. They are evaluated just before an operation execution. Precondition involves component state and operation parameters. They specify obligations that a software client component must meet before invoking a particular operation of the component. Second, *postconditions* specify conditions that must hold after an operation completes. Consequently, postconditions must be true after the operation execution. Postconditions on a provided operation involve the old system state, the new system state, the operation parameters, and the operation's return value. Postconditions on a required operation involve only the operation's return value. They define the guarantees that a software component makes to its clients. If a postcondition is violated, the supplier component does not work correctly.

For example, in a provided operation *withdraw* of component type *Bank*, we can specify that the first parameter should be upper than 5. Therefore, if a client asks a debit lower than five, an exception will be risen. Architects can also define a postcondition to ensure that the *Bank* attribute, called *accountValue*, will be always positive (see Figure 3).

3.2.2 Behavioral contracts.

The structural description of an architecture type does not allow a complete efficient re-use. Indeed, the component integration requires a well understand-

```
    A = (p.?request → computation → p.!reply → A) + {p.?request,
p.!reply}.
    B = (p.?request → q.!request → q.?reply → p.!reply → B) +
{p.?request, p.!reply, q.!request, q.?reply}.
    ||C = (A || B) /{p/q}@{p}.
```

Figure 4. Behavioral specification example

ing of each component's role. The structural information are necessary but insufficient to improve software architecture understanding tasks. In this section, we define our architecture type extension that adds behavioral information on component types. These additional details are gathered in *behavioral contract.* It describes the messages scheduling of a component type with its environment.

The building of new formal languages to describe component external behavior is a specific part of software architecture research area. In SafArchie Studio, we integrate existing works. External component types behavior are specified with FSP language [18]. This language is already used in Darwin ADL[17]. It is well-adapted with hierarchical architecture model.

The external behavior of a component type is described in term of sequences of exchanged messages with its environment. The two kinds of interaction (Remote Procedure Call (RPC) and messages) are described by asynchronous messages. These messages can be sent or received by the component. Then, in a behavioral contract, one or two messages represents each operation.

A component type behavior is defined as finite state processes in FSP using action prefix "→", choice "|", and recursion. Let $?x$ a received message and $!p$ a sent message, $?x \rightarrow !p$ defines a process that initially waits the message x and then sends message p.

A composite type process is the parallel composition of their contained component behaviors. Communication is achieved through synchronization of shared actions. The FSP hiding operator @ captures the concept of delegated port in SafArchie. The behavior of every composite is computed from that of its subcomponents with reachability analysis. Messages that are not in one composite interface are hidden and the behavior is minimized with respect to observational equivalence[21].(see [19] and section 3.3 for more details).

Take the example of a replicated calculator described in section 1. The behavior of a component typed A is simple. When it receives a request, it computes the calculus and replies a result.

The behavior of a component typed B is a proxy behavior. When it receives a request, it forwards it to a component typed A and waits the reply to send it. The composition identifies a synchronization between the messages belonging to *accept* and the messages belonging to *forward* (see behavioral specification in Figure 4). The behavioral specification identifies the message with the name of its port (*P*, or *Q*) and the name of its operation (*accept*, *reply*). The composition result gives a simple external behavior of a composite typed C. It can receive a request and it can reply a result.

3.3 Towards a safe software architecture type

Each architecture type should be consistent in order to build safe software architecture. From three sources of information, which are architecture types, assertion contracts and behavioral contracts, we check the architecture type consistency.

Structural reasoning about an architecture type

From the architecture type, we link several checking mechanisms. The first step studies the architecture type without the contracts.

First, we check that software architecture type is being carried out in accordance with the software architecture type meta-model presented in Figure 1.

Second, we verify the assembly by checking the compatibility of two bound port types. Two port types are compatible if all the provided operations of one port type have a compatible required operation in the other port type. Moreover, the compatibility between operations consists in checking their signatures compatibility.

Finally we check the possible cardinality violation. Indeed each binding declared in a composite type specifies a possible link between two ports. The cardinality analysis compares the two couples of cardinality: *(component type child cardinality, port type cardinality)*. A couple (1,1) could be safely bound only with a couple (1,1). All other bindings could create cardinality violation.

Reasoning about assertion contracts

The second step of analysis checks the assertion contracts defined in OCL (Object Constraint Language) [23] in an operation context (see example in section 3.2.1). First, we consider well-formed OCL expressions by verifying the syntax and the navigation expressions in accordance with the architecture type meta-model. For example, the post condition in Figure 3 will be valid if the component has an attribute named *accountValue* that can be compared to an integer value.

Second, OCL expression analyzer verifies compatibility between two operation assertion contracts belonging to two bound port types. Indeed assertion contracts can be viewed as subtype definition for parameters. OCL analyzer checks the covariance and contravariance respect for these sub-types. For example, in Figure 3, if the required operation defines a precondition with a withdraw lower than 4, the OCL expression is true on the first check step, but the second pass detects an incompatibility between these two assertion contracts.

Reasoning about behavioral contracts

Finally, we work on behavioral contracts. From an architecture type, there exists an infinity of valid software architectures instances. Therefore, we can not analyze all the possible behaviors. We only check the behavior consistency from only one software architecture behavior specification. This specification is

defined from the architecture type with only one instantiation of each composite type, component type, and port type with its minimum cardinality. From each binding between two port types, we create a connection between two ports. We generate FSP specification relating to this generated software architecture behavior.

LTSA tool checks this specification [9] with Compositional Reachability Analysis (CRA). It performs an exhaustive search of the state space of the Label Transition System model. This LTS model corresponds to the behavior specification. More specifically, given the structural architecture of a system, and the behavioral contracts of the component types, the behavior of the system is computed for analysis in steps. The behavior of every composite type is computed from its sub-components FSP specification with reachability analysis. Actions without communication interface are hidden. The behavior is minimized with respect to *observational equivalence* [21].

Due to the automatic generation of only one behavioral specification, the behavioral analysis is incomplete. We could only conclude when the analysis detects a problem that the architecture type is invalid. However a non failure detection does not mean the behavioral safety of architecture type. New verifications should be done at the deployment perspective (see section 4). If a consistency mistake is detected during a checking step, architect can not define a logical model (see section 4.2) instance of this architecture type.

3.4 A new diagram in ArgoUML for building safe software architecture type

Building new plugin in ArgoUML [27] is easy and well-documented. Moreover ArgoUML environment is a free, well-known, multi-platforms software engineering tool. Finally ArgoUML add the concept of *design critics*[24] to assist the software designer. Therefore, we define our design perspective prototype of SafArchie Studio as a *plugin* in ArgoUML in accordance with our meta-model. This plugin provides graphical design of architecture types. By this way, architects have a global tree view of their architecture type. They can edit all their architecture type properties inside simple panels. They can load or save their architecture types through XML files defined with a public XML schema. SafArchie Studio type perspective checks architecture type consistency and creates a global report for an architecture type.

On the left side of the figure 5, a graphical tree represents the architecture type hierarchy. Architect uses it for browsing or editing an architecture type. For each element, contextual menu and drag and drop actions improve the tool ergonomy.

On the right side, the second frame provides a graphical tool to manipulate an architecture type. An architecture type is built from component types, composite types, port types and bindings between port types. We manage one graphical layer for each composite content. Only one layer can be edited at one moment. However we could have a reduced graphical view of the content of a composite type. The navigation between the graphical layers is automatic thanks to the

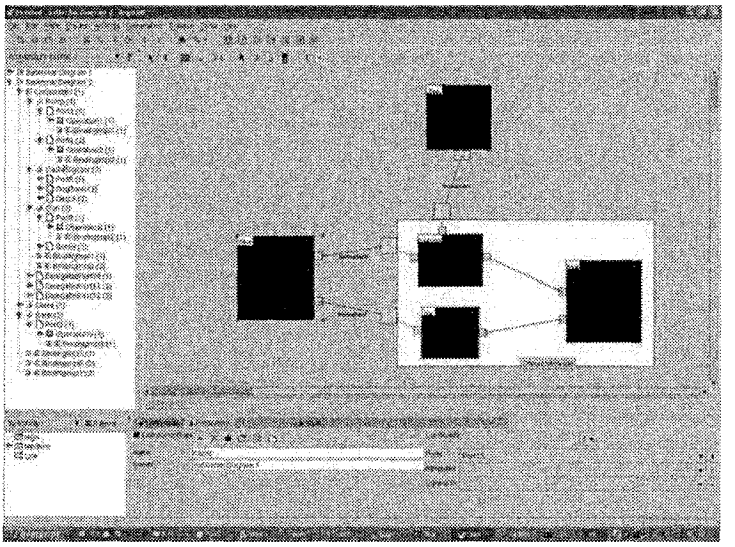

Figure 5. SafArchie Studio snapshot

graphical tree. When choosing an element in the tree, you immediately see its graphical layer.

The selection of an object in the tree panel or in the graphical panel activates the property panel (bottom right panel in Figure 5) of the target element. From it, architect can edit internal properties of each architecture type element.

The architecture type meta-model is described in UML (in Figure 1) but also with a XML schema. Architect can save an architecture type in a XML representation. XML schema corresponds to the grammar of the SafArchie Architecture Type Description Language.

An architect can automatically check an architecture type definition. SafArchie Studio checks the structure (described in section 3.3) but also assertion contracts validity and conformity (described in section 3.3) with two different tools. The first one, already used in ArgoUML, is an OCL library[14] for the checking of well-formed OCL navigation expression. We develop a facade pattern for our architecture type meta-model in order to use it. The second is an internal tool based on CIAO prolog[8]. It checks the covariance and contravariance principles between bound operations that are enriched with assertion contracts.

Finally, to verify the compatibility between behavioral contracts, SafArchie Studio generates global system behavior specification (as described in section 3.3), and starts the LTSA external model checker to verify the consistency of this specification.

4. Deployment of a typed software architecture on a distributed environment

In SafArchie, we decompose our approach in several points of view. In section 3, we presented the design perspective of our approach for building *architecture types*. They define a set of respected invariants in a design stage. In this section, a second perspective instantiates a typed software architecture and deploys it on a physical environment.

4.1 A three-step deployment step

An architecture type defines a set of possible values (i.e. infinity of possible architecture instances). An architecture instance is a connected graph of component instances through their ports. Each component instance and its relations should be conformed to its referenced architecture type. We call this instance model: *logical software architecture*.

Software architecture scope is larger than this model of interconnected component instances. This component based software architecture represents a static view of component instances deployed on a physical architecture at one moment. In SafArchie, the physical model is an abstract view of hosts configuration, hosts capacities and network topology. This abstract view of physical platforms called *Physical Architecture* is light. We only model distributed nodes, their interconnection and some information about their based software as middleware and network protocols. The last step of deployment perspective consists in mapping the logical software architecture on an abstract physical architecture.

4.2 Logical software architecture

Logical software architectures are represented by a graph of component instances connected through their ports. Each logical architecture instance must conform to its referenced architecture type. Therefore, ports, components, or composites are typed by an existing port type, component type, or composite type of architecture type. Besides, any relation between these elements should be declared in the architecture type. Finally all port connection must be defined in the binding description between these two port types.

Example Let's take the example described in section 1. A valid software architecture could be the following (see Figure 6). The replicated calculator has three replicas (a1, a2, a3). Each one is connected to a component Proxy (b1) through ports. Proxy can receive requests from the environment. It delegates the computation to a replica.

In SafArchie Studio, we add a second diagram for defining logical software architecture. As the first diagram, it is composed of four panels: A panel for global tree-view of software architecture, a panel for graphical software architecture building, a panel for adding critics on this specification and a set of property panels for each logical architecture element.

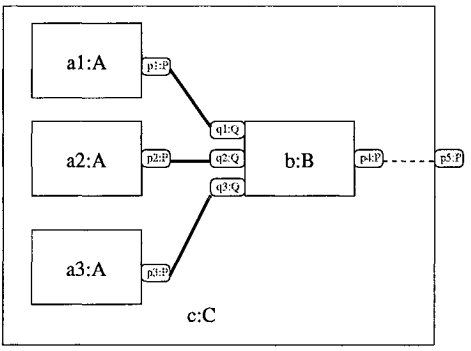

Figure 6. Replicated calculator architecture

From a logical software architecture definition, we could directly map its model to non distributed platform as ArchJava [2] or Julia, the implementation of Fractal Model [7]. SafArchie Studio transforms automatically its logical architecture XML representation and architecture type XML representation towards XML Fractal ADL. On the other hand, ArchJava projection is manual. SafArchie Studio creates an ArchJava file specification for each component specification.

4.3 Abstract physical architecture

For our model of physical architecture, we choose the highest abstraction level possible. The abstract physical architecture model contains five main elements: Computing node, communication node, communication interface, communication link, and route . They represent hosts configuration, hosts capacities and network topology.

A computing node communicates with its environment through communication interfaces. It can contain software components. By default, we consider that it has an appropriate middleware. A software component can be deployed, executed and managed on it. However, architects can specify the middleware properties.

A communication node also communicates with its environment through communication interfaces. It could not contain any software component. But an architect could declare route between two of its own communication interfaces. A route specifies that all messages that arrive into a communication interface are forwarded towards the "routed" communication interface.

Finally two communication interfaces could be bound. The communication link represents the ability of a communication interface to send or receive a message to or from an other communication interface. The communication link can only bind two communication interfaces.

Therefore, in SafArchie studio, we define a diagram for modelling the network topology.

4.4 Mapping a logical architecture to a physical architecture

Starting from a logical architecture, this step consists in mapping the component or composite instances onto the computing nodes of the physical architecture.

When architect finishes his/her mapping, he/she can check the accordance of the connection declared in the logical model architecture with the communication link declared in the physical architecture. For each connection, a route must exist between the computing nodes that contain the connected component instances.

4.5 Next features of SafArchie Studio: Supervision of a software architecture

The next step consists in building the run-time perspective. The goals of this perspective consist in following the software architecture evolution. This work is still in progress. The software architecture supervision could be synchronous or asynchronous. In synchronous mode, we follow the architecture evolution in *debug* mode. Each architecture evolution should be in accordance with the architecture type (creation of component instances or connection). A *watch dog* component will be in charge of this checking task. In asynchronous mode, we follow the architecture evolution in order to keep a global view of software architecture. But the architecture model is only notified of the evolution, it can not interfere with it.

In other hands, lots of work should be done to add the support of evolution in physical architectures. This feature is essential to use SafArchie in an ubiquitous computing context where the physical topology changes.

5. Related work

Nowadays, most of industrial projects use informal notations (boxes and arrows) for specifying software architecture. These notations are ambiguous, imprecise, unanalyzable. Due to the lack of tools, software architecture management is expensive and time consuming. On the other hand, software architecture research community focuses on building formal notations in order to define structure and behavior of software architecture. They are recognized in Architecture Description Languages (ADL) domain. An ideal architectural description language would provide[12]: A composition model that provides operators for composing existent libraries of components and connectors. An abstraction level for defining components and connectors with only useful details putting in evidence some properties. A potential reusability for existing components, connectors, and whole architectures in order to develop new components, connectors and system architectures. Finally a set of analysis for validate architectural descriptions.

Currently, no ADL or architecture software tool responds to all these goals. In this section, instead of a catalog of ADL characteristics, we focus on three

significant directions for software architecture definitions. SafArchie Studio is highly inspired by the main characteristics of these works: the specification and the analysis of dynamic distributed software behavior for Darwin[17] or Wright[3], the strong link with the implementation for ArchJava[2] or Fractal [7], and the building of architecture-driven software development environment for ArchStudio[10].

Some ADLs, such as Wright[3] and Darwin[17], support the specification and analysis of relatively complex component communication protocols. They provide formal basis for architectural description. They can be used to provide a precise, abstract, meaning to an architectural specification and to analyze a component assembly.

ADLs as Fractal[7] or ArchJava[2] are implementation oriented. Lots of existing approaches decouple implementation code from architecture. ArchJava is an extension to Java that seamlessly unifies software architecture with implementation, using a type system to ensure that the implementation conforms to architectural constraints. Fractal is generic software composition framework. As ArchJava, it is based on a hierarchical component model. Components can be nested in composite components - hence the "Fractal" name. Besides, this model is reflexive, i.e. components have full introspection and intercession capabilities. Finally, the Fractal component model is language independent, and fully modular and extensible. Fractal provides an XML based Architecture Description Language. Contrary to Darwin or Wright, ArchJava and Fractal ADL provide no behavior specification. The composition analysis is only structural. However, the abstract Fractal's or ArchJava's component models are efficient and adapted for programming step. In SafArchie, we follow their models to keep a strong link between the specification and the component platform. Moreover, we add a physical platform specification with the deployment perspective.

On the fringes of ADLs, other projects aim at improving the use of software architecture concepts in software engineering. For example, ArchStudio[10] mainly developed by the Institute for Software Research at the University of California, Irvine, is an architecture-driven software development environment. Indeed, while most development environments, like Microsoft Visual Studio and IBM Eclipse are code-driven development environments, ArchStudio focuses on software development from the perspective of software architecture.

SafArchie Studio is higly inspired by these three aproaches. It provides a complete tool suite to build, analyse and deploy a software architecture and to transform ADLs in an effective vehicle for communication and analysis of a software system.

6. Conclusion

This paper highlights our research in architectural modelling, refinement, and consistency checking. As part of architectural modelling, we define a three-view approach. The first specifies the software architecture type. It defines invariant properties for a software architecture. From this structural and behavioral specification, we analyze the component assembly. The second perspective specifies

a typed software architecture, i.e a component graph deployed on an abstract physical architecture. Finally, the run-time perspective gives administrator a global view of his/her system for the architectural supervision. For consistency among the different views, we use SafArchie's ability to provide a common component baseline to link those perspectives.

For the moment, we have tooled many of the concepts discussed in this paper and created a tool suite: SafArchie Studio. This tool is a set of new extensions for ArgoUML. It gives programmers, administrators, or designer an architectural view of software with different perspectives.

SafArchie approach is also the foundation for an other project: TranSAT[5] [6] (Transformation of Software ArchiTecture). Indeed, most of the time, the software architecture models are unsuitable for improving features of software systems. So, adding new concerns in software architecture is often a difficult and manual operation for architect. Moreover, the rules of integration are not described, saved, or analyzed. TranSAT focuses on these lacks of software architecture models. It proposes a framework for designing software architecture with step by step refinements: from an architecture which contains only business concern to a global architecture which contains business and technical concerns. The main idea of this project is inspired from Aspect Oriented Software Development (AOSD) [16] concepts.

TranSAT ensures three main features: It defines a mechanism to add new concerns in software architecture specification. It provides a description model for software architecture which saves the integration rules of a specific concern. It specifies rules which guarantee the correct integration of a technical concern inside a business model. Associated with TranSAT, SafArchie provides an architecture centric approach to build consistent software step by step.

References

[1] J. Aldrich, C. Chambers, and D. Notkin. Architectural Reasoning in Archjava. In *Proceedings ECOOP 2002*, volume 2374 of *LNCS*, pages 334–367, Malaga, Spain, june 2002. Springer Verlag.

[2] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting Software Architecture to Implementation. In *Proceedings of the 24th International Conference on Software Engineering (ICSE-02)*, pages 187–197, New York, May 19–25 2002. ACM Press.

[3] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, Janvier 1997. Issued as CMU Technical Report CMU-CS-97-144.

[4] R. Allen, R. Douence, and D. Garlan. Specifying Dynamism in Software Architectures. In *Proceedings of the First Workshop on the Foundations of Component-Based Systems*, Zurich, Switzerland, Septembre 1997.

[5] O. Barais and L. Duchien. SafArch : Maîtriser l'Evolution d'une Architecture Logicielle. In *Langages, Modèles, Objets - Journées Composants (LMO-JC'04)*, Lille, France, Mars 2004. Hermès Sciences.

[6] O. Barais, L. Duchien, and R. Pawlak. Separation of Concerns in Software Modeling: A Framework for Software Architecture Transformation. In *IASTED International Conference on Software Engineering Applications (SEA)*, pages 663–668, Los Angeles, USA, november 2003. ACTA Press. *ISBN 0-88986-394-6.*

[7] E. Bruneton, T. Coupaye, and J.B. Stefani. The Fractal Component Model, version 2.0-3. Online documentation http://fractal.objectweb.org/specification/, February 2004.

[8] *F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López, and G. Puebla.* The Ciao Prolog System. *Technical University of Madrid, June 2002.*

[9] *S. C. Cheung, D. Giannakopoulou, and J. Kramer. Verification of Liveness Properties using Compositional Reachability Analysis. In* Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering, *pages 227–243. Springer-Verlag New York, Inc., 1997.*

[10] *E.M. Dashofy, A. Van Der Hoek, and R.N. Taylor. An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. In* Proceedings of the 24th International Conference on Software Engineering (ICSE2002), *Orlando, Florida, 2002.*

[11] *L.G. DeMichiel. Entreprise Javabeans Specification, version 2.1. Online documentation http://java.sun.com/products/ejb/docs.html, June 2002.*

[12] *D. Garlan and M. Shaw. An Introduction to Software Architecture. In V. Ambriola and G. Tortora, editors,* Advances in Software Engineering and Knowledge Engineering, *volume 1, pages 1–40. World Scientific Publishing Company, 1993.*

[13] *G. Heineman and W. Councill, editors.* Component-Based Software Engineering, Putting the Pieces Together. *Addison-Westley, 2001.* ISBN: 0-201-70485-4.

[14] *H. Hussmann, F. Finger, and R. Wiebicke. Using Previous Property Values in OCL Postconditions - An Implementation Perspective. In* International Workshop UML 2.0 - The Future of the UML Constraint Language OCL, *York, UK, October 2 2000. http://dresden-ocl.sourceforge.net/.*

[15] *J.M. Jezequel and B. Meyer. Object Technology: Design by Contract: The Lessons of Ariane.* Computer, *30(1):129–130, January 1997.*

[16] *G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In Mehmet Aksit and Satoshi Matsuoka, editors,* Proceedings ECOOP, *volume 1241, pages 220–242. Springer-Verlag, 1997.*

[17] *J. Magee. Behavioral Analysis of Software Architectures using LTSA. In* Proceedings of the 21st international conference on Software engineering, *pages 634–637. IEEE Computer Society Press, 1999.*

[18] *J. Magee and J. Kramer.* Concurrency. State Models and Java Programs. *wiley, 1999.*

[19] *J. Magee, J. Kramer, and D. Giannakopoulou. Behaviour Analysis of Software Architectures. In* 1st Working IFIP Conference on Software Architecture (WICSA1), *San Antonio, TX, USA, 22-24 February 1999.*

[20] *B. Meyer. Applying "Design by Contract".* Computer, *25(10):40–51, October 1992.*

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

[22] *OMG. CORBA Component Model, v3.0. Online documentation http://www.omg.org, June 2002.*

[23] *OMG. OMG Object Constraint Language specification, version 2.0. http://www.omg.org/docs/ptc/03-10-14.pdf, October 2003.*

[24] *J.E. Robbins and D.M. Hilbert D.F. Redmiles. Using Critics to Analyze Evolving Architectures. In* Second International Software Architecture Workshop held at SigSoft'96, *1996.*

[25] *R. Soley. MDA, an introduction. Online documentation http://www.omg.org/mda/mda_files/Soley-MDA/MDA-Seminar-Soley.htm, 2002.*

[26] *C. Szyperski.* Component Software: Beyond Object-Oriented Programming. *ACM Press and Addison-Wesley, New York, NY, 1998.*

[27] *L. Tolke and M. Klink. Cookbook for Developers of ArgoUML, an Introduction to Developing ArgoUML, March 2004.*