A contractual approach for the verification of UML2.0 software architectures

Taoufik Sakka Rouis*
Cristal Laboratory, ENSI,
University of Manouba,
Manouba, Tunisia
Email: srtaoufik@yahoo.fr
*Corresponding author

Mohamed Tahar Bhiri
MIRACL Laboratory, ISIMS,
University of Sfax,
Sfax, Tunisia
Email: tahar_bhiri@yahoo.fr

Mourad Kmimetch
UR-OASIS Laboratory, ENIT,
University of Tunis El Manar,
Tunis, Tunisia
Email: mkmimetch@gmail.com

Faouzi Moussa
Cristal Laboratory, ENSI,
University of Manouba,
Manouba, Tunisia
Email: faouzimoussa@gmail.com

Abstract: The functional and qualitative properties are conventionally considered after software is completed. Currently, researchers consider treating those properties as soon as the architectural design phase. In this paper, the modelling and verification of the syntactic, behavioural and qualitative properties in UML2.0 software architectures are studied. To achieve this, a UML profile extending the UML2.0 component model by a set of qualitative concepts is proposed. The new profile, called CUMLQoS, is able to model the UML2.0 software architectures equipped with qualitative properties. Our verification approach suggests using the Acme/Armani ADL as a checking machine of syntactic and qualitative properties of UML2.0 software architectures deriving from our CUMLQoS profile. The choice of this ADL is justified by its ability of formal verification of different types of properties related to software architectures. As a second step of our verification approach, UML2.0, Port State Machine (PoSM), Wright and CSP are combined to check the behavioural consistency of UML2.0 software architecture. To achieve this, a set of systematic rules is proposed allowing the translation of UML2.0 source model to the Wright target model. Using Wr2fdr tool, the Wright specification can automatically be translated to a CSP specification acceptable by the FDR2 model-checker.

Keywords: software architecture; verification; contract; UML2.0; PoSM; model-checker; FDR2.


Biographical notes: Taoufik Sakka Rouis is a PhD student in Computer Science at Manouba University. He received his Master’s degree in Computer Science from the University of Sfax in 2010. His main research interests are component-based software engineering and formal verification of software architectures.
Mohamed Tahar Bhiri is a Professor at the Faculty of Sciences of Sfax (Tunisia). He led numerous studies on the subject of object, component and transformation models. He uses the formal models such as Wright, Acme, B, Event-B and CSP to check the consistency of the UML2.0 diagrams. Currently, he is interested in the integration of competition in the Event-B method.

Mourad Kmiche is an Associate Professor at the High School of Computer Science and Mathematics, Monastir University. He holds a PhD (2010) in software engineering from the University of Pau and Pays of the Adour, France. His research interests include service computing, service component architecture (SCA), services reliability/flexibility and formal verification.

Fatouzi Moussa is a Doctor Engineer in Computer Science and Professor at the Faculty of Sciences of Tunis (Tunisia). His main research interests are software engineering and the modelling and the formal verification of human–computer interfaces.

## 1 Introduction

Today, we witness a growing interest in the domain of Component-Based Software Engineering (CBSE). This interest is mainly motivated by the reduction of cost and development time of complex systems. In the CBSE approach, software architecture is based on the Architecture Description Languages (ADLs) (Taylor et al., 2009). The latter deal with the description, analysis and reuse of software architectures. Among the most well-known ADLs, we quote: Acme (Garlan et al., 1995), Rapide (Luckham et al., 1995), AADL (SAE, 2004) and Wright (Allen, 1997). In this paper, we accept the idea of considering the UML2.0 as an ADL (El Boussaidi and Mili, 2006). Indeed, the UML2.0 component model (OMG, 2005) introduces the major architectural elements such as component, connector and port. Each ADL is focused on a particular architecture family and offers a modelling language and a technical analysis which are highly specialised. For example, Acme is an exchange language between ADLs, and Rapide is focused on the specification and analysis of the dynamic aspect of architectures. Most of the tools supporting ADLs offer a structural analyser of architectural elements. However, rarely do these tools propose a behavioural and qualitative analysis of these elements. This amounts to the ambiguity of what an ADL should describe at the high level of abstraction.

In order to check the coherence – absence of contradiction – of software architecture, a contractual approach based on the assembly contracts is established between the server components and the client components. This distinguishes four levels of assembly contracts: syntactic contracts, semantic contracts, synchronisation contracts (behavioural contracts) and qualitative contracts. This inter-components contractual approach is seen as an extension to the design by contracts proposed by Meyer (1992), famous in the object-oriented world and supported by various languages such as Eiffel, OCL and JML (Java Modelling Language).

In this paper, an MDE approach allowing the verification of the syntactic, qualitative and behavioural contracts of a UML2.0 Software Architecture (SA) is proposed. To achieve this, the use of Acme ADL as a checking machine for the verification of the syntactic and qualitative contracts is suggested. To check the behavioural contracts, the translation of the UML2.0 SA to a Wright (Allen, 1997) specification is selected. The choice of this formal ADL is justified by the following main reasons:

- UML2.0 and Wright share the same architectural concepts such as component, connector, port, etc. This promotes the back and forth between these two ADLs.
- The Wright ADL is accompanied with a powerful tool called Wr2fdr (Hammami et al., 2012). The latter generates a set of CSP behavioural contracts related to a Wright architecture. This promotes the reuse of these contracts in a UML2.0 framework.

The paper is structured as follows: Section 2 presents the main related work; Section 3 describes our proposed approach; Section 4 introduces a UML2.0 profile that models the qualitative aspect in UML2.0 SA; Section 5 depicts the syntactic and qualitative verification approach. The behavioural verification is presented in Section 6; Section 7 exhibits a validation of our approach on an ATM/Bank system. Finally, Section 8 provides a conclusion and possible future work.

## 2 Related work

Most ADLs share the same architectural concepts such as components, connectors and configuration. One of the challenges of ADLs is their ability to perform the validation and/or verification of software system properties very early in their life cycle and throughout their development process. The approach shared by most of the existing works in the field of SA consistency verification is the use of techniques and general tools such as B (Abrial, 1996) and CSP (Hoare, 1985). To achieve this, numerous works offer more or less systematic translations of source component models to the target formalism. In this section, the contract classification proposed by Meyer (1992) is chosen to present the main works related to the verification of software architecture in ADL.

### 2.1 Checking syntactic contracts

Syntactic contracts include properties related to the compatibility of operation signatures (offered and required) and structural properties limiting connections between components.
These contracts depend on typing possibilities (predefined types, constructor simple type, constructor structured type, method redefinition and overloading) and composing component model rules. Component models that support constraint languages can specify the syntactic contracts, specifically the structural ones as invariant properties. These properties are verified with an evaluator of logical predicates. Among the ADLs supporting a logical predicate language we mention: UML2.0 (OMG, 2005), Fractal (Bruneton et al., 2006) and Acme (Garlan et al., 1995) that have, respectively, a constraint language: OCL, CCLJ and Armani. But, unlike CCLJ and in particular Armani, OCL is not dedicated to constraints on component models. Rather, it is designed to specify constraints on object-oriented models.

2.2 Checking of the semantic contracts

In traditional object-oriented programming, the semantic aspects have been a topic of great interest for a long period of time. These semantic contracts can be described through precondition and post-condition using a constraint language like OCL or Eiffel. But in the component approach, the semantic aspects are not so well supported and are often entirely left out.

Messabihi et al. (2010) propose to use the B formal method to check the semantic properties of a Kmelia SA (Attigbé et al., 2006). The Atelier B tool is used in Chouali et al. (2006) to prove compatibility between two interconnected UML2.0 component interfaces. Mouakher (2011) improves the solution proposed in Chouali et al. (2006) by checking the semantic compatibility between two interconnected interfaces. In this approach, the semantic aspect is specified by Port State Machine (PoSM). In addition, Mouakher (2011) suggests a solution based on the B refinement to verify compatibility between the abstract semantic of the components assembly and the assembled semantic obtained by a refinement of that of the system constituent components.

2.3 Checking of behavioural contracts

Behavioural contracts can be specified by formalism based on process algebra, labelled transition systems and Protocol State Machine (PSM). The verification of these contracts is often entrusted to model-checkers. Rare are the ADLs that allow an explicit description of the behavioural property. That amounts to the ambiguity of what an ADL should describe at the behavioural level. In this section, the main ADLs supporting this behavioural property are presented.

Oquendo (2004) proposes a formal language with a solid theoretical foundation. This language called Pi-ADL is a well-trained extension to Pi-Calcul (Milner, 1999; Lanese et al., 2016) aimed at the structural and behavioural descriptions of component-based software architectures. The behavioural aspect of a Pi-ADL architectural element is expressed using scheduled actions. As Pi-Calcul is complete in the Turing sense, any behavioural aspect can be modelled using Pi-ADL. Additionally, Pi-ADL offers structural and behavioural properties. The latter are expressed in the AAL ArchWare analysis language which is able to check these properties. Leroux et al. (2012) propose to use the Input-Output Symbolic Transition System (IOSTS) for the verification of the structural and behavioural implementation conformity with its architecture described in Pi-ADL.

In the Tracta project, Giannakopoulou (1999) suggests an extension of Darwin ADL allowing the addition of a behavioural specification of primitive components and a set of properties within an architectural description. This behavioural information is described at a high-level language derived from CSP (Hoare, 1985) called FSP (Kramer, 1999). The LTSA analysis tool detects the presence of a deadlock and checks these FSP properties.

Although Fractal (Bruneton et al., 2006) does not provide a mechanism for specifying the behavioural aspects of architectural elements, various ongoing works come to extend this model through different contract levels. For example, Collet et al. (2005) are working on defining assertions at the Fractal component interfaces. The Labelled Transition Systems (LTS) are used in Barros et al. (2005) to specify the behavioural aspects of Fractal components.

The AADL (SAE, 2004) international standard provides two fundamental mechanisms (property and annex) that can be used for specifying any notion according to the needs of the user. These mechanisms are used by Vergnaud et al. (2005) to specify the components behaviour, but there are no semantics defined in the standard to detect problems with this behaviour. Chkouri (2010) deals with this problem by providing a translation of AADL to BIP, which has formal operational semantics defined in terms of Labelled Transition Systems (LTS). This translation allows the analysis and detection of potential deadlocks and the verification of different properties.

The works presented in Hillah et al. (2010) and Reza and Chatterjee (2014) suggest opening AADL on Petri nets to verify certain behavioural properties. The main weakness of AADL is the lack of rigorous operational semantics.

2.4 Checking of qualitative contracts

In software engineering, a quality, also called non-functional property or QoS, is a general term that covers a system performance, as opposed to system operation (i.e. functionality). Chung et al. (1999) presented more than 160 qualities, among which we quote: reliability, availability, usability, efficiency, safety, maintainability, portability, etc. The ISO-9126 (ISO, 2001) standard proposes a framework for the specification and evaluation of software quality. This framework is composed of four parts as follows: quality model, quality in use, external and internal quality. The first part presents a framework for quality specification, while the three other parts specify how such specifications can be measured during the execution.

Few ADLs provide mechanisms to describe qualitative contracts. The AADL (SAE, 2004) model introduces the concept of property. For each component, we can associate
properties and give them values. The Open Source AADL Tool Environment (OSATE) has already been successfully used in Delange and Feiler (2014) to validate several operational qualities such as Safety, Security, Performance or Latency. The Acme ADL (Garlan et al., 1995) offers facilities for the description of the qualitative properties using the Property concept. Also, the Armani predicate language coupled with the Acme ADL allows specifying the qualitative contracts of an Acme software architecture. Apart from the predicate evaluator Armani, which is a part of the AcmeStudio platform, Acme does not support specific tools for analysing those qualitative properties.

2.5 Results

The UML2.0 components model (OMG, 2005) does not describe all aspects of an SA based on components. It needs other formalisms more or less integrated into UML2.0 as PoSM to specify the behavioural aspects. Several qualitative languages are proposed to attach the UML2.0 components by their qualities. For example, Aagedal (2001) proposes the CQML language and Defour et al. (2004) suggest the QoSCL language to specify the qualitative aspects. No ADL covers the various components of these contracts. For example, Wright, Darwin and Pi-ADL allow the description and verification of behavioural contracts, while AADL and Acme models allow the description and verification of qualitative contracts.

Table 1 summarises the ability of the studied component models vis-à-vis the specification and verification of contracts.

<table>
<thead>
<tr>
<th>Contract</th>
<th>Component model</th>
<th>Checking tool</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntactic</td>
<td>UML2.0/OCL, Fractal/CCLJ, Acme/Armani</td>
<td>Evaluator of predicate logic</td>
<td>OCL is adopted on OO model</td>
</tr>
<tr>
<td>Semantic</td>
<td>Kmelia (\rightarrow) B</td>
<td>B proves</td>
<td>Gap between the source and target models</td>
</tr>
<tr>
<td>Behavioural</td>
<td>Fractal/LTS, Darwin/FSP, AADL/Petri Net</td>
<td>CADP, LTSA</td>
<td>PSM can describe only one direction of communication, Wright proposes a standard contracts</td>
</tr>
<tr>
<td></td>
<td>Pi, ADL, Wright/CSP</td>
<td>ArchWare, FDR</td>
<td></td>
</tr>
<tr>
<td>Qualitative</td>
<td>AADL/proper (property concept)</td>
<td>Plugin, OSATE</td>
<td>AADL proposes specific qualities such as safety and security</td>
</tr>
<tr>
<td></td>
<td>Acme/Armani (property concept)</td>
<td>AcmeStudio</td>
<td>It offers a powerful evaluator of predicate logic</td>
</tr>
</tbody>
</table>

3 Proposed approach

In recent years, the Unified Modelling Language (UML) has emerged as a de facto industrial standard for describing both the structure and behaviour of software systems. It is simple, intuitive and easy to understand. Furthermore, its graphical notations have been extensively applied to design systems in various application domains, from small embedded-software systems to huge distributed systems. In addition, the UML2.0 standard introduces a number of new concepts and refines a number of existing ones for modelling software architectures. Indeed, the UML2.0 component model introduces the major architectural elements such as component, connector and port. The introduction of these concepts and the distinction between provided interfaces and required interfaces allow an interesting set of notations for the modelling of software architectures.

But the UML2.0 component model does not support a notation permitting the modelling of all aspects of software architectures. As a solution for this problem, we propose to use the extensibility strategy offered by the OMG. Indeed, we define a new UML profile extending the UML2.0 component model by a set of qualitative concepts. This profile, called CUMLQoS, is able to model UML2.0 SA equipped with their qualitative properties.

Seeing that UML is a semi-formal language, it comprises several notations with no rigorous semantics. Therefore, it is not possible to apply a rigorous analysis on a UML model. To deal with this defect, we advocate, in this paper, to opening the UML2.0 ADL on the formal ADLs: Acme and Wright (see Figure 1). The source model and the two target models rely on the component paradigm. This promotes the continuity and reversibility (backtracking to correct errors) between UML2.0 on one side and Acme and Wright on the other side.

Hence, we propose to use the Acme ADL as a checking machine of the syntactic and qualitative contracts of the UML2.0 SA derived from our CUMLQoS profile. The choice of this Acme ADL is justified by its richness and its ability of typing, which makes it possible to define new types of architectural elements (Components, Connectors, Port, Role, etc.) and properties specific to these elements. Moreover, the Acme ADL is coupled with a powerful predicate language called Armani. The latter allows constraints specification in the form of invariants and heuristics. In addition, Acme/Armani is supported by the AcmeStudio environment. This allows analysing and evaluating the syntactic and qualitative constraints specified in Armani. In this work, we do not address verifiable qualitative properties when executing a component assembly. Indeed, our objective is to verify statically the consistency of a component assembly from the architectural analysis phases.

To describe the behavioural view of a component or of its interfaces or ports, UML2.0 proposes four behaviour specification mechanisms (subtypes of behaviour) to provide concrete semantics of the generic behaviour model; the subclasses provided are Interaction, Activity, State Machine (SM) and its specialisation PSM. These SMs are used to describe the component behaviour as part of its implementation, while the PSMs define the protocol interfaces which can be considered as a specialisation of the UML state machines without action or activity. The PSM can describe only one
direction of communication. It also means that PSM cannot describe the communication relationships between the requested and provided interfaces. Mencl (2004) proposes an extension of UML2.0 PSM called PoSM. The main contributions of this PoSM model are the ability to define the required and provided behaviour and the possibility to differentiate between a request and a response. An operation call of a component which is inherently non-atomic is modelled in PoSM with two atomic events: a request (corresponding to the start of the call) and a response (corresponding to its termination). However, none of the tools supporting UML2.0 offers behavioural analysers. This gives birth to works that open UML on formal languages to verify the behavioural consistency of UML diagrams. For example, Luong et al. (2012) and Lambolais et al. (2009) propose to translate UML2.0 state machines towards an LTS specification verifiable by LOTOS tool. Garis et al. (2012) propose to formalise a PSM in Alloy lightweight formal modelling language. Using Alloy analyser, this formalisation can be used to simulate and verify consistency between UML artefacts such as components. The formalisation of PSMs is implemented using the model transformation language ATL. Jacobs and Simpson (2014) propose to translate the UML sequence diagrams to a CSP formal specification verifiable by the FDR2 model-checker.

In a second step of our verification approach, we target the behavioural verification of UML2.0 software architectures described by CUMLQoS profile endowed with a behavioural specification described by PoSM. To achieve this, the model transformation approach is used to translate the CUMLQoS architectural elements into Wright specification. This formal ADL is opted for as an intermediate modelling language. The choice of this ADL is justified by the existence of the Wr2fdr tool. The latter can translate a Wright specification into another CSP acceptable by the FDR2 model-checker. The Wr2fdr tool generates a set of contracts aimed at connecting behavioural compatibility and compatibility between the interfaces of a component and the component itself.

**Figure 1** Verification approach (see online version for colours)

4 The CUMLQoS profile

The UML2.0 standard does not support a notation permitting the modelling of the qualitative aspects of software architectures. As a solution to this problem, several researchers take into consideration all the possibilities leading to the increase of the UML2.0 metamodel. This strategy offered by the OMG is used in different qualitative languages such as QML, CQML and QoSCL. These latter propose new UML2.0 metamodels to modelling the qualitative aspect of UML2.0 software architectures. But all these metamodels do not address the verification of qualitative contracts between the interconnected components. This is due to the fact that all tools supporting UML2.0 do not offer evaluators dedicated to components. Even tools supporting OCL are not yet amenable to express constraints on component models. Indeed, OCL is rather designed to specify constraints on object-oriented models.

However, the first strategy offered by OMG has another problem: the increase of the UML metamodel gives a new modelling language supporting natively the architectural concepts. But this involves the following two major drawbacks:

- The new UML version is becoming increasingly complex. This has an impact on the ease of the use of this language.
- The new UML version loses its standard character and, therefore, becomes incompatible with the existing UML platform.

Another strategy offered by the OMG is to extend UML2.0 models. This strategy – called Profile – is based primarily on extensibility mechanisms provided by the language such as stereotypes, tagged values and constraints. The use of UML profiles to adapt to the qualitative properties is an important advantage:

- Explicit representation of qualitative properties without changing the UML2.0 metamodel.
- These profiles can be manipulated through a platform supporting the concept of UML profiles.

In this study, the second strategy will be used. Indeed, we proposed a profile called CUMLQoS (see Figure 2) that extends the UML2.0 component model to include specific components with their qualitative properties. These extensions proposed in this profile minimise the problems of UML2.0 regarding the description of qualitative aspects. All quality concepts specified in this profile are inspired by quality standards like ISO-9126 (ISO, 2001) and metrics standards like IEEE-1061 (IEEE, 1998).

In this profile, a component stereotyped by ‘ComponentQoS’ represents a simple component that can propose ports stereotyped by ‘PortQoS’. Every ‘PortQoS’ port is typed by functional interfaces and qualitative interfaces. Functional interfaces exhibit functional services offered and/or required by the component, while qualitative interfaces ‘InterfaceQoS’ exhibit the qualities offered and/or required by the component. To distinguish between a required quality and another offered,
we proposed two associations between the ‘PortQoS’ element and the ‘InterfaceQoS’ element: one to represent the qualities required by the component through this port and another to represent the qualities guaranteed by the component through this port.

According to the ISO-9126 standard (ISO, 2001), quality and characteristic constitute the most fundamental terms to express a qualitative contract. In our profile, each ‘Quality’ is characterised by a couple (property, weight), where ‘property’ is a qualitative characteristic ‘Characteristic’ and ‘weight’ is the value of this characteristic, where the ‘Characteristic’ entity represents a non-functional aspect such as performance, reliability, availability, etc.

Each ‘Characteristic’ is identified by its variance (‘increasing’ or ‘decreasing’) and its unity. The ‘decreasing’ variance means that the decrease in the characteristic value allows the increase in the service quality and vice versa for ‘increasing’ variance which means that the increase in the characteristic value allows the increase in the service quality.

Figure 2 The CUMLQoS profile

To improve the readability of our CUMLQoS profile, we profitably took the opportunity to change the symbol of some elements proposed by UML to represent a quality interface ‘InterfaceQoS’ by an icon if this interface requires qualities and by an icon if this interface offers qualities. Figure 3 shows a representation of a component called Server endowed with a simple port. The latter provides an ‘IRequest’ interface and requires two interfaces: ‘IResponse’ and ‘IQoS’. This ‘IQoS’ interface regroups the qualities required by the port.

In order to ensure the consistency of a UML2.0 SA that derives from the CUMLQoS profile, a set of syntactic and qualitative contracts between interconnected interfaces must be specified. These contracts are informally specified in the next section.

5 Syntactic and qualitative verification

The Acme ADL offers architectural concepts such as component, connector, role, port, representation, system and family. In addition, the Acme Property concept can attach different details to the architectural elements. This concept can be used in the description of the qualitative properties of the software architecture. Similarly, the Acme ADL supports the notion of Type. We can define types of architectural elements. Furthermore, Acme offers easiness for the description of qualitative properties using notably the Property and Type concepts. In addition, Acme provides an enough powerful language predicate called Armani with functions suitable to the domain of software architecture. The Armani language allows describing the architectural constraints in the form of an invariant or heuristic attached to any architectural element (component, family, system, connector, etc.). Such Armani constraints are executable in the AcmeStudio environment.

In this work, we used this AcmeStudio tool as a verification machine of the syntactic and qualitative properties of UML2.0 software architectures. For this, we proposed an Acme/Armani style called SUMLQoS (see Figure 4) that allows the modelling of UML2.0 SA derived from our CUMLQoS profile by Acme/Armani configurations. In order to ensure the structural coherence between the source model and the target model, a set of coherence rules must be modelled in the target family level. These structural rules can be modelled in Acme using the invariant concept. Informally, these rules can be specified as follows:

- All ports of a ComponentQoS must be of a PortQoS type.
- An interface of a PortQoS must be a functional interface ‘InterfaceFunc’ or a qualitative interface ‘InterfaceQoS’.
- Each PortQoS must be obligatorily equipped with functional interfaces and can have qualitative interfaces. Hence, we cannot obtain a PortQoS without a functional interface.
- A UML2.0 assembly connector is a binary connector.
5.1 Formalisation of syntactic contracts

Many rules relative to an assembly of UML2.0 components can be modelled by invariant properties. These rules are defined at the metamodel level (style or family level). Such rules of coherence allow verifying the syntactic properties such as:

- accepted_components: This rule stipulates that only the components of ComponentUML type are accepted in a UML2.0 software architecture.
- accepted_connectors: This rule stipulates that only the assembly connectors of the ConnectorUML type are accepted in a UML2.0 SA.
- interface_required_satisfied: This rule stipulates that each required interface must be satisfied.
- caller callee: This rule states that the caller and callee must be different in a component assembly. That is to say that each connector of assembly is a binary connector and each connector attaches an interface required at another provided.

Figure 5 illustrates the formalisation of these different syntactic constraints in Acme/Armani. A non-respected property indicated by a predicate (Boolean expression following the reserved word invariant) evaluated to false necessarily reflects a structural architectural error that should be corrected by the architect.

5.2 Formalisation of qualitative contracts

We benefited from using these possibilities offered by Acme/Armani in order to enrich the SUMLQoS style by two new types of properties (Characteristic and Quality) allowing the formalisation of the QoS properties of UML2.0 components in Acme/Armani. These two types of properties are inspired by our CUMLQoS profile previously presented.

5.2.1 Formalisation of the characteristic concept

The characteristic concept is the base construction of all qualitative specification. This characteristic represents a non-functional aspect such as performance, reliability, availability, etc. We can formalise a quality characteristic by an Acme/Armani property. This property must be modelled with a record type made up of three fields:

- Name: It represents the name of the characteristic (Performance, Availability, etc.). This field can be modelled by a property of string type.
- Variance: It models the direction (increasing or decreasing) of the characteristic. This field must be of an enumerated type (enum {increasing, decreasing}).
- Unity: It models the unit of the characteristic, if it exists.

Figure 6 illustrates the formalisation of the Characteristic concept by a property type in Acme/Armani. This new type represents a base type of all non-functional characteristics.
5.2.2 Formalisation of the quality concept

The Quality concept specifies a non-functional property proposed by a qualitative interface. We can formalise a Quality concept by a record type composed of two fields:

- Property: It models the qualitative characteristic of this quality. This field has to be of the ‘Characteristic’ type.¹
- Weight: It models the value allowing the restriction of the characteristic of this quality. This field has to be a real type since the latter puts together all numerical types.

Figure 7 illustrates the formalisation of the Quality concept in Acme/Armani. This Quality is modelled with a property type which represents a base type of all qualities formalised in Acme/Armani.

**Figure 7** Formalisation of quality concept (see online version for colours)

```
property type Quality = record
    [ Property : Characteristic;
      Weight : float ];
```

The SUMLQoS style described above provides a predetermined structure that brings to Acme/Armani a software architecture described by an assembly of UML2.0 components derived from our profile CUMLQoS. The translation of UML2.0 to Acme/Armani could be automated using an MDE approach (Model-Driven Engineering): Development of a metamodel CUMLQoS, metamodel SUMLQoS and an expression of transformation rules CUMLQoS to SUMLQOS using a model transformation language such as ATL.

In order to verify the qualitative coherence of a UML2.0 software architecture source, we proposed to extend the SUMLQoS style by an Armani contract (see Figure 8) defined in an informal way as follows:

CQuality: All the qualities required by a component have to be ensured by the components attached to the latter. A required quality RQ is said ensured by another provided quality PQ if and only if:

- The two qualities relate to the same quality characteristic;
- The weight of the characteristic in the RQ quality is:
  - Higher than the PQ quality if the characteristic variance is downlink ‘decreasing’.
  - Lower than the PQ quality if the characteristic variance is upward ‘increasing’.

This CQuality constraint and the last constraints specified above will be evaluated at the configuration level (system) deriving from the architectural style. In these constraints, we successfully used the predefined sets offered by Armani such as Components and Properties, respectively, denoting the components set of the running configuration that derives from our SUMLQoS style (self.Components) and the set of properties attached to a component belonging to self.Components. In addition, we judiciously used both Armani quantifiers for all (‘∀’) and exists (‘∃’).

**Figure 8** Formalisation of the qualitative contract (see online version for colours)

```
rule CQuality = invariant
   forall Comp1, Comp2: ComponentQoS in self.Components |
   Connected (P1, P2) ->
   forall I1:InterfaceQoS in {select I1: P1.properties| I1.typeInterface == required} |
   exists I2:InterfaceQoS in {select I2: P2.properties| I2.typeInterface == provided} |
   forall RQ: Quality in I1.properties |
   exists PQ: Quality in I2.properties |
   RQ.Property == PQ.Property and
   RQ.Property.Variance== increasing -> (RQ.Weight <= PQ.Weight) and
   RQ.Property.Variance== decreasing -> (RQ.Weight >= PQ.Weight);
```

6 Behavioural verification approach

The Wright ADL provides the basic architectural concepts such as: component, connector, configuration and style. In Wright, a component (respectively a connector) may be provided with one or more interfaces called ports (roles respectively). Wright was one of the first approaches to allow the description of the behaviour of architectural elements. The behaviour of a Wright component (respectively of a connector) is described locally through the ports (respectively roles) and, generally, through a computation (glue respectively) using a type of CSP process algebra. In addition, Wright defines 11 standard properties related to the consistency of software architecture, among which four – assimilated to behavioural contracts – are automated by the Wr2fdr tool accompanying the Wright ADL. The four behavioural properties are specified informally as follows:

- Port/Computation consistency: the port specification should be a Computation projection, under the condition that the environment obeys the specification of all other ports. Intuitively, this first property states that the component does not care about events not covered by the ports.
- Connector without deadlock: The glue of a connector interacting with roles must be without deadlock. The connector description must verify that the roles coordination by the glue is consistent with the expected component behaviour, knowing that a CSP process is said to be in deadlock situation when it can opt out at any event, but has not so far terminated correctly (by participating in the event $\$$). Conversely, a process is without deadlock if it can never be in deadlock situation.
A contractual approach for the verification of UML2.0 software architectures

- Role without deadlock: each role of a connector must be without deadlock. Another category of inconsistency is detectable as a deadlock situation when a role specification is itself inconsistent.
- Compatibility port/role: any port attached to a role must always continue its protocol in a direction that the role can have.

These properties are implemented by the Wr2fdr tool as a set of contracts aimed at connecting behavioural compatibility and compatibility between the interfaces of a component and the component itself. These contracts can automatically be checked by the FDR2 model-checker.

In this section, we propose a set of rules allowing translating a UML2.0 SA deriving from our CUMLQoS profile endowed with a behavioural aspect specified by PoSM state machines into a Wright/CSP specification.

6.1 Components translation

Regarding the static aspect, we considered to translate a UML2.0 componentQoS by a Wright component according to the following rules:

- The name of this Wright component is that of the UML2.0 componentQoS.
- Any port associated with a UML 2.0 componentQoS is translated by a Wright port with the same name.

For the formalisation of the behavioural aspect of a UML2.0 componentQoS in Wright/CSP, we proposed to translate those PoSM state machines by the CSP process. This systematic formalisation is inspired by Ng and Butler (2003), Jacobs and Simpson (2014) and Dong et al. (2008) binding, respectively, the state machine diagrams, the sequence diagrams and the activity diagrams to CSP. Specifically, we have offered to translate any state machine modelling the behaviour of a UML2.0 portQoS with a CSP process modelling the behaviour of the machine which models the Wright port corresponding to the Wright component. Similarly, the state overall behaviour of a UML2.0 componentQoS is translated into a CSP process modelling the computation behaviour of the corresponding Wright component.

A PoSM transition is a simple UML transition that models the events invoking operations on its own from other components. In a PoSM, each event is prefixed by one of the operators (! or ?) and suffixed by one of the operators (↑ or ↓). There, ?Oper stands for receiving an observed event Oper and !Oper for sending an initialised event Oper. A call of an operation Oper is captured with two atomic events, where the event’s name Oper has either the suffix ↑ for request or ↓ for response.

It can be concluded that for each operation call, four types of events can be present in a PoSM state machine. These types are: sending a request to execute an operation, receiving the request, sending a response and receiving the response. For this, four rules to translate a simple transition PoSM by a CSP event were proposed. The translation rules are the following:

- Sending a request to execute an operation, translated by an initialised event having the same name as this operation.
- Receiving a request to execute an operation, translated by an observed event having the same name as this operation.
- Sending a response to an execution of an operation, translated by an initialised event. The name of this event is that of the operation preceded by the R character (R to say Response).
- Receiving a response to an execution of an operation, translated by an observed event. The name of this event is that of the operation preceded by the R character (R to say Response).

A transition to a final state is translated by the successfully terminated event: TICK (noted in CSP by ⊥).

Transitions delayed by a compound state are translated with a Pcomp CSP process. The latter is composite with Pci sub-process separated by one of the following operators: |~| or []. The formalisation of this composition process is presented by the following formula:

\[
Pcomp = \sum \text{operator } Pci \text{ where operator is:}
\]

- []: if the choice between these transitions is an external choice. In other words, if these Pci are observed events (receiving a request or a response).
- |~|: if the choice between these transitions is an internal choice. In other words, if these Pci are initialised events (sending a request or a response).

6.2 Assembly connectors translation

Similarly to the Wright connector, a UML2.0 assembly connector represents a communication link between two elements (e.g. ports or interfaces). Hence, we propose to translate a UML2.0 assembly connector to a Wright connector. The name of the connector is obtained by the concatenation of the names of the two interconnected components. This connector must have two roles:

- The name of the first role is obtained by the concatenation of the ‘R’ character (to say Role) with the port name of the first component.
- The name of the second role is obtained by the concatenation of the ‘R’ character (to say Role) with the port name of the second component.

To ensure the validation of the fourth property related to the consistency of the attachment Port/Role (see Section 6), we suggest creating the CSP process modelling the behaviour of each role by the same CSP process specifying the behaviour of the port attached to this role. The CSP process modelling the Glue (the overall behaviour of the connector) will be written by a process compatible with its roles. This
ensures the second property (connector without deadlock). For this, we propose to associate, for each called method in the role, the following process:

Receiving a request -> sending this request -> receiving a response to this request -> sending of this response.

Example:

**Connector** BankAtm
**Role** Client = _verifyPin->RverifyPin->
TICK
**Role** Server = verifyPin->_RverifyPin->
TICK
**Glue** = Client.verifyPin ->_Server.
verifyPin ->
Server.RverifyPin->_Client.RverifyPin->
>TICK

### 6.3 Component instance diagram translation

An instance diagram of UML2.0 component deriving from our CUMLQoS profile is translated to a Wright configuration according to the following correspondence:

- Any UML2.0 componentQoS instance is translated to an adequate Wright component instance;
- Any assembly of two UML2.0 componentQoS instances is translate by:
  - An instance of the assembly connector that relates these two components;
  - An attachment of the portQoS of the first component with the first role of this connector; and
  - An attachment of the portQoS of the second component with the second role of this connector.

### 7 Validation

The ATM/Bank is a complex system. This study does not seek to probe into its detailed operation, but to illustrate some of the main concepts of a component-based system. Our system consists essentially of two components: an ATM that represents an automated silver teller machine, and a Bank component representing a subsystem of a bank. The first component is equipped with a port called PATM which requires an interface that requires three functional services (authenticate, consult and debit). The second has a port called PBank that provides an interface grouping four functional services (authenticate, consult, debit and deposit).

The operating system of these components is as follows: when inserting a bank card into an ATM, it must authenticate itself by sending a PIN code and the IBAN code of the card to the appropriate bank. At this time, the bank checks the validity of this information. If the codes are correct, the Bank component allows the ATM component to consult, debit or deposit from this account. To simplify this system, we ignore the part of ATM communication with the user. Moreover, the PATM port offers a qualitative interface IQoS1 that models the ATM component which requires a response time ≤ 15 msec, while the PBank port offers a qualitative interface IQoS2 that models the Bank component which can guarantee a response time ≤ 20 msec.

#### 7.1 UML2.0/PoSM modelling

Figure 9 shows a UML2.0 software architectural description of this ATM/Bank system modelled by an assembly of components derived from our CUMLQoS profile.

**Figure 9** UML2.0 architectural description of the ATM/Bank

The p behaviour and the overall behaviour of each component of this ATM/Bank system are specified in Figures 10 and 11 by state machines using the PoSM profile.

**Figure 10** PoSMs of the ATM components
7.2 Syntactic and qualitative verification

A UML2.0 software architecture derived from the CUMLQoS profile can be modelled by an Acme/Armani system drifting the SUMLQoS style. The latter ensures the consistency rules as well as the syntactic and qualitative constraints specified at the SUMLQoS style on the components assembly derived from the CUMLQoS.

The formalisation presented in Figure 12 gives the translation of the UML2.0 description of the ATM system in the form of an Acme/Armani system through the SUMLQoS style.

The AcmeStudio environment shows (see Figure 13) a graphical representation of the ATM/Bank system in Acme/Armani. The Armani evaluator constraints supported by this tool detect that the qualitative constraint ‘CQuality’ specified at the SUMLQoS level is evaluated to false. This necessarily results in a qualitative incoherence in the assembly of CUMLQoS components previously proposed.

7.3 Behavioural verification

Using the translation rules presented in Section 6, we got the Wright configuration (see Figure 14) corresponding to the translation of the UML2.0/PoSM component assembly of the ATM/Bank system presented above.

The Wr2fdr tool allows the translation of a Wright configuration to a CSP description which can be verified using the FDR2 tool. Figure 15 shows that the use of these tools on the UML2.0/PoSM SA presented above allows concluding that, although this assembly appears correct from the structural consistency between required and offered interfaces, it is nonetheless incorrect from a behavioural standpoint. This occurs as the PATM port of the ATM component is incompatible with that of the Bank component. Indeed, although both ports seem similar, the first accepts the sequence of execution methods (consult after debit), while the second does not allow it.

Figure 11 PoSMs of the bank component

![PoSM of PBank Port](image1)

![PoSM of Bank component](image2)

Figure 12 The ATM/Bank system in Acme/Armani (see online version for colours)

```plaintext
Component Bank : ComponentQoS = new
ComponentQoS extended with {
    Port PBANK : PortQoS = new PortQoS extended with {
        Property PBank : InterfaceFunc = [
            TypeInterface = provided;
            Services = { "authenticate", "consult", "debit", "deposit" };]
    }
    Property IQoS2 : InterfaceQoS = [
        TypeInterface = provided;
        QoS = { [ Property = [ Name= "ResponseTime" ; Variance= decreasing ; Unity= "msec" ; Weight=20; ]; ]; } }
    Connector Client_Server : ConnectorQoS = new
    ConnectorQoS extended with {
        Role Client = { }
        Role Server = { }
    }
    Attachment Bank.PBank to Client_Server.Client;
    Attachment ATM.PATM to Client_Server.Server;
}
```

Figure 13 Graphical representation of the ATM/Bank system in Acme/Armani

![Graphical representation of the ATM/Bank system](image3)
Figure 14 Wright specification of ATM/Bank system

```plaintext
Configuration ATM/BANK

Component Atm

Port PAtm = authenticate -> Rauthenticate ->
(TICK |~| _debit -> Rdebit -> TICK |~| _consult -> Rconsult ->
(TICK |~| _debit -> Rdebit -> TICK ))

Computation = PAtm.authenticate -> PAtm.Rauthenticate ->
(TICK |~| PAtm.debit -> PAtm.Rdebit -> (TICK |~| PAtm.debit ->
PAtm.Rdebit -> TICK ))

Component Bank

Port PBank = authenticate -> _Rauthenticate ->
(TICK [ ] consult -> _Rconsult -> TICK [ ])

debit -> _Rdebit -> (TICK [ ] consult -> _Rconsult ->
(TICK [ ])

deposit -> _Rdeposit -> (TICK [ ] consult ->
(RBank.Rconsult -> TICK [ ])

Computation = PBank.authenticate ->
_PBank.Rauthenticate -> (PBank.consult ->
PBank.Rconsult -> TICK [ ])

_PBank.debit -> _PBank.Rdebit -> (TICK [ ] PBank.consult ->
PBank.Rconsult -> TICK [ ])

_PBank.deposit -> _PBank.Rdeposit -> (TICK [ ])

_PBank.consult -> _PBank.Rconsult -> TICK [ ]

Connector AtmBank

Role RAtm = authenticate -> _Rauthenticate ->
(TICK |~| _debit -> Rdebit -> TICK |~| _consult ->
Rconsult -> (TICK |~| _debit -> Rdebit -> TICK ))

Role RBank = authenticate -> _Rauthenticate ->
(TICK [ ] consult -> Rconsult -> _debit -> Rdebit ->
(TICK [ ] consult -> Rconsult -> TICK [ ])

deposit -> _Rdeposit -> (TICK [ ] consult ->
(RBank.Rconsult -> TICK [ ])

Glue = RAtm.authenticate -> RBank.authenticate ->
_RAtm.Rauthenticate -> (TICK [ ] r1 [ ] r2 [ ] r3)

where |
r1 = RAtm.consult -> _RBank.consult ->
_RBank.Rconsult -> _RAtm.Rdebit -> TICK

r2 = _RBank.deposit -> RBank.deposit -> (TICK [ ] r1)

r3 = RAtm.debit -> RBank.debit -> RBank.Rdebit ->
_RAtm.Rdebit -> (TICK [ ] r1)

Instances

A : Atm
B : Bank
AB : AtmBank

Attachments

A.PAtm As AB.RAtm
B.PBank As AB.RBank

End Configuration
```

Figure 15 Checking the CSP description of ATM/Bank system

8 Conclusion

In this study, we came up with a process based on model transformation allowing the modelling and verification of the syntactic, qualitative and behavioural properties of UML2.0 software architectures. For this reason, we proposed to extend the UML2.0 component model with new concepts in order to model the qualitative aspects of UML2.0 components. Our qualitative concepts are inspired by quality standards like ISO-9126 (ISO, 2001) and metrics standards like IEEE-1061 (IEEE, 1998).

In order to complete our verification approach, we proposed, as a second step, an Acme style called SUMLQoS. The latter offers a pre-established structure to Acme/Armani software architecture described by an assembly of UML2.0 components derived from our profile CUMLQoS. The AcmeStudio environment assesses the syntactic and qualitative constraints specified at the style level. In addition, our contribution covers the verification of the behavioural contracts related to component and connector consistency, and port/role compatibility. To achieve this, systematic translation rules to Wright ADL have been developed. The verification of these behavioural contracts is entrusted to FDR2 model-checker through the Wr2fdr translator able to translate a Wright configuration to a CSP specification.

Currently, we are interested in the automation of our translation rules using an IDM approach. This requires the implementation of source metamodels, target metamodels and a translation rules expression using a model transformation language such as ATL.
References


Chkouri, M.Y. (2010) Modélisation des systèmes temps réel embarqués en utilisant AADL pour la génération automatique d'applications formallement vérifiées, Thèse de doctorat, Université Joseph Fourier. [In French]


**Note**

1 Considering that the property concept is a key word in the Acme language, the ‘Proprt’ word (without e) is used to describe the quality’s property field in Acme.