Modelling and Verifying Smell-Free Architectures with the Archery Language

Alejandro Sanchez[1,2], Luis S. Barbosa[2], and Alexandre Madeira[2(✉)]

[1] Departamento de Informática, Universidad Nacional de San Luis, Ejército de los Andes 950, D5700HHW San Luis, Argentina
asanchez@unsl.edu.ar

[2] HASLab INESC TEC and Universidade Do Minho, Campus de Gualtar, 4710-057 Braga, Portugal
{asanchez,lsb,madeira}@di.uminho.pt

Abstract. Architectural (bad) smells are design decisions found in software architectures that degrade the ability of systems to evolve. This paper presents an approach to verify that a software architecture is smell-free using the Archery architectural description language. The language provides a core for modelling software architectures and an extension for specifying constraints. The approach consists in precisely specifying architectural smells as constraints, and then verifying that software architectures do not satisfy any of them. The constraint language is based on a propositional modal logic with recursion that includes: a converse operator for relations among architectural concepts, graded modalities for describing the cardinality in such relations, and nominals referencing architectural elements. Four architectural smells illustrate the approach.

1 Introduction

Software systems evolve to cope with contextual change. This change compromises the value a system delivers as it might come, for instance, from the market or legislation in which the system is embedded. The principal design decisions governing a system, *i.e.*, the *software architecture* [14], play a fundamental role in its ability to evolve and address change.

Architectural (bad) smells are recurrent architectural decisions that have a negative impact on the ability of a system to evolve [5]. A catalogue is presented in [6], where they are characterized in terms of the basic building blocks that *architectural description languages* (ADL) offer, *i.e.*, components, connectors, interfaces, and configurations. These design decisions may not constitute an error or fault, but violate engineering principles such as isolation of change and separation of concerns. They affect the ability to evolve since they difficult understanding, testing, maintaining, extending and reusing parts of a system.

In the context of *open source software* (OSS), architectural smells acquire further relevance. This is because one of the most important success factors is the voluntary contribution of OSS community members [1]. Thus, the easier the system is to understand, test, maintain, extend and reuse, the greater the chances

C. Canal and A. Idani (Eds.): SEFM 2014 Workshops, LNCS 8938, pp. 147–163, 2015.
DOI: 10.1007/978-3-319-15201-1_10

of involving volunteers. Formal approaches enabling the automatic verification of smell-freeness of architectures will have a positive impact in both the quality of OSS projects and in the health of the involved community [2].

The work reported in this paper aims toward such end. The approach consists in using the Archery language [8,9], an ADL with formal semantics, to verify constraints specifying the absence of architectural smells in software architectures. It does not aim at replacing existing practices in OSS communities, but to complement them, as suggested by the proposal discussed in [2]. Archery is organized as a basic language, named Archery-Core, and extensions built on top of it. Archery-Core allows modelling the structure and behaviour of software architectures in terms of architectural patterns, and the extensions are for specifying reconfiguration scripts and constraints.

Archery-Constraint is the extension for specifying constraints upon either structure, behaviour or reconfiguration processes of architectures. The specification language is based on a propositional modal logic. As a consequence, constraints become formulæ of a modal logic, interpreted over Kripke structures obtained from Archery's specifications (see [10] for reconfiguration and [11] for structure). Since the proposed approach focuses on structural constraints, modalities allow inspecting the Kripke structure obtained from an architecture, by regarding the configuration constituents and their relationships as the Kripke structure's worlds and relationships, respectively.

The underlying logic is a fully enriched μ-calculus [3]. It includes fixed points, a converse operator, two graded modalities and hybrid features. Fixed points are for specifying recursive formulæ, and thus liveness and safety conditions. The converse operator allows exploring the converse of relations, and graded modalities allow describing their cardinalities. Hybrid features consist of a mechanism to explicitly refer to specific worlds through nominals, elementary propositions, each of which is only true at the world it identifies, and a reference operator which asserts that a formula is satisfied at the world named by a specific nominal. These features make possible, for instance, to express the equality between two worlds, to denote that a world is accessible through a relation from another world, or to assert the irreflexivity of a relation. Moreover, they make possible to describe acyclic structures when included in recursive constraints [11].

The approach can be used upon recovery techniques are applied to obtain an Archery model. In fact, techniques were applied in [12] to recover an Archery model for an existing software system, and subsequent model-based analysis and modifications were carried out. It is worth noting that the unrestricted access to source code renders OSS systems a natural target for the presented approach.

Architectural smells described in [6], and architectures of actual software systems illustrate the approach. The architectures were either documented during development, or recovered from source code, and are described in references also available in [6]. Observe that one of the example architectures was recovered from Linux [4], an open source operating systems widely adopted.

The obtained constraints correspond to decidable fragments of the underlying logic. The fully enriched μ-calculus is known to be not decidable, however, the

fragments obtained by omitting one of either the converse operator, the graded modality operators, or the hybrid features, is [3]. None of the constraints that characterize the smells requires recursion, and three of them exclude either the converse operator, the graded modality operators, or the hybrid features.

The contribution of the paper is two fold. First, the constraint language presented in [11] is extended by including graded modalities. Second, the extended language is applied to precisely model architectural bad smells, which enables formally verifying the absence of these violations to design principles.

The rest of the paper is structured as follows: Sect. 2 briefly describes the Archery language; Sect. 3 characterizes the smells as structural constraints; Sect. 4 describes the fully enriched μ-calculus, the translation of structural constraints to it, and illustrates how a constraint is verified; Sect. 5 summarizes results and describes future work.

2 The Archery Language

This section describes Archery-Core in a brief and partial way (detailed descriptions can be found in [8,9]), and extends the structural part of Archery-Constraint presented in [11]. The language is illustrated with an architectural pattern inspired in the Java Messaging Service (JMS). It prescribes three architectural elements: *queues*, where messages are kept in a specific order; *producers*, that send messages to the queue; and *consumers*, that receive messages from the queue. In the example pattern, a consumer provides one of three possible services, depending on the received message.

2.1 Archery-core: Modelling Structure

An Archery-Core specification comprises one or more (architectural) patterns, a variable that references the main architecture, and global data specifications (not part of the examples in this paper). A *pattern* defines one or more (architectural) elements (connectors and components), such as the `JMS` pattern and the `Queue`, `Producer` and `Consumer` elements shown in Listing 1.

```
1  pattern JMS()
2  element Queue()  interface in rcvMsg; out dlvr;
3  element Producer()  interface in start; out sndMsg;
4  element Consumer()  interface in onMsg; out func;
5   act funcA,funcB,funcC;
6  end
7  jms:JMS = architecture JMS()
8   instances
9    q:Queue();
10   p:Producer=Producer(); c:Consumer=Consumer();
11  attachments
12   from p.sndMsg to q.rcvMsg;
13   from q.dlvr to c.onMsg;
```

```
14  interface p.start as produce;  c.func as consume;
15 end
```

Listing 1. JMS Pattern and architecture

Each *element* includes an *interface* that contains one or more ports. A *port* is defined by a polarity, either `in` or `out` and a name. For instance, the interface of `Queue` defines two ports in line 2. An element can optionally include a set of actions, and a set of process descriptions expressed in a subset of the mCRL2 process algebra. An *action* represents an event that is not a port activation, *e.g.*, see line 5. Process descriptions are not considered in the sequel.

A variable (see line 7) has an identifier and a type that must match an element or pattern name. Allowed values are instances of a type (element or pattern), that do not necessarily need to match the variable's own type.

An architecture describes the configuration a set of instances adopt. It contains a token that must match a pattern name, a set of variables, an optional set of attachments, and an optional interface. The type of each variable in the set is limited to an element in the pattern the architecture is instance of, such as in line 10. Each attachment includes port references to an output and an input port. A port reference is an ordered pair of identifiers: the first one matching a variable identifier, and the second matching a port of the variable's instance. Then, an attachment indicates which output port communicates with which input port – see e.g. `p.sndMsg` with `q.rcvMsg` in line 13. The architecture interface is a set of one or more port renamings. Each port renaming contains a port reference and a token with the external name of the port. An example interface is shown in line 14. Ports not included in this set are not visible from the outside.

2.2 Archery-constraint: Describing Structure

Structural constraints are verified over Kripke models obtained from Archery-Core specifications. Each model includes a set W of worlds and a family R of binary relations among them, with Mod a set of relation labels. The meta-model of Archery's architectures is shown in Fig. 1. The worlds are the constituents: instances, ports, actions, variables, port references, attachments, names, and renamings. The relationships among constituents conform the family R of relations. The labels of relationships in Fig. 1 become the modality symbols $m \in Mod$. For convenience, modality symbols `attd` and `evt` are included. The former names the relationship that relates two worlds representing variables connected through an attachment. It is obtained as $\mathcal{R}[vref]^\circ \circ \mathcal{R}[strt]^\circ \circ \mathcal{R}[end] \circ \mathcal{R}[vref]$, where $\mathcal{R}^\circ$ denotes the converse of a relation. The latter is obtained as $\mathcal{R}[prt] \cup \mathcal{R}[act]$.

Propositions test if a specific condition is present at a (world) w. They are classified in: (*a*) *Naming* propositions exist for each action and port name, and hold when evaluated at a world representing the corresponding action or port. (*b*) *Meta-type* propositions hold when w belongs to a specific participant set, *e.g.*, `PatternInstance`. (*c*) *Emptiness* is checked by a single proposition, namely `Empty`, which holds when w is a variable with no associated instance. (*d*) *Type*

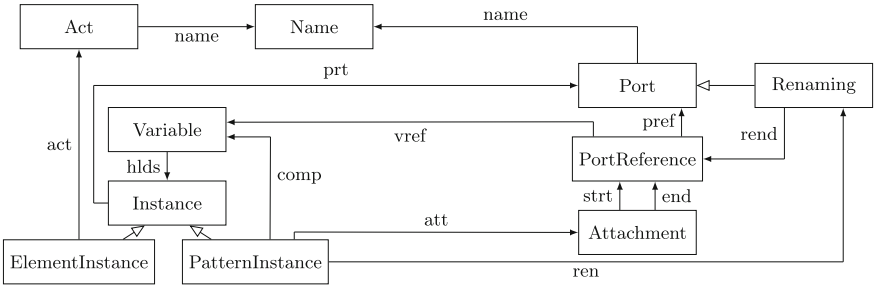

Fig. 1. Relations and roles in spatial specifications

propositions depend on the pattern definition. They test if w is an instance or a variable of a type. For example, the JMS pattern generates four proposition symbols: `JMS`, `Queue`, `Producer` and `Consumer`.

Each variable in a specification defines a nominal in the set Nom_{var}. In addition, depending on the variable's type, they are also included in a subset $Nom_{var:TYPEID}$. Then, each nominal holds exactly at the world that represents the corresponding variable.

Structural constraints are associated to a pattern or to a pattern instance. They allow precisely describing design decisions that characterize architectural patterns [11], and the absence of smells, as it is shown in this paper.

Pat	::=	**pattern** *THeader* Elem+ *SConsts*? **end**
PatInst	::=	**architecture** *IHeader ABody SConsts*? **end**
SConsts	::=	**structural constraints** *SConst*+
SConst	::=	**const** ID *Q*? *F*; *Rec** **end**
Q	::=	(**all** | **exists**) ID (**:** TYPEID)? **.**
Rec	::=	(**finite** | **infinite**) ID = *F*;
F	::=	**True** | **False** | PROP | **not** *F* | *F* **or** *F* | *F* **and** *F*
	|	*F* **implies** *F* | *F* **iff** *F*
	|	**[***M***]***F* | <*M*>*F* | **[(Nat,)**?*M***]***F* | <(**Nat,**)?*M*>*F*
	|	**A** *F* | **E** *F* | ID | NOM | **at** NOM *F*
M	::=	MOD | MOD-

Fig. 2. Grammar of structural constraints

A well-formed constraint is either a propositional formula, a modal formula, a converse formula, a graded modality formula, a recursive formula, or a hybrid formula (see grammar in Fig. 2). In a modal formula, a $\langle M \rangle F$ indicates that there exists a relationship M (named by expression M) between the present world and another world satisfying (formula) F, whereas a $[M]F$ indicates that any relationship M leads to a world satisfying F. An M non-terminal describes either a modal symbol `Mod`, that names a relation $\mathcal{R}[Mod]$ in the Kripke model,

or the converse $\mathcal{R}[Mod]^{\circ}$ indicated with `Mod-`. Graded modality formuæ, `<n,M>`*F* and `[n,M]`*F*, describe a world where *F* holds in at least `n+1` `M`-related worlds, and a world where *F* holds in all but at most `n` `M`-related worlds, respectively. In recursive formulæ, an `ID` designates a formula, and it is indicated if the recursion is expected to be finite or infinite. Hybrid formulæ are built of a nominal *Nom*, that is satisfied if the current world is the unique world referenced by such *Nom*, and of a reference operator **at** *Nom F*, which is satisfied if at the world named by *Nom*, *F* is. Global modality formulæ **E***F* and **A***F* are also included in the logic, as they allow defining duals for the reference operator. They are as $\langle M \rangle F$ and $[M]F$ but with $W \times W$ as the underlying relation.

The quantifiers **all** and **exists** can only occur in the beginning of a constraint and have as domain the variables of the configuration. The meaning of an **all** x:TYPEID *F* is the conjunction of formulæ **at** *i F*, for each $i \in Nom_{var:TYPEID}$. The meaning of an **exists** x:TYPEID *F*, is a disjunction of formulæ **at** *i F*, for each $i \in Nom_{var:TYPEID}$.

3 Architectural Smells

In this section, the ARCHERY language is used to characterize the architectural smells in [6]: *connector envy*, *scattered parasitic functionality*, *ambiguous interfaces*, and *extraneous adjacent connector*. The smells are illustrated using the same examples used in [6], which are specified and then verified using ARCHERY. The examples do not aim at including an exact model of the software architecture, but to cover the fragment which is relevant to the smell.

3.1 Connector Envy

Components with connector envy assume responsibilities that a connector typically assumes. These responsibilities supporting interaction are classified as either concerning *communication*, *coordination*, *conversion*, or *facilitation* [7]. Communication and coordination services carry out the transfer of data and control, respectively. Conversion services address mismatches between required and provided interactions. Facilitation services cover streamlining and optimization needs in interactions.

The filesystem daemon of the Grid Datafarm [13] is an instance of connector envy [6]. The Grid Datafarm is a framework for petabyte scale data-intensive computing. It offers a filesystem distributed over the nodes of a PC cluster, where the operations in each node are facilitated by a daemon. The smell emerges as each daemon incorporates, besides its domain specific functionality, coordination behaviour that relies in a private remote procedure call (RPC) mechanism to interact with other daemons.

Listing 2 shows the specification of the pattern fragment and an instance. It only includes the daemon element `GFSD`, which has ports to coordinate work with peers through RPC (`sndRpcCoord` and `rcvRpcCoord`), and to allow accessing its functionality (`sndResFun` and `rcvReqFun`). The architecture consists of two instances of the daemon connected through the ports for RPC coordination.

```
pattern GDatafarm()
element GFSD()
 interface
   in rcvReqFun, rcvRpcCoord; out sndResFun, sndRpcCoord;
end
df:GDatafarm = architecture GDatafarm()
 instances d1:GFSD=GFSD(); d2:GFSD=GFSD();
 attachments
  from d1.sndRpcCoord to d2.rcvRpcCoord;
  from d2.sndRpcCoord to d1.rcvRpcCoord;
end
```

Listing 2. Fragment of Grid Datafarm pattern and example architecture

The constraint that verifies that an architecture does not suffer of connector envy is shown in Listing 3. It is divided in two parts, one that is generic and another that is specific to the pattern. The generic part comprises lines 1 to 4. It states that if a world represents an element instance, then it is not possible to access to a world that represent domain functionality and to a world that represent interaction (communication, coordination, conversion, or facilitation) from it. The specific part, line 4–8, establishes the worlds that represent functionality and interaction by indicating the propositions that hold in such worlds.

```
const ConnEnvy
 A (ElementInstance implies
   not (<evt> Function and <evt> Interaction));
 finite Interaction = Comm or Coord or Conv or Fac;
 finite Function = rcvReqFun or sndResFun;
 finite Comm = False;
 finite Coord = rcvRpcCoord or sndRpcCoord
 finite Conv = False;  finite Fac = False;
end
```

Listing 3. Specification of connector envy for Grid Datafarm

3.2 Scattered Parasitic Functionality

The scattered parasitic functionality is found when a set of architectural elements share a concern while at the same time, some of them individually address an additional unrelated concern. Thus, the principle of separation of concerns is violated in two different ways: by scattering a concern among a set of elements, and by making a single element responsible of two concerns.

This smell is found in the Linux kernel architecture [6] as recovered in [4]. The PROC file system contains status information about the kernel, including its executing processes. However, it relies on other kernel subsystems to report their own status. As a result, the Process Scheduler and the Network Interface subsystems depend on the PROC file system.

Listing 4 shows an ARCHERY's specification for a fragment of the recovered architecture of the Linux kernel. The pattern includes a ProcFS element that receives status reports in port rcvStatus. It also includes the elements NetInterface and ProcScheduler that share a port sndStatus and an action statusChk, as their instances send a status report to an instance of ProfFS. These two elements also have unshared functionality, modelled by other actions. The architecture contains an instance of each element, and connects the other two with the ProcFS instance.

```
pattern Kernel()
element ProcFS()  interface in rcvStatus;
element NetInterface()  interface out sndStatus;
 act connect, access, statusChk;
element ProcScheduler()  interface out sndStatus;
 act schedule, statusChk;
end
k:Kernel = architecture Kernel()
 instances
  prc:ProcFS=ProcFS(); sch:ProcScheduler=ProcScheduler();
  net:NetInterface=NetInterface();
 attachments
  from sch.sndStatus to prc.rcvStatus;
  from net.sndStatus to prc.rcvStatus;
end
```

Listing 4. Fragment of Linux kernel architecture

The constraint specifying the absence of the scattered parasitic functionality is shown in Listing 5. It requires that for each instance in an architecture, referenced by a nominal x, if there is a name that corresponds to an action of (the instance referenced by) x, then, it is not possible to find two actions with that name that belong to instances in the same architecture as x, which also have at least another action. The meaning of some of the expressions is as follows: <name-><act->x describes a name that corresponds to an action of x; <name-><2,act-> holds in a name shared by at least two actions; <comp-><comp>x holds in an instance placed in the same architecture as x; and <2,act>True holds in an instance with at least two actions.

```
const ScatteredParasiticFunc
all x. A ((Name and <name-><act->x) implies not
  (<name-><2,act->(<comp-><comp>x and <2,act>True));
end
```

Listing 5. Specification of scattered parasitic functionality

3.3 Ambiguous Interfaces

An ambiguous interface offers a single entry point into an architectural element that offers multiple services. Instance of this smell are found in the JMS pattern, as reported in [6]. The example pattern is described in Sect. 2.

Listing 1 shows the specification that corresponds to a fragment of the JMS pattern and a software architecture. The smell is present in consumer instances that receive messages in port `onMsg`, but can perform any of three functionalities represented by actions `FuncA`, `FuncB` and `FuncC`.

The absence of cases of this smell is specified for the JMS example in Listing 6. The constraint detects the cases in which there is a single entry point, but multiple services are offered. The constraint holds if whenever there is an element instance, it is not the case that it has a number of ports less or equal to two, with one having inward direction, and it also has at least two actions that correspond to specific functionality. Note that the expressions `[2,prt]False` holds at worlds that represent instances that have at most two ports.

```
1 const AmbInt
2  A (ElementInstance implies not
3   ([2,prt]False and <prt>In and <2,act>Function);
4  finite Function = FuncA or FuncB or FuncC
5 end
```

Listing 6. Specification of ambiguous interfaces for JMS architectures

3.4 Extraneous Adjacent Connector

This smell occurs when two architectural elements interact through two different connector types. The presence of an extra connector type may cause a cancellation of the benefits that each of them offers individually.

The MIDAS System shows an instance of extraneous adjacent connector as reported in [6]. Communication in the system is mainly supported by event-based connectors, which are used by all high-level services. An exception is the *service discovery engine* that accesses the *service registry* using procedure calls. Then, the two components interact through two different connector types, which constitutes an instance of the extraneous adjacent connector.

The specification in Listing 7 characterizes a fragment of the pattern of the MIDAS system, and an architecture where the smell is found. It includes four elements: two connector types, and two component types. The former represent the event-based connector type `Channel` and the procedure call connector type `PC`. The component types are `ServiceDiscovery` and `ServiceRegistry`. The architecture includes an instance of each of the elements, and connects the two components using two connectors of different types. This configuration constitutes an instance of the extraneous adjacent connector.

```
pattern MIDAS()
element Channel()  interface in rcvEvnt; out sndNtf;
element PC()  interface in rcvPcComm; out sndPcComm;
element ServiceDiscovery()
 interface in rcvNtf; out sndEvnt, sndPc;
element ServiceRegistry()
 interface in rcvNtf, rcvPc; out sndEvnt;
end
m:MIDAS = architecture MIDAS()
 instances c:Channel=Channel(); pc:PC=PC()
 sd:ServiceDiscovery = ServiceDiscovery();
 sr:ServiceRegistry = ServiceRegistry();
 attachments
  from sd.sndEvnt to c.rcvEvnt;
  from sr.sndEvnt to c.rcvEvnt;
  from c.sndNtf to sd.rcvNtf;  from c.sndNtf to c.rcvNtf;
  from sd.sndPc to pc.rcvPcComm;
  from pc.sndPcComm to sr.rcvPc;
end
```

Listing 7. Fragment of MIDAS Pattern and architecture

The constraint in Listing 8 specifies the absence of a case of extraneous adjacent connector. The constraint holds if whenever there is an element instance, it is not attached to connectors of different type. It is formulated in a very specific way, as it only considers the connector types of the pattern. If the pattern includes different connector types, the conjunction of the constraint needs to be reformulated, to consider all different pairs.

```
const ExtAdjConn
 A (ElementInstance implies not
  (<attd>PC and <attd>Channel));
end
```

Listing 8. Specification of extraneous adjacent connector for MIDAS

4 Verifying Architectural Constraints

This section describes the syntax and semantics of the fully enriched μ-calculus [3], provides a translation that takes a constraint and yields a formula in such logic, establishes the fragment of the logic used to characterize each architectural smell, and illustrates the logic with a manual verification of the formula that corresponds to the absence of the ambiguous interface smell on the model for the JMS example architecture.

The syntax of the fully enriched μ-calculus is shown in Definition 1.

Definition 1. Let *Prop* be a set of *propositional symbols*, *Mod* a set of *atomic modal symbols*, *XVar* a set of *states variables*, and *Nom* a set of *nominals*. A modal symbol β is either

(a) an atomic modal symbol α, or
(b) the converse of an atomic modal symbol (denoted as) α°.

Then, the set *SForm* of well-formed *state formulæ* of the *fully enriched* μ*-calculus* is the smallest set such that a state formula is either

(c) the top constant $\top$,
(d) a proposition p,
(e) a negation $\neg\varphi$,
(f) a conjunction $\varphi \wedge \psi$,
(g) a possibly operator $\langle\beta\rangle\varphi$,
(h) a state variable X,
(i) a maximal fixed point formula $\nu X.\varphi$, with every free X in φ occurring positively, *i.e.*, within the scope of an even number of negations,
(j) an at least graded modality $\langle n, \beta\rangle\varphi$ with $n \in \mathbb{N}$,
(k) a global possibly operator $\mathbf{E}\varphi$,
(l) a nominal i,
(m) a formula satisfaction operator $@_i\varphi$

where $p \in Prop$, $\{\varphi, \psi\} \subseteq SForm$, $X \in XVar$, and $i \in Nom$. □

Derived constants and operators are obtained as follows:

$$\begin{aligned} \bot &= \neg\top & \varphi \vee \psi &= \neg(\neg\varphi \wedge \neg\psi) \\ \varphi \rightarrow \psi &= \neg\varphi \vee \psi & \varphi \leftrightarrow \psi &= \varphi \rightarrow \psi \wedge \psi \rightarrow \varphi \\ [\beta]\varphi &= \neg\langle\beta\rangle\neg\varphi & \neg\langle n, \beta\rangle\varphi &= [n, \beta]\neg\varphi \\ \mathbf{A}\varphi &= \neg\mathbf{E}\neg\varphi & \mu X.\varphi &= \neg\nu X.\neg\varphi[X/\neg X], \end{aligned}$$

where $\varphi[X/\neg X]$ denotes a formula φ with all occurrences of X replaced with occurrences of $\neg X$.

Table 1. Fragments of the fully enriched μ-calculus

Logic	Clauses	Constraint (listing)
Fully enriched μ-calculus	(a)–(m)	Scattered parasitic func. (5)
Full graded μ-calculus	(a)–(j)	–
Full hybrid μ-calculus	(a)–(i), (k)–(m)	–
Hybrid graded μ-calculus	(a), (c)–(m)	Ambiguous interfaces (6)
Graded μ-calculus	(a), (c)–(j)	–
Hybrid μ-calculus	(a), (c)–(i), (k)–(m)	Extraneous adjacent conn. (8), Connector envy (3)

Restricted groups of clauses define less expressive, but useful logics. Five of these logics and the specific clauses that define them are shown in Table 1. The third column indicates which logic is required to specify each of the four architectural smells. Note that an actual recursion is not required by any of the four constraints, which may allow defining them in a less expressive logic. The translation in Definition 4 provides the correspondence between the structural constraint extension and the logic, which is used to classify the smells.

Fully enriched μ-calculus formulæ are interpreted over Kripke models.

Definition 2. A *Kripke model* for the fully enriched μ-calculus is a triple $\mathfrak{M} = (W, R, V)$ where

- W is a non-empty set of *worlds*, also called *states* or *points*;
- $R : Mod \to W \times W$ is a *relation function* that yields, for a given atomic modal symbol α, a binary relation on W; and
- $V = V : Prop \uplus Nom \to \mathcal{P}(W)$ is a *valuation function* that returns the set of worlds where a given propositional symbol or nominal holds. □

The interpretation of formulæ is described relying on the notation as follows: the expression $\mathfrak{m}[d \mapsto r]$ denotes a map $\mathfrak{m}'$ in which $\mathfrak{m}'(d') = \mathfrak{m}(d')$ for all $d' \neq d$ and $\mathfrak{m}'(d) = r$ otherwise; the set of values in the domain mapped by $\mathfrak{m}$ is called its *support*, and is denoted as $supp(\mathfrak{m})$.

The meaning of a state formula is defined in terms of sets of W, as it is described in Definition 3.

Definition 3. Let $\mathfrak{M}$ be a Kripke structure for the fully enriched μ-calculus, and $\mathfrak{s} : XVar \to \mathcal{P}(W)$ be a *state environment* that yields a set of worlds for a given state variable. The set of worlds that satisfy a state formula $\varphi \in SForm$ (Definition 1) is given by the interpretation function $[\![\cdot]\!]_{\mathfrak{s}} : SForm \to \mathcal{P}(W)$ inductively defined as

$$[\![\top]\!]_{\mathfrak{s}} \triangleq W \tag{1}$$

$$[\![p]\!]_{\mathfrak{s}} \triangleq \{w \in W : w \in V(p)\} \tag{2}$$

$$[\![\neg\varphi]\!]_{\mathfrak{s}} \triangleq W \setminus [\![\varphi]\!]_{\mathfrak{s}} \tag{3}$$

$$[\![\varphi \wedge \psi]\!]_{\mathfrak{s}} \triangleq [\![\varphi]\!]_{\mathfrak{s}} \cap [\![\psi]\!]_{\mathfrak{s}} \tag{4}$$

$$[\![\langle\beta\rangle\varphi]\!]_{\mathfrak{s}} \triangleq \{w \in W : \exists w' \in W.(w, w') \in \mathcal{S}[\beta] \wedge w' \in [\![\varphi]\!]_{\mathfrak{s}}\} \tag{5}$$

$$[\![X]\!]_{\mathfrak{s}} \triangleq \mathfrak{s}(X) \tag{6}$$

$$[\![\nu X.\varphi]\!]_{\mathfrak{s}} \triangleq \bigcup \{W' \subseteq W : W' \subseteq [\![\varphi]\!]_{\mathfrak{s}'}\} \text{ with } \mathfrak{s}' = \mathfrak{s}[X \mapsto W'] \tag{7}$$

$$[\![\langle n, \beta\rangle\varphi]\!]_{\mathfrak{s}} \triangleq \{w \in W : n <| \{w' \in W : (w, w') \in \mathcal{S}[\beta] \wedge w' \in [\![\varphi]\!]_{\mathfrak{s}}\} |\} \tag{8}$$

$$[\![\mathbf{E}\varphi]\!]_{\mathfrak{s}} \triangleq \begin{cases} W & \text{if } \exists w \in W.w \in [\![\varphi]\!]_{\mathfrak{s}} \\ \emptyset & otherwise \end{cases} \tag{9}$$

$$[\![i]\!]_{\mathfrak{s}} \triangleq \{V(i)\} \tag{10}$$

$$[\![@_i\varphi]\!]_{\mathfrak{s}} \triangleq \begin{cases} W & \text{if } V(i) \in [\![\varphi]\!]_{\mathfrak{s}} \\ \emptyset & otherwise \end{cases}, \tag{11}$$

provided that

$$\mathcal{S}[\beta] = \begin{cases} \mathcal{R}[\alpha] & \text{if } \beta = \alpha \\ \mathcal{R}[\alpha]^{\circ} & \text{if } \beta = \alpha^{\circ}, \end{cases}$$

$fsv(\varphi) \subseteq supp(\mathfrak{s})$, and $fsv(\varphi)$ denotes the free state variables of φ. □

Definition 4 presents the translation that takes structural constraints, built as described in Fig. 2, and yields a fully enriched μ-calculus formula. A notational convention adopted to present the translation is to consider non-terminals of the grammar as sets. For instance, $f \in F$ is used to indicate that expression f is built according non-terminal F. In addition, the substitution of `x` by `i` in a constraint is denoted as $[x/i]$.

Definition 4. Given a constraint $c \in SConst$, consisting of an optional quantifier $q \in Q$, an expression $f \in F$ and optional recursion definitions $rs \in Rec*$, the translation $\mathcal{T} : SConst \to SForm$ is defined as follows:

$$\mathcal{T}(q, f, rs) = \begin{cases} \bigwedge_{i \in Nom_{var:type}} @_i\, \mathcal{T}(f, \mathcal{R}(rs))[x/i] & \text{for } q = \text{all x:type} \\ \bigvee_{i \in Nom_{var:type}} @_i\, \mathcal{T}(f, \mathcal{R}(rs))[x/i] & \text{for } q = \text{exists x:type} \\ \bigwedge_{i \in Nom_{var}} @_i\, \mathcal{T}(f, \mathcal{R}(rs))[x/i] & \text{for } q = \text{all x} \\ \bigvee_{i \in Nom_{var}} @_i\, \mathcal{T}(f, \mathcal{R}(rs))[x/i] & \text{for } q = \text{exists x} \end{cases}$$

$$\mathcal{T}(f, rs) = \mathcal{T}(f, \mathcal{R}(rs))$$

where the translation of the recursion definitions is carried out by function $\mathcal{R} : Rec* \to (ID \to SForm)$ defined as

$$\mathcal{R}(r\ rs, V) = \begin{cases} \mathcal{R}(rs, V[ID \to \mu\ ID.\mathcal{T}(f, rs)]) & \text{for } r = \text{finite ID f} \\ \mathcal{R}(rs, V[ID \to \nu\ ID.\mathcal{T}(f, rs)]) & \text{for } r = \text{infinite ID f} \end{cases}$$

$$\mathcal{R}([\,], V) = V$$

with $t \in$ `(finite|infinite)`, $f \in F$, $rs \in Rec*$, and $V \in ID \to SForm$, and the translation of $f \in F$ defined as

$$\mathcal{T}(\text{True}, V) = \top \qquad \mathcal{T}(\text{False}, V) = \bot$$

$$\mathcal{T}(\text{p}, V) = p \qquad \mathcal{T}(\text{not f}, V) = \neg\mathcal{T}(\text{f}, V)$$

$$\mathcal{T}(\text{f or g}, V) = \mathcal{T}(\text{f}, V) \vee \mathcal{T}(\text{g}, V) \qquad \mathcal{T}(\text{f and g}, V) = \mathcal{T}(\text{f}, V) \wedge \mathcal{T}(\text{g}, V)$$

$$\mathcal{T}(\text{f implies g}, V) = \mathcal{T}(\text{f}, V) \to \mathcal{T}(\text{g}, V) \qquad \mathcal{T}(\text{f iff g}, V) = \mathcal{T}(\text{f}, V) \leftrightarrow \mathcal{T}(\text{g}, V)$$

$$\mathcal{T}([\text{m}]\ \text{f}, V) = [\mathcal{M}(m)]\ \mathcal{T}(\text{f}, V) \qquad \mathcal{T}([\text{n,m}]\ \text{f}, V) = [n, \mathcal{M}(m)]\ \mathcal{T}(\text{f}, V)$$

$$\mathcal{T}(\langle \mathrm{m} \rangle \mathrm{f}, V) = \langle \mathcal{M}(m) \rangle \, \mathcal{T}(\mathrm{f}, V) \qquad \mathcal{T}(\langle \mathrm{n,m} \rangle \mathrm{f}, V) = \langle n, \mathcal{M}(m) \rangle \, \mathcal{T}(\mathrm{f}, V)$$
$$\mathcal{T}(\mathrm{A\ f}, V) = \mathbf{A}\, \mathcal{T}(\mathrm{f}, V) \qquad \mathcal{T}(\mathrm{E\ f}, V) = \mathbf{E}\, \mathcal{T}(\mathrm{f}, V)$$
$$\mathcal{T}(\mathrm{id}, V) = V(\mathrm{id}) \qquad \mathcal{T}(\mathrm{i}, V) = i$$
$$\mathcal{T}(\mathrm{at\ i}, V) = @_i$$
$$\mathcal{M}(\mathrm{m}) = m \qquad \mathcal{M}(\mathrm{m\text{-}}) = m^\circ$$

where $m \in \mathit{Mod}$. □

Then, the translation of the constraint in Listing 6 yields formula

$$\mathbf{A}\ (\mathit{ElementInstance} \rightarrow \neg([2, \mathit{prt}]\bot \wedge \langle \mathit{prt} \rangle \mathit{In} \\ \wedge \langle 2, \mathit{act} \rangle\ \mu \mathit{Function}.(\mathit{FuncA} \vee \mathit{FuncB} \vee \mathit{FuncC}))).$$

A partial Kripke model for the architecture in Listing 1 is shown in Fig. 3. The model is partial since worlds representing names and their relationships are omitted. Each node in the graphic represents a world and includes: an identifier in the first line; the satisfied propositions in the second line; and the satisfied nominals

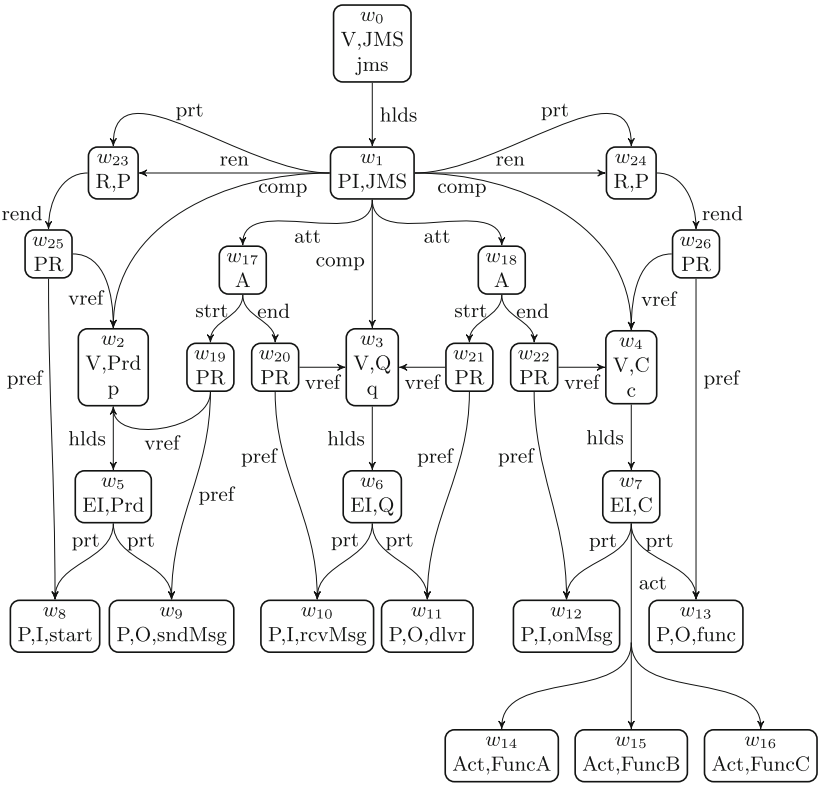

Fig. 3. Partial Kripke model for the JMS example configuration

in the third line. A short code is used instead of the actual name of propositions. The codes are: V (Variable), PI (PatternInstance), EI (ElementInstance), P (Port), I (In), O (Out), A (Attachment), R (Renaming), PR (PortReference), Act (Action), Q (Queue), Prd (Producer), and C (Consumer).

The verification of the formula is as follows:

$$
\begin{aligned}
& \mathbf{A}\ (\mathit{ElementInstance} \rightarrow \neg([2,\mathit{prt}]\bot \wedge \langle \mathit{prt}\rangle \mathit{In} \\
& \wedge \langle 2,\mathit{act}\rangle\ \mu \mathit{Function}.(\mathit{FuncA} \vee \mathit{FuncB} \vee \mathit{FuncC}))) \\
= \quad & \{\ \text{duality and definition of implication}\ \} \\
& \mathbf{A}\ (\neg \mathit{ElementInstance} \vee \neg(\neg\langle 2,\mathit{prt}\rangle\neg\bot \wedge \langle \mathit{prt}\rangle \mathit{In} \\
& \wedge \langle 2,\mathit{act}\rangle\ \neg\nu \mathit{Function}.(\neg(\mathit{FuncA} \vee \mathit{FuncB} \vee \mathit{FuncC})))) \\
= \quad & \{\ (2)\ \text{and duality}\ \} \\
& \mathbf{A}\ (\neg\{w_5,w_6,w_7\} \vee \neg(\neg\langle 2,\mathit{prt}\rangle\top \wedge \langle \mathit{prt}\rangle\{w_8,w_{10},w_{12}\} \\
& \wedge \langle 2,\mathit{act}\rangle\ \neg\nu \mathit{Function}.(\neg \mathit{FuncA} \wedge \neg \mathit{FuncB} \wedge \neg \mathit{FuncC}))) \\
= \quad & \{\ (1),\ \text{duality, and}\ (2)\ \} \\
& \mathbf{A}\ \neg(\{w_5,w_6,w_7\} \wedge (\neg\langle 2,\mathit{prt}\rangle W \wedge \langle \mathit{prt}\rangle\{w_8,w_{10},w_{12}\} \\
& \wedge \langle 2,\mathit{act}\rangle\ \neg\nu \mathit{Function}.(\neg\{w_{14}\} \wedge \neg\{w_{15}\} \wedge \neg\{w_{16}\}))) \\
= \quad & \{\ (3),(4),(7),\ \text{and}\ (3)\ \text{again}\ \} \\
& \mathbf{A}\ \neg(\{w_5,w_6,w_7\} \wedge (\neg\langle 2,\mathit{prt}\rangle W \wedge \langle \mathit{prt}\rangle\{w_8,w_{10},w_{12}\} \\
& \wedge \langle 2,\mathit{act}\rangle\ \{w_{14},w_{15},w_{16}\})) \\
= \quad & \{\ \text{duality,}\ (8),\ (5)\ \text{and}\ (8)\ \text{again}\ \} \\
& \neg\mathbf{E}\ (\{w_5,w_6,w_7\} \wedge (\neg\emptyset \wedge \{w_5,w_6,w_7\} \wedge \{w_7\})) \\
= \quad & \{\ (3)\ \text{and}\ (4)\ \} \\
& \neg\mathbf{E}\ (\{w_7\}) \\
= \quad & \{\ (9)\ \text{and}\ (3)\ \} \\
& \emptyset.
\end{aligned}
$$

Then, the constraint is not satisfied by the architecture in Listing 1, *i.e.*, the architecture contains an instance of the ambiguous interface smell.

5 Conclusion and Future Work

This paper proposes the usage of the Archery ADL to verify that software architectures are free of architectural smells found in catalogue [6]. The approach consists in specifying the absence of smells as constraints, and then verifying that architectures satisfy them. The constraint language is translated to a fully enriched μ-calculus, whose syntax and semantics are described. An architectural smell is detected in an example architecture, by showing that it fails to verify the corresponding constraint.

Future work includes the extension of the constraint language to cover the behaviour of instances and of reconfiguration scripts, and the development of a

verification tool. The application of the language to case studies in Healthcare and e-Gov is also part of future developments.

Acknowledgment. This work was funded by ERDF - European Regional Development Fund, through the COMPETE Programme, and by National Funds through FCT within project FCOMP-01-0124-FEDER-028923.

References

1. Aberdour, M.: Achieving quality in open-source software. Softw. IEEE **24**(1), 58–64 (2007)
2. Barbosa, L.S., Henriquez, P.R., Sanchez, A.: Towards rigorous analysis of open source software. In: Proceedings of the 5th International Workshop on Harnessing Theories for Tool Support in Software, TTSS 2011, University of Oslo (2011)
3. Bonatti, P.A., Lutz, C., Murano, A., Vardi, M.Y.: The complexity of enriched μ-calculi. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4052, pp. 540–551. Springer, Heidelberg (2006)
4. Bowman, I.T., Holt, R.C., Brewster, N.V.: Linux as a case study: its extracted software architecture. In: Proceedings of the 21st International Conference on Software Engineering, ICSE 1999, pp. 555–563. ACM, New York (1999)
5. Garcia, J., Popescu, D., Edwards, G., Medvidovic, N.: Identifying architectural bad smells. In: Proceedings of the 2009 European Conference on Software Maintenance and Reengineering, CSMR 2009, pp. 255–258. IEEE Computer Society, Washington, DC (2009)
6. Garcia, J., Popescu, D., Edwards, G., Medvidovic, N.: Toward a catalogue of architectural bad smells. In: Mirandola, R., Gorton, I., Hofmeister, C. (eds.) QoSA 2009. LNCS, vol. 5581, pp. 146–162. Springer, Heidelberg (2009)
7. Mehta, N.R., Medvidovic, N., Phadke, S.: Towards a taxonomy of software connectors. In: Proceedings of the 22Nd International Conference on Software Engineering, ICSE 2000, pp. 178–187. ACM, New York (2000)
8. Sanchez, A., Barbosa, L.S., Riesco, D.: A language for behavioural modelling of architectural patterns. In: Proceedings of the Third Workshop on Behavioural Modelling, BM-FA 2011, pp. 17–24. ACM, New York (2011)
9. Sanchez, A., Barbosa, L.S., Riesco, D.: Bigraphical modelling of architectural patterns. In: Arbab, F., Ölveczky, P.C. (eds.) FACS 2011. LNCS, vol. 7253, pp. 313–330. Springer, Heidelberg (2012)
10. Sanchez, A., Barbosa, L.S., Riesco, D.: Verifying bigraphical models of architectural reconfigurations (short paper). In: Proceedings of the 7th International Symposium on Theoretical Aspects of Software Engineering, TASE 2013, Birmingham, UK. IEEE (2013)
11. Sanchez, A., Barbosa, L.S., Riesco, D.: Specifying structural constraints of architectural patterns in the ARCHERY language. In: Proceedings of the International Conference of Numerical Analysis and Applied Mathematics 2014 (ICNAAM 2014): Symposium on Computer Languages, Implementations and Tools (SCLIT). AIP Proceedings (2014, to appear)
12. Sanchez, A., Oliveira, N., Barbosa, L.S., Henriques, P.: A perspective on architectural re-engineering. Sci. Comput. Program. **98**, 764–784 (2014)

13. Tatebe, O., Morita, Y., Matsuoka, S., Soda, N., Sekiguchi, S.: Grid datafarm architecture for petascale data intensive computing. In: 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid, May 2002, pp. 102–102 (2002)
14. Taylor, R.N., Medvidovic, N., Dashofy, E.M.: Software Architecture: Foundations, Theory, and Practice. Wiley, Chichester (2009)