A template for formalising reliable Acme-based software architecture

Rida Mezghache* and Fadila Atil

LISCO Laboratory,
Department of Computer Science,
Badji Mokhtar University,
POB 12, 23000 Annaba, Algeria
Email: r_mezghache@yahoo.fr
Email: atil_fadila@yahoo.fr
*Corresponding author

Abstract: Acme/Armani is a declarative language based on first-order predicate logic. Acme supports the component and connector paradigm with types, as well as invariants and architectural styles. It also supports constraints. Our main goal in this work is to provide Acme with a new architectural style that fully supports component-based architectures. To this end, we propose first a formalised model based on Acme architectural elements to have no gap between the initial design and Acme semantics. Based on this model, we create an architectural style, and then we present architecture constraints, while outlining how they are specified and interpreted. We will firstly express these constraints in the first-order predicate logic, and then we will translate them in the Acme/Armani formalism, to ensure syntactic and composition conformance, which makes the configuration as reliable and consistent as design.

Keywords: component-based architectures; reliability; integrity constraints; software architecture; architecture description language; component; connector; configuration; Acme; Armani.


Biographical notes: Rida Mezghache is an Assistant Professor of Computer Science at ENSET SKIKDA. He is currently a researcher pursuing his PhD in the Department of Computer Science of the UBMA (Annaba University) in the field of software engineering with special focuses on software architecture and software reuse.

Fadila Atil is a Professor-Researcher in the Department of Computer Science at the University of Badji Mokhtar. She received her PhD in computer science from the same university. Her research interests lie in the general field of software engineering with special focuses on software architecture and software reuse. She participated as member at several research projects MERCI (Code B2302/02/04/95), PLAN (Code B2302/04/97), MEPL (Code B2301/06/99), Aspect One (Code B2301/03/2003), MODE-NP (Code B2301/80/06), then as chair of the project ERSYCO-I (Code B*01120080061) and actually the project SAFAXY-I (Code B*01120110012). This last turns around themes of software architectures and formal approaches in complex systems.

1 Introduction

The problem we are interested in this work is how to ensure a reliable component-based architecture. By reliable, we mean any compositions where all instances are correct in the sense that they meet designer’s requirements.

Today’s software products are large and complex, which led to the need of a new trend in software development, by introducing new methods and practices of software engineering.

Architecture-based software aims to help developers to abstract away unnecessary details and focus on the whole system. Software architecture helps not only to develop large complex applications but also to reduce cost and facilitate evolution (Medvidovic et al., 2002).

In order to provide an explicit usage of architectural-based software development process, a suitable specification language is required. Architecture Description Languages (ADLs) have been proposed for this purpose. ADLs provide abstractions of parts used for modelling large systems, reducing costs for detecting and removing of errors (Medvidovic et al., 2002). Those languages are formal and can be used to represent the architecture of a software intensive system (Clements, 1996).

‘An ADL for software applications focuses on the high level structure of the overall application rather than the
ADLs are crucial in software development and particularly in architecture-based software development. ADLs are different from each other, each of them follows a particular approach.

Some ADLs have been proposed for modelling architectures within a particular domain or as general-purpose architecture modelling languages. Some of the different ADLs that we found in literature are: LIEANNA (Tracz, 1993), Artek (Terry et al., 1995), MetaH (Binns et al., 1996), AESOP (Shaw and Garlan, 1996), SADL (Moriconi and Riemenschnieder, 1997; Moriconi et al., 1995), Weaves (Gorlick and Quilici, 1994; Gorlick and Razouk, 1991), Wright (Allen, 1997), AESOP (Garlan, 1995), Darwin (Magee et al., 1995), Acme (Garlan et al., 2000a), Rapide (Luckham et al., 1995), UNICON (Shaw et al., 1995) and C2 (Taylor et al., 1996).

ADLs are used to describe the elements of architecture and their composition, and they also set the constraints to which the system must conform.

In this paper, we formalise a reliable architecture configuration. In this purpose, we describe the constraints in the first-order predicate logic then we translate them using the architectural concept style of Acme and Armani detailed in Monroe (2001).

Since the architecture of a system influences nearly all of a software system’s quality attributes. That makes it relevant to develop support for efficient software quality at the architectural level.

We define this work as the base layer including architectural style, components and connectors and configuration constraints, in order to later support a runtime layer, with reconfiguration constraints, keeping the conformance from the architectural layer to the application layer (Le Goaer, 2009).

Beyond ensuring the consistency of the Acme model, according to Garlan et al. (2000b) and Monroe (2001) a formal specification for the Acme model can serve several purposes:

- To allow a formal verification of Acme design;
- To provide the basis of a formal architecture description language for Acme;
- To provide a more abstract, programming-language-independent specification of the Acme model;
- To allow a rigorous comparison with other component models;
- To allow a formal specification and verification of Acme tools;

The last is important because the Acme specification aims to define a very general component model, from which more specialised component models can be derived and combined. A formal specification can thus help in assessing whether a component model constitutes a proper refinement or specialisation of the Acme model.

The remainder of the paper is organised as follows: Section 2 present the Acme ADL, a model that we have chosen as the basis of our study and our experiments because of its relative simplicity and its flexibility and high-level abstraction. Section 3 defines the overall approach that has been followed for specifying and verifying the consistency of Acme’s configurations. Section 4 discusses a simplified Acme metamodel of the ADL model and the possibility of extending it by sets of constraints. We finally see, in Section 5, how to check consistency between the constraints of the model and how to report and validate these constraints in practical applications. Discussion and a conclusion round out the paper.

2 The Acme language

Acme is an architecture description language considered as a pivot and unifying language of all the concepts used by the ADLs while being simple enough to facilitate its analysis.

Acme project started in early 1995 with the aim of providing a common language dedicated to the interchange of architectural descriptions between a variety of architectural tools. But since then the project evolved and many support tools have been developed, in a way that Acme and its library (Acme, 2007) provide a generic, extensible infrastructure for describing, generating and analysing software architecture descriptions.

2.1 Language features

Garlan et al. (1997) describe Acme highlighting its fundamentals features as follows:

- an architectural ontology consisting of seven basic architectural design elements;
- a flexible annotation mechanism supporting association of non-structural information using externally defined sublanguages;
- a type mechanism for abstracting common, reusable architectural idioms and styles; and
- an open semantic framework for reasoning about architectural descriptions.

Acme is built on core ontology of seven types of entities for architectural representation: components, connectors, systems, ports, roles, representations, and representations-maps. These are illustrated in Figure 1.

2.2 Acme properties

The elements outlined above are the key elements for describing an architecture as a hierarchical graph of components and connectors. However, there is more of interest in an architectural description than the structural aspect.

Some auxiliary information has to be added to complete the description with some aspects such as type of data communicated between components, protocols of interaction and more. In order to achieve this, Acme supports annotation of architectural structure with properties. An Armani property is a name, type, value triple. The annotations can be added to any of the seven elements.
2.3 Implementations and model associated tools

The ecosystem of the Acme model offers a certain number of tools and languages for systems engineering. Among these tools, we mainly use the following:

- AcmeStudio (AcmeStudio, 2009), a customisable editing environment and visualisation tool for software architectural designs based on the Acme architectural description language (ADL).

- Armani the integrated constraint checker to verify architectural design rules.

The predicate language Armani is the part of Armani design language used to specify the predicates invariants and design heuristics. The predicate language is based on the predicate of first-order logic, with terms, functions and quantifiers.

3 The proposed approach

This section addresses the problem of specifying architecture configurations in Acme ADL and the definition of integrity constraints on those configurations to ensure consistency of system architectures. By consistency we mean that a system must conform to its architecture in terms of composition and syntactic correctness.

3.1 Checking coherence of configurations

Acme ADL is defined in a formal textual specification. In order to easily determine if a configuration of an Acme described architecture is valid, we want to model well-formed component-based configurations (or architectures).

The first part of the specification captures the underlying core of the Acme model, and then we define constraints based on the textual Acme specification. We call them integrity constraints. This set of constraints has to be necessarily satisfied by a configuration for it to be consistent. Constraints are initially specified in logic formalism, independent from any implementation language. They are then translated in a specification language, Armani (Monroe, 2001) which thanks to its analyser permits to check the non-contradiction between constraints.

We made a primary study of the Acme capabilities and found some research that helped us model our global vision (Figure 2). Based on Plastik (Batista et al., 2005), an integration of Acme/Armani and a reflective component runtime OpenCOM (Coulson et al., 2004).

We focused initially on the architectural level (architectural and instance level of Figure 2). This work has to be followed by the support of runtime and even evolution level, compliant to this initial paper.

4 Specification and extension of Acme model

4.1 A metamodel for Acme

To clarify the specification of the Acme model, initially we introduce a metamodel in Figure 3. This representation allows highlighting the basic concepts of the model: the architectural elements (components, connectors, systems, ports, roles, representations, and rep-maps), the relationships between these elements (link, hierarchy, etc.) and the type system.

To simplify this metamodel for easy handling, we provide an abstraction in Figure 4. This representation is essentially a subset of the metamodel of Figure 3, but is more homogeneous as it considers only architectural elements and relations between these elements.

The purpose of this graph is to maximally simplify the representation of the model to facilitate the validation of configurations while retaining as much information as
possible. The graph includes the main concepts defined in the first metamodel. Elements of the metamodel, however, have been simplified in the graph. This is the case of links (bindings and attachments) that appear as the class hierarchy in the UML diagram and have been reduced to a simple relationship (arc) in the graph. This remains the philosophy of Acme model.

Figure 3  The Acme metamodel (Baudry, 2006)

Figure 4  Simplified Acme-metamodel

4.2  Graph representation of the Acme model

Our approach is greatly inspired by Léger and Cointe (2009). At a given time, an Acme application is modelled by a configuration of architectural elements.

We define therefore an Acme configuration as a directed graph whose vertices (or nodes) and edges are typed. A configuration is called valid (or consistent) if its graph is consistent with the Acme model (the semantics of the conformance relation is defined in the following). This graph is defined as \( S = (E, R) \) where the set of vertices \( E \) represents the architectural elements of the system as defined in the model and the set of arcs \( R \) the relationships between these elements. The architectural elements handled in the configurations are data of the components model that can be represented by the triplet \( M = (M_c, M_r, M_i) \), where:

- \( M_c \) is the set of architectural elements that can appear in the instances of the model.
- \( M_r \) is the set of possible relations between architectural elements.
- \( M_i \) is the set of integrity constraints which must be verified by the instances of the model to be considered valid.

An integrity constraint is a binary predicate \( \pi(E, R) \).

We define then that a configuration (or system) \( S \) conforms to the model \( M \) (\( M \) models \( S \) or vice versa), and note \( M|S = S \) if and only if:

\[
E \in M_c \\
R \in M_r \\
\forall \pi \in M_i, \pi(E, R)
\]

We focus initially specifically for modelling the core of Acme model defined by its three constituents: \( Acme = (Acme_c, Acme_r, Acme_i) \)

We use first-order logic as model’s specification formalism, i.e. to specify the architectural relationships and integrity constraints on configurations. We consider the use of classic logical operators (Boolean) \( \neg, \land, \lor \) and implication \( \Rightarrow \), universal quantifiers \( \forall \) and existential \( \exists \).

The set-operators used are the following: the union \( \cup \), the intersection \( \cap \), difference \( \setminus \) and equality \( = \). The configuration model graph is defined in the following with this formalism.

Data types: We manipulate the primitive data types (Integer, String, Float, Boolean, Sequence, Set, Enum, Record).

We define the following types that are extensions of the primitive types:

- Name that extends String: name of an Element.
- Signature extending String: Signature of interfaces.

Arithmetic operators: The usual arithmetic operators (+, –, \( \times \), \( / \)) are used for types Number and also for comparisons. The equality tests (operator \( = \)) are defined for all data types.

A. The architectural elements and their properties: Architectural elements of Acme model are the vertices of the configuration’s graph and are four: components, connectors, ports and roles. Each vertex of the graph can be associated with a number of properties. These primitive data characterise the represented architectural elements:

- The components: Component = Name
- The connectors: Connector = Name
- The Roles: Role = Name \( \times \) Type (client/Server)
- **The ports**: Port = Name × Signature × Type (Required/Provided)

We have all the architectural elements of Acme model:

\[ \text{Acme}_e = \text{Component} \cup \text{Connector} \cup \text{Role} \cup \text{Port} \]

Any instance of architectural element has a unique name to be distinguishable at system level.

**B. The architectural relations**: Relations between the architectural elements in the model Acme are binary relations \( xRy \) where \( (x,y) \in E^2 \) and correspond to the arcs in the configuration graph in Figure 5. We consider the following primitive relations in Acme.

- The relationship hasPort set to Component \( \times \) Port: determines if a component has a given port (internal or external).
- The relationship hasRole set to connector \( \times \) Role: determines if a connector has a given role.
- The relationship hasChild set to Element \( \times \) Element: determines if an element is direct subcomponent of another element. hasPort and hasRole are specialisations of this relation.
- The relationship hasBinding set to Component \( \times \) Component: determines if a Component is linked to another Component. This relation is not reflexive because a port cannot be linked to itself.
- The relationship hasAttachment set to Port \( \times \) Role: determines if a port is directly linked to a connector’s role. It is not transitive because it considers only the direct connections.

We have all primitive relationships defined in the model:

\[ \text{Acme}_r = \text{hasRole} \cup \text{hasPort} \cup \text{hasChild} \cup \text{hasBinding} \cup \text{hasAttachment} \]

Figure 5 Representation of architectural elements and relations in Acme

4.3 Model specification by integrity constraints

In addition to the definition of architectural elements, their properties and architectural relations, we use integrity constraints to complete the model specification. The constraints used in this specification are structural and behavioural invariants. We can distinguish the architectural constraints on relationships and constraints on the values of properties of architectural elements.

We focus initially on the Acme model’s constraints. These integrity constraints are binary predicates rated \( \pi_1 (E,R) \) ensuring the syntactic and composition conformance of a configuration. These are generic invariants that specify the modelling of an Acme system. One difficulty is not to under-constrain the model to avoid inconsistent configurations (false positives), or over-constrain preventing consistent configurations to be validated (false negatives). Furthermore, constraints may not be contradictory to each other and to the extent possible, not be redundant.

A first set of constraints relates cardinality of the relationship between components and ports and between components and properties

**Uniqueness belonging of port**: A port belongs to a single component.

\[ \text{uniquePortOwner}(E,R) = \forall P \in E, \exists ! C \in E, \]

\[ \text{hasPort}(C,P) \]

**Uniqueness belonging of a role**: A role belongs to a single connector.

\[ \text{uniqueRoleOwner}(E,R) = \forall R \in E, \exists ! Cn \in E, \]

\[ \text{hasRole}(Cn,R) \]

Constraints concerning the uniqueness of architectural elements names

**Uniqueness of names of ports with the same visibility**: A component can have at more one port with a given name.

\[ \text{uniquePortName}(E,R) = \forall (C,P,P) \in E^3, \]

\[ (\text{hasPort}(C,P) \land \text{hasPort}(C,P)) \land \text{name}(P) = \text{name}(P_1) \Rightarrow E = P_1 \]

**Uniqueness of names of roles**: A connector can have at more one role with a given name.

\[ \text{uniqueRoleName}(E,R) = \forall (Cn,R_1,R_2) \in E^3, \]

\[ (\text{hasRole}(C,R_1) \land \text{hasRole}(C,R_2)) \land \text{name}(R_1) = \text{name}(R_2) \Rightarrow R_1 = R_2 \]

**Uniqueness of property names**: an element (component, connector, property role) can have at more one property with a given name.

\[ \text{uniquePropName}(E,R) = \forall (e,p_1,p_2) \in E^3, \]

\[ (\text{hasProperty}(e,p_1) \land \text{hasProperty}(e,p_2)) \land \text{name}(p_1) = \text{name}(p_2) \Rightarrow p_1 = p_2 \]
Compatibility of related ports: A link between two ports is valid if and only if the two ports are of the same type. Let \( \leq \), the relationship of subtyping of ports, then:

\[
\begin{align*}
\text{bindingType}(E,R) &= \forall (p_i, p_j) \in E', \\
\text{hasBinding}(p_i, p_j) \Rightarrow \text{signature}(p_i) \\
&\leq \text{signature}(p_j)
\end{align*}
\]

All required ports must be linked:

\[
\begin{align*}
\text{mandatoryBindings}(E,R) &= \forall (C, P) \in E', \\
\text{hasPort}(C, P) \land \text{client}(P) \\
&\Rightarrow \exists P' \in E, \text{hasBinding}(P, P')
\end{align*}
\]

All roles must be linked: All roles are attached.

\[
\begin{align*}
\text{mandatoryAttachment}(E,R) &= \forall (Cn, R) \in E', \\
\text{hasRole}(Cn, R) \Rightarrow \exists P \in E, \text{hasAttachment}(R, P)
\end{align*}
\]

Local primitive bonds: Primitive connections cannot cross the limits of composites to respect the principle of encapsulation. There are only three possible categories of primitive links:

- Normal connections between ports whose components have at least one direct common parent (Attachment).
- The export connections between an internal client port of a composite and an external server port of one of its direct subcomponents.
- The import connections between an external client port of a component and an internal server port of its relatives (Binding).

\[
\begin{align*}
\text{normalAttachment}(P_i, P_j) &= \text{client}(P_i) \\
&\land \text{external}(P_i) \land \text{server}(P_j) \\
&\land \exists C_c, C_s, C_p \in E, \\
&\text{hasInterface}(C_c, P_i) \land \text{hasInterface}(C_s, P_j) \\
&\land \text{hasChild}(C_p, C_s) \\
\text{exportBinding}(P_i, P_j) &= \text{client}(P_i) \land \text{internal}(P_i) \\
&\land \text{server}(P_j) \land \text{external}(P_j) \land \exists C_c, C_s \in E, \\
&\text{hasInterface}(C_c, P_i) \land \text{hasInterface}(C_s, P_j) \\
&\land \text{hasChild}(C_s, C_c) \\
\text{importBinding}(P_i, P_j) &= \text{client}(P_i) \land \text{external}(P_i) \\
&\land \text{server}(P_j) \land \text{internal}(P_j) \land \exists C_c, C_s \in E, \\
&\text{hasPort}(C_c, P_i) \land \text{hasPort}(C_s, P_j) \land \text{hasChild}(C_c, C_s) \\
\text{localBinding}(P_i, P_j) &= \text{normalAttachment}(P_i, P_j) \\
&\lor \text{exportBinding}(P_i, P_j) \lor \text{importBinding}(P_i, P_j)
\end{align*}
\]

The locality predicate is thus the following:

\[
\text{localBinding}(E,R) = \forall (P_i, P_j) \in E^2
\]

\[
\text{hasBinding}(P_i, P_j) \Rightarrow \text{localBinding}(P_i, P_j)
\]

According to the precedent definitions of integrity constraints, we can finally define the set of Acme, model's constraints.

\[
\text{Acme} = \left\{ \text{uniqueName}, \text{uniquePortOwner}, \text{uniqueRoleOwner}, \text{uniquePropOwner}, \text{bindingType}, \text{mandatoryBindings}, \text{mandatoryAttachment}, \text{localBinding} \right\}
\]

4.4 Armani formalism translation

4.4.1 Architectural elements

We propose the following architectural style named WFCA (Well-Formed Component-based Architecture) in Figure 6.
4.4.2 Constraints

For the purpose of the formalisation, it is necessary to do a mapping to the exact Armani functions, as stated in Monroe (2001):

**Graph connectivity functions**

*attached(conn: Connector, comp: Component): boolean*

Returns True if connector conn is attached to component comp, else False.

*attached(r: Role, p: Port): boolean*

Returns True if role r is attached to port p, else False.

*connected(c1, c2: Component): boolean*

Returns True if component c1 is directly connected to component c2 by at least one connector, else False.

*reachable(c1, c2: Component): boolean*

Reachable(…) is the transitive closure of Connected(…). It returns True if component c2 is reachable from component c1, else False. This function examines only undirected graph connectivity.

**Parental child functions**

*parent(c: Component): System*

Returns the System in which Component c is instantiated or nil if c is not a child of anything.

*parent(c: Connector): System*

Returns the System in which Connector c is instantiated or nil if c is not a child of anything.

*parent(p: Port): Component*

Returns the Component in which Port p is instantiated or nil if p is not a child of anything.

*parent(r: Role): Connector*

Returns the Connector in which Role r is instantiated or nil if r is not a child of anything.

*parent(s: System): Representation*

Returns the Representation in which System s is declared, or nil if s is a top-level System and thus not declared in a representation.

*parent(p: Property): Element*

Returns the Element of which p is a Property, or nil if p is not a property of anything.

*parent(r: Representation): Element*

Returns the Element of which r is a Representation, or nil if r is not a Representation of anything.

A first set of constraints relates cardinality of the relationship between components and ports and between components and properties

Uniqueness belonging of port:

*Rule uniquePortOwner(sys:System): boolean =*

Forall c1 : Component in sys.Components | forall c2 : Component in sys.Components | Forall P:Port in sys.Ports | parent(p1)=c1 and parent(p1)=c2 => c1 = c2

Uniqueness belonging of a role:

*Rule uniqueRoleOwner(sys:System): boolean =*

Forall c1 : Connector in sys.Connectors | exists c2 : Connector in sys.Connectors | forall RRRole in sys.Roles | parent(R)=c1 and parent(R)=c2 => c1 = c2

Constraints concerning the uniqueness of identifiers and architectural elements names

Uniqueness of identifiers:

*Analysis uniqueld(sys:System): boolean =*

Forall el : Element in sys.Elements | Forall e2 : Element in sys.Elements | el.ld = e2.ld => el=e2

Local primitive bonds:

*Rule normalAttachement(sys:System): boolean =*


*Rule exportBinding(sys:System): boolean =*


*Rule importBinding(sys:System): boolean =*

A template for formalising

Component in sys.Components | Parent(Pc)=Cc and Parent(Ps)=Cs and Pc.role="client" and Pc.visibility="external" and ps.role="server" and Ps.visibility="internal" and Connected (Cc,Cs)

Rule localBinding(sys:System) : boolean = normalAttachment(sys:System) or exportBinding(sys:System) or importBinding(sys :System)

5 Test and validation

We have developed our example after studying (ATMExample, n.d.), from a series of complete examples of object-oriented analysis, from the design to the programming.

5.1 Requirements statement for example ATM system

The ‘automated teller machine’ (ATM) system has a set of components:
- A magnetic stripe reader for reading an ATM card,
- A customer console (keyboard and display) for the interactions;
- A slot for printing customer receipts;
- A slot for depositing envelopes;
- A dispenser for cash;
- A dedicated communication link, which the communication with the bank’s computer is done through.

The ATM will service one customer at a time. A customer will be required to insert an ATM card and enter a Personal Identification Number (PIN). The combination will be used by the bank for the validation of each transaction.

When the customer indicates that he/she no longer desires extra transactions, the card will be released by the ATM.

The ATM provided services are:
- **Withdrawal** from any suitable account linked to the card.
- **Deposit** to any account linked to the card, consisting of cash and/or checks in an envelope. The customer will enter the amount of the deposit into the ATM, subject to manual verification when the envelope is removed from the machine by an operator.
- **Balance inquiry** of any account linked to the card.

All the provided transactions have to be approved by the bank after a verification process.

The bank validates the customer PIN, if the PIN is invalid the customer will be required to re-enter the PIN. If the operation fails, the card will be retained by the ATM, until the customer contacts the bank to get it back.

The ATM will provide the customer with a detailed printed receipt for each successful transaction.

Figure 7 represents the ATM class diagram. Only name of each class is shown in the diagram, in order to contain the size. The components are detailed later, since this form of representation is the closest to architectural style.

Figure 7 ATM class diagram

```
CashDispenser | EnvelopeAccepter | NetworkToBank

ATM

CardReader | CustomerConsole | ReceiptPrinter

Transaction

Withdrawal | Deposit | Inquiry
```

5.2 ATM modelling diagram

5.2.1 Component diagram

The component diagram associated with the system is given in Figure 8. The component diagram main purpose is to show the structural relationships between the components of a system. Components expose their functionality through their ports. A port represents a point of contact between the component and its environment. A port may represent a source of requests, a service provided by the component, or represent a source of or destination for data.

Component parts of the ATM:
- CardReader
- CashDispenser
- CustomerConsole
- EnvelopeAccepter
- NetworkToBank
- ReceiptPrinter
**ATM**

The ATM collaborates with CashDispenser, NetworkToBank and CustomerConsole. Starts a new session when card is inserted by customer and provides access to component parts for transactions.

**CardReader**
- It collaborates with ATM and card.
- Tells ATM when card is inserted, and reads information from card.

**CashDispenser**
- It keeps track of cash on hand, starting with initial amount.
- Reports whether enough cash is available, and dispense cash.

**CustomerConsole**
- It displays a message.
- Displays a prompt, accepts a PIN from keyboard.
- Displays a prompt and menu, accepts a choice from keyboard.
- Displays a prompt; accepts a dollar amount from keyboard.

**EnvelopeAcceptor**
It accepts envelope from customer; reports if timed out or cancelled.

**NetworkToBank**
It sends message to bank and waits for response.

**ReceiptPrinter**
It prints receipt.

---

**Components corresponding to the various operations:**
- Transaction
- Withdrawal
- Deposit
- Inquiry

**The abstract transaction**
- It allows customer to choose a type of transaction.
- Performs transaction use case.
- Performs invalid PIN extension.

**Withdrawal**
It performs operations peculiar to withdrawal transaction use case.

---

**Deposit**
It performs operations peculiar to deposit transaction use case.

**Inquiry**
It performs operations peculiar to inquiry transaction use case.

---

**Figure 8** ATM component diagram

---

5.2.2 **Interfaces of the ATM system**

In this section, we present signatures of interfaces’ services belonging to components. The ATM component has the same interfaces as its composites.

**Interface IEnvelopeAcceptor**
The role of this interface is to accept the customer envelope. It offers two services:
- `acceptEnvelope()`
  If the deposit transaction is approved, the machine accepts an envelope from the customer containing cash and/or checks.
- `cancelOperation()`
  After asking the customer to insert the envelope a countdown is started. If the customer fails to insert the envelope in a reasonable period of time, the transaction is automatically cancelled.
Interface ICardReader

This interface handles customer’s card and offers two services:

- readCard()

  When a customer inserts an ATM card into the card reader slot of the machine, the ATM pulls the card into the machine and reads it.

- ejectCard()

  When the customer is through performing transactions, the card is ejected from the machine.

Interface ReceiptPrinter

This interface just prints receipt from different possible transactions, offering a single service:

- printReceipt()

  Print the corresponding receipt.

Interface ICashDispenser

The interface provides one service:

- dispenseCash()

  This operation dispenses cash to customer.

Interface INetworktoBank

- sendMessage()

  This service sends message to bank and wait for response.

Interface ITransaction

- makeTransactions()

  The operation validates transactions’ operations, like updating account amount.

Interface ICustomerConsole

The interface provides interaction with customer, offering the following services:

- display()

  Display a message to the customer such as prompting the customer to enter PIN or prompting the customer to enter amount.

- readPIN()

  This operation reads customer’s pin.

- readMenuChoice()

  The operation lets the customer choose from a menu of possible types of transaction.

- readAmount()

  When the customer chooses a transaction type from a menu of options, the customer will be asked to furnish appropriate details including money amount.
5.3 Acme/Armani formalisation of an ATM

In this section we propose an Acme/Armani formalisation of the ATM model described in Section 5.2. To do this, we describe the data types and signatures of the services offered by the application by judiciously using the Typing and family construction offered by Acme.

Component diagram of the system is formalised in Acme using the construction system. Finally, the coherence rules are established using the invariant constraint supported by Armani.

The assembly of UML2.0 components of the application is modelled by a system called ATM which derives from the WFCA family. To achieve this, we have applied the following rules defined by Kminech and Aniorté (2010), which are quite applicable to our example:

- **R1**: A UML component is translated by an Acme component.
  - **R1.1**: An interface attached to a UML2.0 component is translated by an Acme port.
  - **R1.2**: A service declared within a UML2.0 interface is translated by a property typed Acme, attached to the port that formalises this interface. The type of property represents the signature of the service.
- **R2**: A UML2.0 assembly connector connecting an available interface and a required interface is modelled by an Acme binary connector with two roles.
- **R3**: The properties attached to a role must be the same as those of the corresponding port.

5.4 Verification

In this section, we propose coherence rules, in addition to the original ones defined in the WFCA style. These rules are modelled using the invariant concept of Acme. They concern verification of the attachments (and binding) of the ATM configuration.

Rule: An attached role and port have the same number of properties, see Figure 14.

5.5 Conclusion

In this section, we have modelled an ATM in Acme/Armani. The aim of this model is to verify the syntactic verification, which is performed by AcmeStudio. To do so we describe the signatures of the provided services of the system. Using the family definition provided by Acme. In case of error detection, the editor displays a message as showed in Figure 15.
6 Related work

Kmimech et al. (2009a, 2009b) present an MDE (Model-Driven Engineering) checking approach based on ADLs. To achieve their goal, they describe the principles concepts of the UML2 component meta-model (level M2) using Acme architectural style. An architecture based on those components (level M1) is described with the Acme system notation. The M1 level conforms to level M2 if it verifies consistency rules defined at M2 level in addition to the rules defined at M1 level. They exploit the AcmeStudio built-in tool to verify invariants to check the structural and non-functional properties of the component model.

Our work focuses more on the Acme core model, regarding to Acme’s capabilities, and since our goal is to provide a solid base for handling application conformance from design to configuration. Acme permits us to express both the architecture and the system levels.

Maraoui et al. (2010) propose a general framework to ensure a safe design and execution of software architectures applied to web services composition by formalising this composition mechanism, and ensuring reliability. To do this they described the Web services composition Metamodel (M2 level) using Acme style architecture. The verification of the structural and non-functional properties of the composition models is performed by Armani to check invariants of an Acme model to ensure the system compliance (M1 level).

Even if the authors restrict their work to web services architectures, the use of the MDE principle remains the same, but we aim at a larger scale, focusing on component-based architectures, and this whatever the application domain.

David et al. (2009) propose a definition of consistency for configurations and reconfigurations in Fractal component architectures by defining a set of invariants and constraints expressed at two levels (style and instance). However, they enhance this mechanism by a transaction model ensuring the maintenance of system consistency. Nevertheless this can’t be achieved without a precise definition of system consistency. Léger et al. (2010) focus on two languages Fpath and Fscript, specific to the Fractal model to make the process of initially defining a consistent model and its integrity constraints, then maintaining a certain level of reliability during system evolution.

Although Fractal is a rich and well-known ADL especially in the dynamic reconfiguration field, we believe that Acme can achieve a lot in this domain. The two languages are quite different in terms of core elements, which make the operations of either introspections or intercessions different. The property concept in Acme gives it a slight advantage in terms of abstraction level and domain handling.

Caracciolo et al. (2010) present an approach that aims at optimising the cost of architectural conformance checking. With their approach they reduce the effort required to describe and maintain rules, and effectively test system conformance. They propose a common DSL (Domain-Specific Language) that unifies the functionalities provided by existing tools by using a declarative language to write rules that are simple enough to be understood by untrained stakeholders and, at the same time, can be interpreted as a rigorous specification for checking architecture conformance.

Using Acme and its set of tools allows us to meet a part of the problem posed in this work since the invariant constraints are based on first-order predicate logic, and the integrated Armani parser checks them automatically into the AcmeStudio environment.

7 Conclusion

In this work, we have set out a well-defined Acme template for specifying software architectures, working on the mastery of the reliability of software architectures and improving consistency between design and system.

This is done by keeping the relation between architecture’s design at initial state and configuration. Our approach is based on the definition of a set of integrity constraints based on the first-order predicate logic. Our template guarantees a reliable software design and a trusted
configuration. This can only be achieved by proposing different configuration rules to permit incremental and atomic system construction that take in account scalability and variability aspects of software architecture. This work is still open to the management of dynamic evolving systems.

References


Kmimche, M. and Aniorté, P. (2010) Vérification d’assemblages de composants logiciels, Université de Pau et des Pays de l’Adour, Pau. [In French]


Le Goaer, O. (2009) Styles d’évolution dans les architectures logicielles, Université de Nantes, Ecole Centrale de Nantes. [In French]


