Specifying structural constraints of architectural patterns in the ARCHERY language

Alejandro Sanchez, Luis S. Barbosa, and Daniel Riesco

Citation: AIP Conference Proceedings **1648**, 310008 (2015); doi: 10.1063/1.4912561
View online: http://dx.doi.org/10.1063/1.4912561
View Table of Contents: http://scitation.aip.org/content/aip/proceeding/aipcp/1648?ver=pdfcov
Published by the AIP Publishing

Specifying Structural Constraints of Architectural Patterns in the ARCHERY Language

Alejandro Sanchez[*,†], Luis S. Barbosa[†] and Daniel Riesco[*]

[*] *Departamento de Informática, Universidad Nacional de San Luis, Ejército de los Andes 950, D5700HHW San Luis, Argentina*
[†] *HASLab INESC TEC and Universidade do Minho, Campus de Gualtar, 4710-057 Braga, Portugal*

Abstract. ARCHERY is an architectural description language for modelling and reasoning about distributed, heterogeneous and dynamically reconfigurable systems in terms of architectural patterns. The language supports the specification of architectures and their reconfiguration. This paper introduces a language extension for precisely describing the structural design decisions that pattern instances must respect in their (re)configurations. The extension is a propositional modal logic with recursion and nominals referencing components, *i.e.*, a hybrid μ-calculus. Its expressiveness allows specifying safety and liveness constraints, as well as paths and cycles over structures. Refinements of classic architectural patterns are specified.

Keywords: software architecture, architectural patterns, formal verification
PACS: 89.20.Ff

INTRODUCTION

Architectural patterns and dynamic reconfiguration play a fundamental role in the the design of a software system. An (architectural) pattern packs a set of design decisions which are applicable to a recurring problem [1], and its application is expected to result in a known balance among a collection of quality attributes. Dynamic reconfiguration, on the other hand, allows modifying software systems in order to preserve quality attributes as conditions vary. However, as rearrangements of the system take place, design decisions entailed by a pattern may be violated.

ARCHERY [2, 3] is an *architectural description language* (ADL) that aims at addressing reconfiguration as a first class concern. Its basic specification concept is that of an architectural pattern, which comprises a set of architectural elements (connectors and components) specified in terms of their interfaces (set of ports) and behaviours. An architecture describes a particular configuration of instances of pattern elements through a set of attachments linking their ports, and a set of renamings changing the externally visible names of ports. An architecture can itself be regarded as an instance of the corresponding pattern, exhibiting an emergent behaviour. Both patterns and elements act as types of configurations, which are kept and referenced through typed variables. The language supports hierarchical composition. Dynamic reconfiguration is specified by scripts consisting of operations intended to cope with the creation and removal of instances, attachments, renamings and variables, as well as with moving instances.

This paper proposes a language extension based on a decidable hybrid μ-calculus [4] to precisely describe design decisions. Formulæ, as it is usual in modal logics, are interpreted over a Kripke model consisting of a set W of worlds and a family $\{\mathscr{R}_m\}_{m\in Mod}$ of binary relations among them, with Mod a set of relation labels. The basis is a propositional logic that includes modalities allowing the inspection of the model's relations. A μ-calculus is obtained by adding fixed points to such a basis, enabling the specification of recursive formulæ, and thus of liveness and safety conditions. The addition of hybrid features results in a hybrid μ-calculus. They 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. Hybrid logic is strictly more expressive than the usual modal languages. For example, it makes possible to express the equality between two worlds, to denote that a world is accessible through $\mathscr{R}_m$ from another world, or to assert the irreflexivity of $\mathscr{R}_m$. Moreover, in combination with fixed points, it makes possible to describe acyclic structures.

Structural constraints are associated to either patterns or to pattern instances. A pattern constraint describes a design principle that must be verified by any of its instances during the application of reconfiguration scripts. A pattern

Proceedings of the International Conference on Numerical Analysis and Applied Mathematics 2014 (ICNAAM-2014)
AIP Conf. Proc. 1648, 310008-1–310008-5; doi: 10.1063/1.4912561

instance constraint represents a design decision that must only be respected by such instance in its (re)configurations. Given a specific instance, its configuration constituents and their relationships become the worlds and the relationships, respectively, of a derived Kripke model. Then, a structural constraint is verified by translating it to a hybrid μ-calculus formula and then interpreting it over the derived model.

Tool supported development and analysis of reconfigurable architectural models entail the need for a formal underlying semantics. Reference [5] provides an extensive discussion of this issue and proposes a classification of ADLs based on the style of semantics adopted. Two groups emerge as particularly important: process algebra and graph-based approaches. The Darwin [6] and Wright [7] ADLs are example of the former, and the ADR ADL [8] combines both approaches. ARCHERY models the structural and behavioural dimensions with a kind of graphs called bigraphs [3] and a process algebra [2], respectively. Design decisions packed in patterns can be enforced either *by construction*, or either *by restriction* [9]. While ADR [8] uses the former mechanism leaving constraints implicit, Darwin [6] and ARCHERY follow the latter approach making constraints explicit. The underlying logic of Darwin is a first order logic which is not decidable, whereas structural constraints in ARCHERY are formulated in a hybrid μ-calculus, which is known to be decidable [4].

The proposed ARCHERY's extension is illustrated through the structural characterization of refinements of the *pipes and filters* and the *blackboard* architectural patterns. The former prescribes architectures built up of two elements, respectively named *pipe* and *filter*, that are arranged to transform a stream of data. The pattern restricts architectures to acyclic configurations in which filters are only connected through pipes. The blackboard pattern has the elements *knowledge source* and *blackboard* as building blocks. An architecture has a single blackboard instance, a repository, through which knowledge sources collaborate towards achieving a common objective.

THE ARCHERY LANGUAGE

This section describes the ARCHERY language in a brief and partial way (detailed descriptions can be found in [2, 3]). An ARCHERY specification comprises global data specifications (not part of the examples in this paper), one or more patterns and a variable that references the main architecture.

```
1  pattern PipeFilter()
2  element Pipe()
3   interface in acc; out fwd;
4  element Filter()
5   interface in rec; out snd;
6  end
```

Listing 1: Pipes and filters

```
1  pattern Blackboard()
2  element Rep()
3   interface in rcv:Nat; out snd:Nat;
4  element Ks(Nat)
5   interface in rcv:Nat; out snd:Nat;
6  end
```

Listing 2: Blackboard

```
1  b1:Blackboard=architecture Blackboard()
2  instances rep:Rep=Rep();
3   ks1:Ks=architecture PipeFilter()
4   instances p1:Pipe=Pipe();
5    f1:Filter=Filter(); f2:Filter=Filter();
6   attachments
7    from f1.snd to p1.acc;
8    from p1.fwd to f2.rec;
9   interface
10   f1.rec as rcv; f2.snd as snd;  end
11 attachments
12  from ks1.snd to rep.rcv;
13  from rep.snd to ks1.rcv; end
```

Listing 3: Blackboard architecture

```
1  b2:Blackboard=architecture Blackboard()
2  instances rep:Rep=Rep();
3   ks1:Ks=Ks(); ks2:Ks=Ks();
4  attachments
5   from ks1.snd to ks2.rcv;
6   from rep.snd to ks1.rcv; end
7  pf1:PipeFilter=architecture PipeFilter()
8  instances p:Pipe=Pipe();
9   f1:Filter=Filter(); f2:Filter=Filter();
10 attachments
11  from f1.snd to p.acc;
12  from p.fwd to f2.rec;
13  from f2.snd to f1.rec; end
```

Listing 4: Incorrect architectures

A pattern defines one or more architectural elements. Listings 1 and 2 show the specification of the pipes and filters, and the blackboard architectural patterns.

Each (architectural) element includes an *interface* that contains one or more ports. Each *port* is defined by a polarity, either `in` or `out`, a name, and an optional associated data type. For instance, in Listing 2 the interface of `Rep` defines in line 3 two ports with data type `Nat`. An element can optionally include a description of its behaviour, which is not considered in the sequel.

A variable (see line 1 of Listing 3) 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 defines a set of variables and describes the configuration adopted by their instances. It contains a token that must match a pattern name, an optional list of actual arguments, a set of variables, an optional set of attachments, and an optional interface. The actual arguments must match in type and order those of the pattern acting as its type. Each variable in the set must have as type an element defined in the pattern the architecture is an instance of. Listing 3 shows a nested architecture and Listing 4 specifies two architectures that violate the design principles of the pattern they were supposed to be instances of. 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 a port of the variable's instance. Then, an attachment indicates which output port communicates with which input port — see e.g. `f1.snd` with `p1.acc` in line 7 of Listing 3. 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 7. Ports not included in this set are not visible from the outside.

RESTRICTING THE STRUCTURE OF ARCHITECTURAL PATTERNS

Structural constraints are verified over architectures specified in the ARCHERY language. The architectural model is verified by interpreting it as a Kripke model. The meta-model of an architecture in the language is shown in Figure 1. The worlds in W are the set of constituents of the meta-model: names, instances, ports, variables, port references, attachments and renamings. The relationships among constituents conform the family $\{\mathscr{R}_m\}_{m\in Mod}$ of relations. The labels of relationships in Figure 1 become the modality symbols $m \in Mod$. For convenience, the modality symbol `attd` is included to name the relationship that relates two worlds representing variables connected through an attachment. It is obtained as $\mathscr{R}^{\circ}_{vref} \circ \mathscr{R}^{\circ}_{strt} \circ \mathscr{R}_{end} \circ \mathscr{R}_{vref}$, were $\mathscr{R}^{\circ}$ denotes the converse of a relation. Propositions test if a specific condition is present at a (world) w. They are classified into three groups according to the condition they evaluate: *a*) *Meta-type* propositions hold when w belongs to a specific participant set, *e.g.*, `PatternInstance`. *b*) *Emptiness* is checked by a single proposition, namely `Empty`, which holds when w is a variable with no associated instance. *c*) *Type* propositions depend on the pattern definition. They test if w is an instance or a variable of a type. For example, the pipes and filters pattern generates three proposition symbols: `Filter`, `Pipe` and `PipeFilter`.

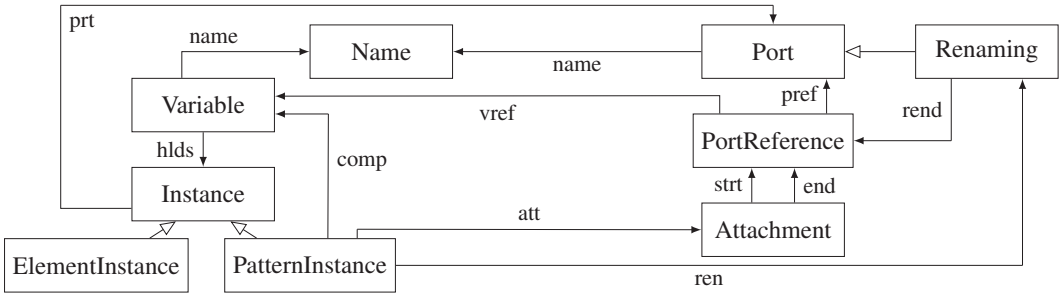

FIGURE 1. Relations and roles in spatial specifications

Structural constraints are associated to a pattern or to a pattern instance. Well-formed constraints are specified according to the grammar below. They allow describing in a precise manner the design decisions packed by the pipes and filters, and the blackboard patterns, as it is shown in Listings 5 and 6 (dots replace element definitions).

A constraint contains a formula that is either a propositional formula, a modal formula, a global modality formula, a recursive formula, or a hybrid formula (see the grammar below). In a modal formula, a `<M>F` indicates that there

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

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 $\mathscr{R}_{\text{MOD}}$ in the Kripke model, or the converse $\mathscr{R}^{\circ}_{\text{MOD}}$ indicated with MOD−. Global modality formulæ $\mathbf{E}F$ and $\mathbf{A}F$ are as $\langle M \rangle F$ and $[M]F$ but with $W \times W$ as the underlying relation. For instance, in constraint bbconn of Listing 6, for all (worlds modelling) variables (of type) Ks, any attachment, if exists, leads to a variable Rep. This constraint makes precise the design principle of the blackboard requiring that knowledge sources can only connect to a repository. The constraint is not satisfied by the configuration b2 in Listing 4. Constraint pfconn adopts a similar approach to address the analogous design principle of the pipes and filters pattern.

In recursive formulæ an ID designates a formula, indicating whether the recursion is expected to be finite or infinite. In line 10 of Listing 5, a finite recursion is specified in which it is either possible to go to another variable through an attachment, or a specific variable has been reached.

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.

The quantifiers **all** and **exists** only occur in the beginning of a constraint and have as domain the variables of the configuration, *i.e.*, the instances in a pattern instance. For each variable, a nominal is defined and included in a set Nom_{TYPEID}. The meaning of an **all** x:TYPEID F is the conjunction of formulæ **at** i F, for each $i \in Nom_{TYPEID}$. For instance, the resulting conjunction for the constraint acycle in Listing 5 and the configuration pf1 in Listing 4 is (**at** f1 (**not** <attd> PathToX) **and** (**at** f2 (**not** <attd> PathToX), where x is replaced by the names of the corresponding variables of the type. The constraint, that represents a design principle of the pipes and filters, is not satisfied by the configuration, as there are attachments establishing a cycle. The other constraint that uses this operator is singleton, which is satisfied by the configuration b2 of Listing 4. The meaning of an **exists** x:TYPEID F, is a disjunction of formulæ **at** i F, for each $i \in Nom_{TYPEID}$.

```
1  pattern PipeFilter() ...
2  structural constraints
3  const pfconn
4   A (((Filter and Var) implies
5    [attd] not Filter)) and
6     ((Pipe and Var) implies
7    [attd] not Pipe))); end
8  const acycle
9   all x:Filter not <attd> PathToX;
10  finite PathToX = <attd> PathToX or x;
11 end end
```

Listing 5: Pipes and filters

```
1  pattern Blackboard() ...
2  structural constraints
3  const singleton
4   all x:Rep A (not x implies not Rep);
5  end
6  const bbconn
7   A (Ks and Var) implies [attd] Rep;
8  end end
```

Listing 6: Blackboard

This extended abstract introduces the syntax and an informal semantics of an extension to the ARCHERY language allowing the specification of structural constraints. The extension aims at making precise design decisions packed in two well-known architectural patterns. As (re)configurations take place, invalid arrangements are detected, such as the ones shown in Listing 4. Its decidable underlying logic allows expressing safety and liveness constraints, as well conditions such as the acyclic property of structures. Future work includes the application of the language to case studies in Healthcare and e-Gov, the extension of the constraint language to cover the behaviour of instances and of reconfiguration scripts, and the development of a verification tool.

REFERENCES

1. R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software architecture : foundations, theory, and practice*, Wiley, 2009.
2. A. Sanchez, L. S. Barbosa, and D. Riesco, "A language for behavioural modelling of architectural patterns," in *Proceedings of the Third Workshop on Behavioural Modelling*, BM-FA '11, ACM, New York, NY, USA, 2011, pp. 17–24.
3. A. Sanchez, L. S. Barbosa, and D. Riesco, "Bigraphical Modelling of Architectural Patterns," in *Formal Aspects of Component Software. 8th International Symposium, FACS 2011, Oslo, Norway, September 14-16, 2011, Revised Selected Papers*, Springer, Berlin Heidelberg, 2012, vol. 7253 of *LNCS*, pp. 313–330.
4. U. Sattler, and M. Y. Vardi, "The Hybrid μ-Calculus," in *Proceedings of the First International Joint Conference on Automated Reasoning*, IJCAR '01, Springer-Verlag, London, UK, UK, 2001, pp. 76–91.
5. J. S. Bradbury, J. R. Cordy, J. Dingel, and M. Wermelinger, "A survey of self-management in dynamic software architecture specifications," in *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, WOSS '04, ACM, New York, NY, USA, 2004, pp. 28–33.

6. I. Georgiadis, J. Magee, and J. Kramer, "Self-organising software architectures for distributed systems," in *Proceedings of the first workshop on Self-healing systems*, WOSS '02, ACM, New York, NY, USA, 2002, pp. 33–38.
7. R. Allen, and D. Garlan, *ACM Trans. Softw. Eng. Methodol.* **6**, 213–249 (1997).
8. R. Bruni, A. Lluch Lafuente, U. Montanari, and E. Tuosto, *Bulletin of the EATCS* **94**, 161–180 (2008).
9. R. Bruni, A. Bucchiarone, S. Gnesi, D. Hirsch, and A. Lluch Lafuente, "Graph-Based Design and Analysis of Dynamic Software Architectures," in *Concurrency, Graphs and Models*, edited by P. Degano, R. Nicola, and J. Meseguer, Springer Berlin Heidelberg, 2008, vol. 5065 of *LNCS*, pp. 37–56.