Analysing Tactics in Architectural Patterns

Alejandro Sanchez*‡, Ademar Aguiar†, Luis S. Barbosa‡, Daniel Riesco*
*Universidad Nacional de San Luis, Ejercito de los Andes 950, D5700HHW San Luis, Argentina
{asanchez,driesco}@unsl.edu.ar
†INESC TEC & Universidade do Porto, Rua Dr. Roberto Frias, 4200-465 Porto, Portugal
ademar.aguiar@fe.up.pt
‡HASLab INESC TEC & Universidade do Minho, Campus de Gualtar, 4710-057 Braga, Portugal
{asanchez,lsb}@di.uminho.pt

***Abstract*—We present an approach to analyse the application of tactics in architectural patterns. We define and illustrate the approach by resorting to Archery, a language for specifying, analysing and verifying architectural patterns. The approach consists of characterising the design principles of an architectural pattern as constraints, expressed in the language, and then, establishing a refinement relation based on their satisfaction. The application of tactics preserving refinement preserves the original design principles expressed themselves as constraints for the architectural pattern. The paper's focus on fault-tolerance tactics, and identifies a set of requirements for a semantic framework characterising them. Model transformations to represent their application are discussed and illustrated through two case studies.**

I. Introduction

Architectural patterns and tactics play a fundamental role during the design of a software system. An architectural pattern packs a set of design decisions that are applicable to a recurring problem [1], and its application is expected to result in a known balance among a collection of quality attributes. A tactic identifies a single design decision that aims at achieving a specific quality attribute [2]. Both of them shape the structure and behaviour of the software system and facilitate understanding and documenting its design.

Catalogs for architectural patterns and for tactics aim at underpinning the architectural design process. In the former case, see, for example, [3], they usually include a description of the problem-solution pair, the consequences and the *rationale* behind them. In the latter case, like in [2], [4], only descriptions of the problem-solution are typically provided. Architectural patterns and tactics can be selected from these catalogs according to the quality attributes required by the business case of the system.

However, predicting whether quality attributes are preserved by the emergent structure and behaviour of a combination of these design decisions is not straightforward. The relation among them needs to be carefully considered in each case, since a tactic may work along, or against, the design principles that a pattern embodies [5]. Reference [6], investigates this relation for fault-tolerance tactics and provide a classification of the impact that a tactic has on a pattern according to five values: *good fit*, *minor changes*, *neutral*, *significant changes*, and *poor fit*. This classification helps as a guidance, but is not enough for ensuring that a specific quality attribute is satisfied up to a given level. In order to achieve this, an approach relying on more precise grounds is required.

This work aims at contributing towards such an end. We provide precise models relying on Archery [7], [8], an experimental language for the specification, analysis and verification of architectural patterns. Then, we characterise tactics by indicating what structural and behavioural modifications are entailed by their application to a given Archery specification.

The language is structured as a core – Archery-Core – and extensions – Archery-Script and Archery-Constraint – built on top of it. Archery-Core allows to specify the behaviour and configuration of a software system in terms of architectural patterns. A pattern specification in the language comprises a set of architectural elements (connectors and components) and their associated behaviours and interfaces (set of ports). An architecture describes a particular configuration of instances of a pattern's elements as a set of attachments among their ports and a description of the externally visible ports. An architecture can be regarded itself as an instance of the corresponding pattern, exhibiting an emergent behaviour. Then, patterns and elements act as types of expected behaviour and structure. Instances are kept and referenced through variables that have a type. The language supports hierarchical composition of architectural patterns, allowing the definition of configurations by indifferently attaching ports of pattern or element instances. Archery-Script is used to specify scripts for (re)configuring software architectures. Finally, the purpose of Archery-Constraint is to characterise patterns' design principles through constrains on their behaviour or structure.

Archery semantics is given by a translation to mCRL2 [9] and by an encoding into *bigraphical reactive systems* [10] for the behavioural and structural parts, respectively. mCRL2 is a specification language for reactive systems which combines a process algebra [11] for describing system's behaviours, with higher-order abstract equational data types, for handling structured data domains. Behavioural constraints can also be described in the modal μ-calculus. Translating the behavioural component of Archery descriptions into mCRL2, provides access to the associated toolset enabling simulation, visualisation, behavioural reduction and verification. Bigraphical reactive systems (BRS) and their theory, on the other hand, were developed to study systems in which locality and linking of computational agents varies independently. They provide a

general unifying theory in which different calculi for concurrency and mobility can be represented. Structural constraints are expressed in a subset of first order logic.

We characterise tactics by indicating which modifications they entail to the structure and behaviour of an Archery specification. In [5], the authors provide a textual description of this sort of modifications in an architectural pattern. We follow a different approach characterising these modifications in terms of Archery. Once a modification is executed, it is possible to verify which constraints no longer hold and which still do. It is also possible to visualise and animate the behaviour of the new configuration (namely, resorting to a translator to mCRL2), which helps to assess and understand the consequences of the modification.

We illustrate our approach by developing two case studies. In both cases we apply tactics to patterns but one differs from the other in the impact [6] of each application. One them has at least neutral impact applications; the other one is a case of poor fit.

This paper is organised as follows: section II briefly describes the Archery language; section III recalls fault-tolerance tactics and the modifications enforced in an architectural pattern by adopting them. The precise characterisation of a number of these fault-tolerance tactics; is presented in section IV. Finally, section V develops the case studies and section VI concludes.

II. The Archery Language

Archery is structured as a core language and two extensions, referred to as Archery-Core, Archery-Script, and Archery-Constraint, respectively. The basic language, Archery-Core, was originally introduced in [7] and the extensions in [8]. Archery-Script adds operations to build configurations, whereas Archery-Constraint offers primitives for defining constraints.

A. Archery-Core

An Archery-Core model comprises global data specifications, one or more patterns and a main architecture. We first describe how patterns and their elements are specified and then how an architecture is formed.

1) Patterns and elements: A pattern (see Figure 1 for its syntactic structure) has a unique identifier and includes an optional list of formal parameters, and one or more architectural elements. Listing 1 show the specification of two typical architectural patterns: Client-Server and Pipes and Filters.

Each architectural element in a pattern is described by an identifier, an optional list of formal parameters, and a description of its *behaviour* and an *interface*. The third consists of a list of actions and a list of process expressions, specified in a slightly modified subset of mCRL2, whose head is the element's initial behaviour. This process expressions yield a sequential process that is equivalent to a labelled transition system. An example of a declaration of a list of action definitions and of a process expression are shown, respectively. in lines 3 and 4 of Listing 1. They represent the behaviour

Pat	::=	**pattern** TYPEID (*IdsDecl*?) Elem+ **end**
Elem	::=	**element** TYPEID (*IdsDecl*?) *Behaviour ElemInterface*
Behaviour	::=	*ActSpec ProcSpec*
ActSpec	::=	**act** *ActDecl*+
ActDecl	::=	*Ids* (**:** *Domain*)?**;**
ProcSpec	::=	**proc** *MainProcDecl ProcDecl**
MainProcDecl	::=	ID(*MainProcPar*? (**,** *MainProcPar*)*) **=** *ArProcExpr*
MainProcPar	::=	*IdDecl* **=** InitValue
ProcDecl	::=	ID(*IdsDecl*? (**,** *IdsDecl*)*) **=** *ArProcExpr*
ArProcExpr	::=	ID
		| ID(*DataExprs*)
		| DELTA
		| TAU
		| **(** *ArProcExpr* **)**
		| *ArProcExpr* **.** *ArProcExpr*
		| **(***DataExpr* **)** –>*ArProcExpr* (<>*ArProcExpr*)?
		| *ArProcExpr* **+** *ArProcExpr*
ElemInterface	::=	**interface** *Port*+
Port	::=	(**in** | **out**) Ids **;**

Fig. 1. Archery Pattern Syntax

of instances of element `Server`, which receive a request (`rreq`), compute a response (`cres`), send a respose (`sres`), and iterate. The corresponding LTS is shown in Figure 2.

```
1  pattern ClientServer()
2  element Server()
3    act rreq, sres, cres;
4    proc Server() = rreq.cres.sres.Server();
5  interface in rreq; out sres;
6  element Client()
7    act sreq, rres;
8    proc Client() = sreq.rres.Client();
9    interface in rres; out sreq;
10 end
11 pattern PipeFilter()
12 element Pipe()
13   act accept, forward;
14   proc Pipe() = accept.forward.Pipe();
15   interface in accept; out forward;
16 element Filter()
17   act rec, trans, send;
18   proc Filter() = rec.trans.send.Filter();
19   interface in rec; out send;
20 end
```

Listing 1. Archery Example Patterns

An extension currently under development is the possibility of indicating time constraints for actions. For instance, the action that follows the reception of a request occurs no later than five units of time, *i.e.*, `rreq@t.cres@(u <= t +5)`.

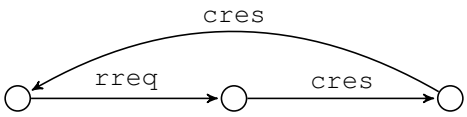


Fig. 2. LTS for Server

The interface, on the other hand, contains one or more ports.

Each *port* is defined by a polarity, either `in` or `out`, and a token that must match an action name in the list of action definitions. For instance, the interface of `Client` defines two ports in line 5. Archery adopts a water flow metaphor for ports: an `in` port receives input from *any* port connected to it, and an `out` port sends output to *all* ports connected to it. Ports are synchronous: actually a suitable process algebra expression can be used to emulate any other port behaviour.

2) Patterns and element instances: A variable has an identifier and a type that must match an element or pattern name. Allowed values are, of course, instances of elements or patterns. Note that the variable that follows the pattern definitions, must contain an architecture (the main one).

Var	::=	ID **:** TYPEID **=** *Inst* **;**
Inst	::=	(*ElemInst* | *PatInst*)
ElemInst	::=	TYPEID **(** *DataExprs*? **)**
PatInst	::=	**architecture** TYPEID **(** *DataExprs*? **)** *ArchBody* **end**
ArchBody	::=	*Instances Attachments*? *ArchInterface*?
Instances	::=	**instances** *Var*+
Attachments	::=	**attachments** *Att*+
Att	::=	**from** *PortRef* **to** *PortRef* **;**
ArchInterface	::=	**interface** *Ren*+
Ren	::=	*PortRef* **as** ID **;**
PortRef	::=	ID**.**ID

Fig. 3. Archery Instance Syntax

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. As an example, a nested architecture is defined in Listing 2 (lines 4 - 14).

```
1  cs:ClientServer =
2    architecture ClientServer()
3  instances
4    s:Server = architecture PipeFilter()
5    instances
6      f1:Filter = Filter();
7      f2:Filter = Filter();
8      p1:Pipe = Pipe();
9    attachments
10     from f1.send to p1.accept;
11     from p1.forward to f2.rec;
12   interface
13     f1.rec as rreq;   f2.send as sres;
14   end
15   c1:Client=Client();  c2:Client=Client();
16 attachments
17   from c1.sreq to s.rreq;
18   from c2.sreq to s.rreq;
19   from s.sres to c1.rres;
20   from s.sres to c2.rres;
21 end
```

Listing 2. Archery Example Pattern Instance

Each attachment includes a port reference to an output port, and another one to 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.send` with `p1.accept` in line 10.

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 13. Ports not included in this set are not visible from the outside. Including the same port in an attachment and the interface is incorrect.

B. Archery-Script

Archery-Script is used to specify scripts for creating architectures or for reconfiguring existing ones. It assumes the existence of a process that triggers a script under some conditions. Its combinators are informally described in Table I and the use of some of them is illustrated through the example in Listing 3. An essential feature is that their definition is independent of any pattern. The design principles for patterns are enforced through constraints, as shown in Section II-C. This independence, and the fact that a variable may contain an instance whose type may not necessarily match the variable's type, allows the reuse of a script in an open family of patterns (related by some refinement relation).

```
1  script
2    import("initial");  // first part
3    s2 : Server;
4    s2 = Server();
5    addInst(cs, s2);
6    detach(cs.c2.sreq, cs.s1.rreq);
7    detach(cs.s1.sres, cs.c2.rres);
8    attach(cs.c2.sreq, cs.s2.rreq);
9    attach(cs.s2.sres, cs.c2.rres);
10   import("pf");  // second part
11   f3 : Filter = new Filter();
12   addInst(pf, f3);
13   attach(pf.p1.forward, pf.f3.rec);
14   remRen(pf.sres);
15   addRen(pf.f3.send, sres);
16   move(pf, cs.s2);
17   c3 : Client =  Client();  // third part
18   addInst(cs, c3);
19   detach(cs.c2.sreq, cs.s2.rreq);
20   detach(cs.s2.sres, cs.c2.rres);
21   attach(cs.c2.sreq, cs.c3.rres);
22   attach(cs.c3.sreq, cs.c2.rres);
23 end
```

Listing 3. Example script

The example in Listing 3 is divided in three parts and assumes the existence of an initial configuration denoted by $cs_{initial}$. The configuration is similar to the one in Listing 2, but for the fact that the nested architecture (between lines 4 and 14) is now replaced by a `Server` instance (in a single line `s1:Server=Server();`). The first part of the example reconfigures $cs_{initial}$ by adding and connecting a

TABLE I
COMBINATORS FOR ARCHERY-SCRIPT

Name	Format	Description
Import	`import(s)`	Receives as a parameter a reference `s` to an Archery specification and imports it to the environment of the executing script (*e.g.*, line 2 in Listing 3)
Create Variable	`v:type`	Creates a variable with name `v` and type `type` (line 3)
Create Instance	`v=type()`	Creates a new instance of type `type` and assigns it to a variable `v` (line 4)
Remove Variable	`remVar(v)`	Removes a variable `v` and any instance it references
Add Instance	`addInst(a,v)`	Adds a variable `v` and the instance it refers to to the architecture in variable `a` (line 5)
Remove Instance	`remInst(a,v)`	Removes a variable `v` and the instance it refers to from the architecture in variable `a`
Attach	`attach(f.o, t.i)`	Attaches the port `o` of the instance in variable `f` to the port `i` of the instance in variable `t` (line 8)
Deattach	`detach(f.o, t.i)`	Removes the attachment between the port `o` of the instance in variable `f` and the port `i` of the instance in variable `t` (line 6)
Add Rename	`addRen(v.p,q)`	Renames port `p` in variable `v` to `q` (line 15)
Remove Rename	`remRen(v.q)`	Removes rename `q` in the architecture in variable `v` (line 14)
Move	`move(s,t)`	Whatever is referred by variable `s` becomes referred by variable `t` (line 16); the reference to the contents of `t` is lost, but its attachments and renamings remain

second server. It starts with an import operation that leaves the configuration in variable `cs`. Operations in lines 3 and 4 create a new variable `s2` and assign a fresh instance of `Server` to it. Upon that, `s2` is included in the architecture in `cs`. Then, the operations in the next two lines remove the attachments of instances referred to by variables `cs.c2` and `cs.s1`. Subsequently, new attachments are created between the instance in variable `cs.c2` with the instance in variable `cs.s2`. The resulting configuration is referred in the sequel as cs_{first}.

The second part of the example starts in line 10 and shows how the interface of an architecture is modified and a server is replaced. It assumes the existence of a configuration pf, similar to the one described between lines 4 and 14 in Listing 2, stored in a variable `pf` of type `PipeFilter`. The script imports such a configuration, creates a new instance of `Filter` in variable `f3` and includes it in `pf`. Line 14 removes renaming `sres` from `pf`. This has a similar effect to deleting line 13 from Listing 2. Then, a new renaming is included in the interface, but now for port `send` in variable `pf.f3`. Subsequently, the instance in `pf` is moved to variable `cs.s2`. The instance referred to by variable `cs.s2` is now the architecture of type `PipeFilter`, but attachments and renamings, set for the previous instance, remain.

The third part begins in line 17. It creates a new client and connects it in an incorrect way. A new variable `c3` is created and a new instance of the type `Client` is assigned to it. Next, the fresh variable is included in the architecture in `cs`. Subsequently, the attachments between the instances in variables `cs.c2` and `cs.s2` are removed. Then, the script creates two attachments between instances in variables `cs.c3` and `cs.c2`. The resulting configuration, referred to as cs_{wrong} in the sequel, violates the design principle underlying a Client-Server architecture by connecting two clients.

C. Archery-Constraint

This extension allows for the definition of constraints to characterise and verify design principles for architectural patterns. A constraint φ can be defined either for an element E – *local*, or for a pattern P – *global*. In both cases, constraints either refer to the structure or to the behaviour, and their evaluation respectively resorts to the bigraphical encoding [8] and the LTS [7] for the specification. The satisfaction relations, $E \models \varphi$ and $P \models \varphi$, are precisely defined somewhere else [12]. We illustrate these possibilities in two cases: first a global structural constraint, then a local behavioural one.

Ruling out incorrect configurations, such as cs_{wrong} above, entails the need for a global structural constraint characterising what the valid instances of a pattern are. Since the variable cs in the script, shown in Listing 3, is of type `ClientServer`, we could add to the pattern specification a constraint φ to express that clients can only connect to servers and vice versa. We define φ for all attachments att in an architecture of type `ClientServer` as follows:

$$client(from(att)) \Leftrightarrow server(to(att)) \wedge \\ client(to(att)) \Leftrightarrow server(from(att)) \qquad (1)$$

where $from$ (respectively, to) is a function that returns the variable with an out (respectively, in) port in att, and where $client$ (respectively, $server$) is a predicate valid whenever its argument is of type `Client` (respectively, `Server`).

By constraining patterns in this way, one can prevent the inclusion in a script of operations which may generate invalid configurations. Clearly, cs_{wrong} does not satisfy the constraint above. In contrast, configuration cs_{first} does.

Assume now that we want to prevent the situation in which a client issues a second request before receiving a response back. This can be achieved by associating to the element `Client` the constraint below.

$$[sreq.(\overline{sreq \cup rres}) * .sreq]\ false$$

The expression states that for any instance of element `Client` in an architecture of type `ClientServer`, it is not possible to issue a request (`sreq`), and then other actions different from sending a request (`sreq`) or receiving a response (`rres`), and then send a second response (`sreq`).

III. FAULT-TOLERANCE TACTICS

The objective of fault-tolerance tactics is to prevent that a fault in a system becomes a service failure [13]. They were first described as tactics in [2] and subsequently refined in [4]. Designing a system for fault-tolerance requires detecting,

recovering from, and preventing faults. Typical measures for achieving such capacities are:

- *Fault detection.*
  - *Ping-echo.* A process learns whether another is reachable and what is the roundtrip delay by exchanging asynchronous request/response messages.
  - *Watchdog.* Monitored processes periodically send messages to reset a timer in the monitor. If the timer expires, the monitor assumes that the monitored process is suffering a fault.
  - *Heartbeat.* A monitor periodically sends messages to the monitored processes and expects a response within a time frame.
  - *Exception detection.* A condition that will alter the normal execution of a process can be detected.
  - *Voting.* A cluster of redundant processes receives the same input and send their output to a voting process that can detect inconsistencies.
- *Fault recovery – preparation and repair.*
  - *Active redundancy.* A set of redundant processes, one active and the other spare, receive and process in parallel the same input and, in consequence, are ready to be exchanged.
  - *Passive redundancy.* An active process among a set of redundant ones, receives and processes all inputs. It also sends periodic state updates to the other (spare) processes.
  - *Spare.* An active process receives and process all inputs while it periodically persists its state at checkpoints. Upon a fail in the active process, another process is initialised with the last persisted state, and processing is resumed from that checkpoint.
  - *Exception handling.* Upon the detection of an exception, an alternative course of execution is started.
  - *Software upgrade.* A mechanism that allows to upgrade the behaviour of a service during runtime without affecting its delivery.
- *Fault recovery – reintroduction.* Processes are corrected upon a failure and reintroduced.
  - *Shadow.* An process that failed runs for a period of time as a spare before it acquires an active role.
  - *State resynchronisation, rollback.* It is a mechanism that allows an process to acquire a state that is equivalent to the one of the active process. A rollback is the case in which the state corresponds to a checkpoint and is acquired from a persistent repository.
  - *Escalating restart.* It allows to restart different sets of processes according to a level of granularity with the aim of minimising the impact on the service delivery.
  - *Non-stop forwarding.* The management of a service is separated from its delivery in a way that allows the service delivery to continue despite the management part failed.
- *Fault prevention.*
  - *Removal from service.* It is a mechanism to prevent that a process with a fault receives a request to fulfil any service in the system.
  - *Transactions.* Allows the atomic, consistent, isolated and durable (ACID) exchange of asynchronous messages among distributed processes.
  - *Process monitor.* A process that monitors the health of a process in order to ensure that it is operating within its nominal parameters.
  - *Exception prevention.* It refers to mechanisms that prevent exceptions from occurring.

IV. Encoding Fault-tolerance Tactics

Characterising the application of fault-tolerance tactics requires a semantic framework for the source and target specifications, and a representation of the transformation. The former must allow modelling the intended structure an behaviour of the tactics in Section III. We discuss requirements for such a framework and identify which of them can be expressed in the Archery language. Subsequently, we provide an informal description of the transformations needed to apply tactics on such models, and a criteria to evaluate whether new constraints are needed.

A. Semantic Framework Requirements

The description of any of the tactics requires a minimal set of behavioural operators (B), data types and variables (D), and structure (S). Operators in B include sequential composition, alternative choice, and parallel composition, supported by standard process algebras [11].

From the textual description of tactics in [2], [4], summarised in Section III, we recognise the need for modelling three more specific behavioural concerns. The need for modelling time (T) emerges from tactics such as watchdog or heartbeat. In cases like escalating restart, software upgrade or shadow, there is a need for creating new processes (P) which requires a proper semantic foundation [14]. The third issue we consider emerges from tactics such as exception detection and exception handling, that require modelling the interruption or alteration of the normal execution of a process, and compensation procedures (C) [15]. In Table II we show what are the requirements for each tactic.

The Archery language supports B, D, S and support for T is under development. The process algebra operators (B) are either embedded or implicit in the language, data types and state (D) is inherited from mCRL2, and structural operators (S) rely on semantics given by a translation to Bigraphical Reactive Systems. Time (T) will be supported by extending the core of the language and the corresponding translation to a process algebra. In Section V we provide illustrative examples for some of the tactics that can be represented, *i.e.*, the ones that do not have an X beyond the fourth column.

B. Transformations

We consider three different cases as follows:

- *Reconfiguration.* The configuration changes but the pattern is not modified. This transformation can be specified in Archery-Script.

TABLE II
SEMANTIC FRAMEWORK REQUIREMENTS

Tactic	B	D	S	T	P	C
Ping-echo	X	X	X	X		
Watchdog	X	X	X	X		
Heartbeat	X	X	X	X		
Exception detection	X	X	X			X
Voting	X	X	X			
Active redundancy	X	X	X			
Passive redundancy	X	X	X			
Spare	X	X	X		X	
Exception handling	X	X	X			X
Software upgrade	X	X	X		X	
Shadow	X	X	X	X	X	X
State resynchronisation	X	X	X			
Rollback	X	X	X			
Escalating restart	X	X	X		X	X
Non-stop forwarding	X	X	X			X
Removal from service	X	X	X			X
Transactions	X	X	X			X
Process monitor	X	X	X			
Exception prevention	X	X	X			

- *Add/Remove pattern.* A pattern in the specification is either added or removed. This transformation contemplates the case in which instances in a configuration are hierarchically composed. A reconfiguration script must also indicate configuration changes.
- *Modify pattern.* The pattern is modified in any of the ways described below. A reconfiguration script may or may not be necessary, depending on the specific configuration under evaluation.
  - *Modify constraints.* A constraint (local or global) is either added, modified, or removed.
  - *Add/remove element.* An element in the pattern is either added or removed.
  - *Modify element.* When an element is modified check whether the modification only affects its internal behaviour or also its external (visible) one.

C. Evaluation

In order to evaluate whether a new constraint is needed, we define a refinement relation among patterns. Such a relation must relate specifications in a way that allows to distinguish when the changes in the target specification go either along or against the design principles of the patterns in the source one.

As it is natural, we define such relation by relying on both local and global constraints defined for the pattern with Archery-Constraint. We define an element E' to be a refinement ($\lesssim$) of E, if and only if, E' satisfies all constraints E does. The expression for such condition is

$$E' \lesssim E \Leftrightarrow \\ \forall\, \varphi \in const(E), E \models \varphi \Rightarrow E' \models \varphi \qquad (2)$$

, in which $const(E)$ is the set of constraints associated with element E. Then, a pattern P' is a refinement ($\lesssim$) of another pattern P', if each element E' in P' is a refinement of a an element E in P, and P' satisfies all global constraints of P. The corresponding expression is shown in (3).

$$P' \lesssim P \Leftrightarrow \\ \forall\, E' \in P', \exists\, E : E' \lesssim E \wedge \\ \forall\, \varphi \in cons(P), P \models \varphi \Rightarrow P' \models \varphi \qquad (3)$$

V. CASE STUDIES

We develop two case studies: one in which fault-tolerance tactics are applied to an architectural pattern without violating its design principles, and another in which the modifications work against such principles.

A. Fult-Tolerance Tactics in a Client-Server Configuration

Assume that the selected architectural pattern for a software system is Client-Server. The main design principle is, as described in Section II-C by expression 1, clients can only connect to servers and viceversa.

Each server offers a service built on top of a legacy system that may fail non-deterministically. We represent this by modifying the original pattern into the one shown in Listing 4. Note that in line 4, there is a non-deterministic choice that may interrupt the service loop. In the sequel, we will refer to the configuration between lines 11 and 19 as $conf_1$.

```
1  pattern ClientServerF()
2  element Server()
3    act rreq, sres, cres, fail;
4    proc Do()=rreq.(cres.sres.Do()+fail);
5    interface in rreq; out sres;
6  element Client()
7    act sreq, rres;
8    proc Do() = sreq.rres.Do();
9    interface in rres; out sreq;
10 end
11 base:ClientServerF =
12   architecture  ClientServerF()
13 instances
14   c1 : Client = Client();
15   s  : Server = Server();
16 attachments
17   from c1.sreq to s.rreq;
18   from s.sres to c1.rres;
19 end;
```

Listing 4. Client-Server with Failures

An important property of the Client-Server architectural pattern is that once a request is issued, a response must follow. For the particular case of $conf_1$, we can formulate constraint φ_1, which states that once `c1` issues a request, a response sent by `s` is inevitable.

$$\varphi_1 = [true * .synch(\texttt{c1}, \texttt{sreq}, \texttt{s}, \texttt{rreq})] \\ \mu X.([\overline{synch(\texttt{s}, \texttt{sres}, \texttt{c1}, \texttt{rres})}]X \wedge \langle true \rangle true)$$

This property is not satisfied, *i.e.*, $conf_1 \not\models \varphi_1$, because of the non-deterministic choice that interrupts the service. In order to address this problem we apply fault tolerance tactics.

We start by applying active redundancy. We do this by defining a server to be hierarchically composed by an instance of a Master-Slave pattern [16] refinement (shown in Listing 5). Note that the transformation incorporates a new pattern to the specification, but does not modify pattern `ClientServerF`.

```
pattern MasterSlave()
element Master(np:Int)
  act req, res, s, f;
  proc Do(n:Int=np) =
       req.s.f.res.Collect(n).Do(n);
    Collect(n:Int)=
      f.(n>0) -> Collect(n-1) <> tau;
  interface in req, f; out res, s;
element Slave()
  act s,d,f;
  proc Do() = s.d.f.Do();
  interface in s; out f;
end
```

Listing 5. Master-Slave with Active Redundancy

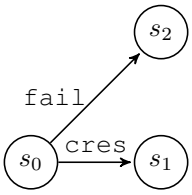
The server is refined as a `MasterSlave` instance that has a `Master` instance, and in turn, `Server` instances acting as `Slave` ones. The behaviour of the former begins when a request is received (`req`). Then, the request is forwarded to all slaves (`s`), and with the first response from any of them (`f`), a response to the requester is sent back (`res`). Subsequently, the rest of the responses, (from the other slaves) are collected. The configuration is shown in Listing 6 and in the sequel we will refer to it as $conf_2$. An Archery-Script can be defined to reconfigure $conf_1$ into $conf_2$. The new configuration, among other issues that may be observed, does not satisfy φ_1, *i.e.*, $conf_2 \not\models \varphi_1$.

```
base:ClientServer =
  architecture  ClientServer()
instances
  c1:Client = Client();
  s:Server = architecture MasterSlave()
  instances
    m:Master = Master(2);
    sl1:Slave = architecture ClientServer()
    instances
      sr1:Server = Server();
    interface sr1.rreq as s; sr1.sres as f;
    end
    sl2:Slave = architecture ClientServer()
    instances
      sr2:Server = Server();
    interface sr2.rreq as s; sr2.sres as f;
    end
  attachments
    from m.s to sl1.s; from m.s to sl2.s;
    from sl1.f to m.f; from sl2.f to m.f;
  interface m.req as rreq;  m.res as sres;
  end
attachments
  from c1.sreq to s.rreq;
  from s.sres to c1.rres;
end;
```

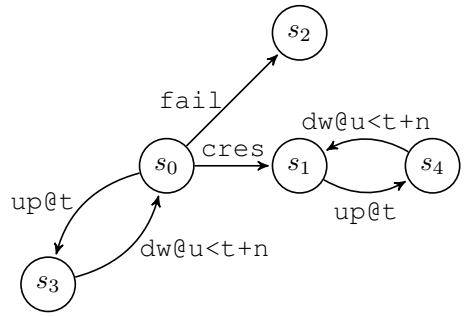
Listing 6. Client-Server with Active Redundancy

In order to ensure that a response is sent back, we need to apply two more tactics: heartbeat to detect a failure in the servers, and escalating restart to repair a failed one, besides having redundant servers. We achieve this by modifying the `MasterSlave` pattern. The `Master` element assumes the role of controller and the `Slave` the one of the controlled. We also modify the latter to have the behaviour of a `Server` and in this way we avoid modifying pattern `ClientServerF`, and having two levels of hierarchical composition. The design principles of the Master-Slave pattern are preserved by this pattern modification, since the structural arrangement is limited to a star configuration with a central Master instance.

We first study the application of the heartbeat tactic by discussing how the respective LTSs for the controlled and the controller processes are modified. Assume that the LTS in Figure 4(a) corresponds to a fragment of the behaviour that needs to be monitored, because it may fail, as in the case of a `Server` instance. In order to represent the behaviour of a controlled process, for each state we consider needs to be monitored and is part of the normal flow of control (s_0 and s_1), we add an outbound transition (`up`) to a new state (s_3 and s_4, respectively), that represents receiving a message from the controller, and an inbound transition (`dw`) from such new state that represents responding back before n units of time has passed. If the process fails, it will not respond to such message, as it is shown in Figure 4(b).



(a) Source Process

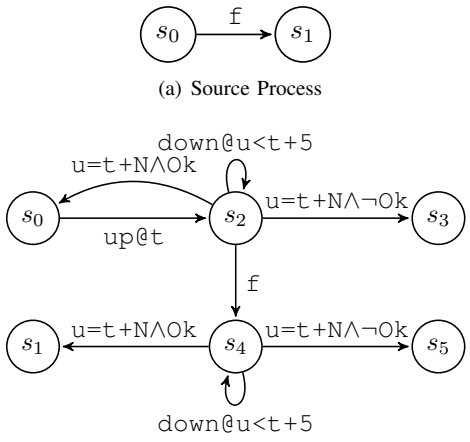


(b) Target Process

Fig. 4. LTS Modifications for a Heartbeat Controlled Processes

The modification to the controller is a little more involved, as it is shown in Figure 5. The controller expects a response (`f`) from each controlled process, as it is shown in Figure 5(a). We add states and transitions, as shown in Figure 5(b), to represent the periodical message sent to all controlled processes, the timeout period, and the reception of the expected message `f`. Note that state s_1 represents the situation in which one answer was received, and all processes responded to monitoring messages. Also note that states s_3 and s_4 respectively represent

the situations in which none of the processes has answered, and at least one of them has not answered. Both states are candidates for starting recovery procedures.

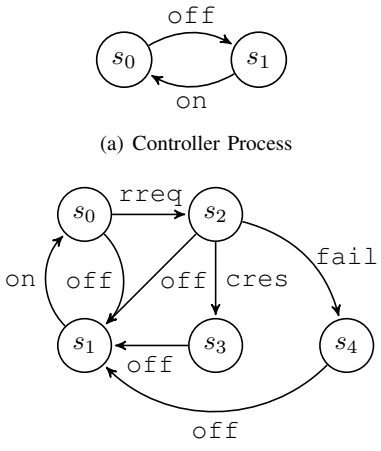


(a) Source Process

(b) Target Process

Fig. 5. LTS Modification for a Heartbeat Controller Processes

We show a specific approach for encoding and applying the escalating restart recovery tactic in Figure 6. Note that, as indicated in Table II, a general treatment requires modelling the interruption and creation of processes. The controller, when a situation requiring a restart is detected, issues an off, followed by an on signal (LTS fragment shown in Figure 6(a)). The controlled needs to consider receiving these messages at all states. This is illustrated by the modified LTS for a fragment of the `Server` behaviour in Figure 6(b). All states, with the exception of s_1, should have a loop transition for the reception of an on signal. This is not shown in the LTS.



(a) Controller Process

(b) Controlled Process

Fig. 6. LTS Fragments for Escalating Restart Processes

The resulting configuration still does not satisfy φ_1. The problem lies in that a loop in which servers always fail is possible. A property considering fairness, that such loop cannot go forever, would be satisfied by $conf_3$.

B. Fault-tolerance Tactics in a Pipes and Filters Configuration

In this case study we assume that latency is important for the software system under development. The architecture can be organised according to a refinement of the Pipes and Filters pattern, as in the case of the nested architecture between lines 4 and 14 in Listing 2.

The design principles of the selected pattern facilitate the analysis of the system's response times. It imposes that filters can only connect to pipes, and that there are no cycles in this connections. The expression below characterises such design principles.

$$\begin{aligned}(\forall\, att\ \in\ conf : attachment(att) \wedge \\ filter(from(att)) \Leftrightarrow pipe(to(att)) \wedge \\ pipe(to(att)) \Leftrightarrow filter(from(att))) \\ \wedge\, (\forall\, f \in\ conf : filter(f)\ \wedge \neg path(f, f))\end{aligned}$$

in which $conf$ is the configuration, $attachment(att)$ is a predicate that holds if att is an attachment, and $filter$ (respectively $pipe$) is a predicate yielding true when evaluated with a variable of type `Filter` (respectively, `Pipe`). Under this design principles, the system can be regarded as the functional composition of the filters. If the worst response time for each filter is known, then, the worst response time for the whole system can be estimated.

The application of some fault-tolerance tactics results in a pattern in which estimating such value by exploting a restricted configuration is no longer possible. Assume some of the flters may fail in a non-deterministic way. We can model this by modifying the pattern in Listing 1 to a pattern such as the one in Listing 7, with an extra element for a failing filter. The level of fault-tolerance of the system could be improved by applying active redundancy, heartbeat and escalating restart tactics on failing filters. However, it is no possible to use an approach based on hierarchical composition, as in the case study in Section V-A, and a controller component needs to be included in the pattern as well. The introduction of this controller breaks the design principles of the pattern, and in consequence prevents analysing the response time by exploiting a lineal configuration.

```
1  pattern PipeFilterF()
2  element Pipe ...  element Filter ...
3  element FilterF()
4    act rec, trans, fail, send;
5    proc Filter() =
6      rec.(fail + trans.send.Filter());
7    interface in rec; out send;
8  end
```

Listing 7. Pipes and Filters with Fails

The introduction of tactics not aligned with the design principles of the pattern can be detected by resorting to the refinement relation as described in Section IV-C. Assuming that it is possible to characterise the introduced controller component as a filter, an instance of such component requires a constant feedback from the filters, which creates a circular

path, and invalidates the constraint characterising the Pipes and Filter pattern. Then the transformed pattern is not a refinement of the original and in consequence, verifying if a specific level of a quality attribute is satisfied requires further analysis.

We resort to an Archery specification with time constraints for such analysis. Upon adding time constraints for the actions in filters and in the controller, we need to make one assumption. We are interested in the worst time with no failures, so we need to remove the non-deterministic choice in line 6 for our analysis. Once this is done, we can evaluate an expression such as the one below,

$$\begin{aligned} &sum\ t : Real. \\ &\quad [true * .action(p1, rec)@t. \\ &\quad true * .action(pn, send)@u] \\ &\quad \wedge (u \geq t + 10)\ false \end{aligned}$$

which must hold for the given configuration, and assumes that the filter receiving the input is in variable `p1`, the one sending the output is in `pn`, and rec and $send$ are actions in the respective interfaces of the instances referenced by the variables. The constraint indicates that it is not possible that a send ($send$) action occurs ten units of time later than the receive (rec) one. If it is verified for a configuration, we ensure that the worst response time is less than 10 units, provided no failure occurs.

VI. Conclusions

In this work we discussed an approach for analysing the application of tactics in architectural patterns by resorting, as long as possible, to precise foundations. We used Archery, a language for specifying, analysing and verifying architectural patterns, as a semantic foundation to define and illustrate it.

We identified a set of requirements for precisely modelling fault-tolerance tactics [2] and discussed the possible transformations a model may undergo when a tactic is applied to an architectural pattern. The set of requirements provides a guide for establishing which tactics can be expressed in a language (Archery in our case) and which not. It turns out that Archery covers an interesting range of them. We subsequently discussed a set of model transformations that serve for representing the application of tactics. What we presented differs from the one in [5] because we systematically covered the possibilities (since we aim at a tool-supported approach), rather than observing tactic applications in software system documentation.

Then, we provide a criteria to establish whether a tactic application goes against, or along, the design principles of an architectural pattern. We precisely characterise such design principles as constraints and we provide a refinement relation based on the satisfaction of such constraints. Then, a derived pattern that does not satisfy the constraints of the original one requires further attention since the design principles were not preserved by the tactic application. Once identified the unsatisfied constraints, it is possible to define new ones to assess to what extent, a quality attribute is fulfilled.

Finally, we illustrated our approach by developing two examples. One in which the design principles of the pattern were preserved, and another in which not.

As part of ongoing and future work we mention the definition of the model and logic to underpin the verification of constraints such as the ones described in this document. Specifically, we are exploring the translation of strutural constraints to *BiLog* [17] – a spatial logic for bigraphs.

Acknowledgment

This work is funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project **FCOMP-01-0124-FEDER-010047**.

References

[1] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[2] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*, 2nd ed. Addison-Wesley Longman Publishing Co., Inc., 2003.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Wiley, 1996.

[4] J. Scott and R. Kazman, "Realizing and refining architectural tactics: Availability," Carnegie Mellon University, Software Engineering Institute, Tech. Rep., 2009. [Online]. Available: http://www.sei.cmu.edu/library/abstracts/reports/09tr006.cfm

[5] N. B. Harrison and P. Avgeriou, "How do architecture patterns and tactics interact? a model and annotation," *Journal of Systems and Software*, vol. 83, no. 10, pp. 1735 – 1758, 2010.

[6] N. Harrison and P. Avgeriou, "Implementing reliability: The interaction of requirements, tactics and architecture patterns," in *Architecting Dependable Systems VII*, ser. Lecture Notes in Computer Science, A. Casimiro, R. de Lemos, and C. Gacek, Eds. Springer Berlin / Heidelberg, 2010, vol. 6420, pp. 97–122.

[7] 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*, ser. BM-FA '11. New York, NY, USA: ACM, 2011, pp. 17–24.

[8] ——, "Bigraphical modelling of architectural patterns (to appear)," in *Post-proceedings of the Eighth International Symposium on Formal Aspects of Component Software*, ser. FACS 2011. Springer, 2011.

[9] J. F. Groote, A. Mathijssen, M. Reniers, Y. Usenko, and M. van Weerdenburg, "The formal specification language mCRL2," in *Methods for Modelling Software Systems: Dagstuhl Seminar 06351*, 2007.

[10] R. Milner, *The space and motion of communicating agents*. Cambridge University Press, 2009, vol. 54.

[11] J. C. M. Baeten, T. Basten, and M. A. Reniers, *Process Algebra: Equational Theories of Communicating Processes*. Cambridge University Press, 2010.

[12] A. Sanchez, "A calculus of architectural patterns (to appear)," Ph.D. dissertation, Universidad Nacional de San Luis, 2012.

[13] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Dependable Secur. Comput.*, vol. 1, no. 1, pp. 11–33, Jan. 2004.

[14] R. Milner, *Communicating and Mobile Processes: the π-Calculus*. Cambridge University Press, 1999.

[15] R. Bruni, H. Melgratti, and U. Montanari, "Theoretical foundations for compensations in flow composition languages," *SIGPLAN Not.*, vol. 40, no. 1, pp. 209–220, Jan. 2005.

[16] F. Buschmann, K. Henney, and D. Schmidt, *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing*. John Wiley & Sons, 2007.

[17] G. Conforti, D. Macedonio, and V. Sassone, "Spatial logics for bigraphs," in *32th International Colloquium on Automata, Languages and Programming, ICALP 2005.*, vol. LNCS 3580. Springer, 2005, pp. 766–778.