Archery– Modeling and analysis of architectural patterns A quick guide

Alejandro Sanchez

April 24, 2012

Contents

1 Introduction

In a number of contexts the term architectural pattern is used as an architectural abstraction. The expression is taken in the usual sense in classical software architecture – a known solution to a recurring design problem. In [4] it is characterised as a description of element and configuration types, and a set of constraints on how to use them. Catalogs [5, 11], provide a vocabulary for their use at a high abstraction level. However, the lack of formality in their documentation prevents developing precise architectural specifications on top of them, and in consequence, any tool-supported analysis and verification.

Such is the motivation behind Archery [10, 9], a language for precise modelling and analysis of architectural patterns. The language is structured as a core that allows to specify the configuration and behaviour

of software architectures in terms of architectural patterns, and extensions, currently under development, to support (re)configuration operations and pattern constraints. 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 element's behaviour is a sequential process that each instance of such element obeys. Its interface is the set of actions that also represent interactions with other instances. An architecture describes a particular configuration that instances of a pattern's elements assume as a set of attachments among their ports and the setting of externally visible ports. It has an emergent behaviour and can be regarded as an instance of the pattern. Then, both patterns and elements act as types of behaviour expected from instances. 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.

The behavioural semantics of Archery are given by translation to a process algebra – mCRL2 [7, 6]. Process algebra [8, 3], broadly defined as *the study of the behaviour of parallel or distributed systems by algebraic means* [2], provides a suitable conceptual framework not only to describe software architectures, but also to *reason* about them either equationally (on top of well studied notions of behavioural equivalence), or through formulation and verification of behavioural requirements expressed in an associated modal logic. Moreover, Process Algebra supports compositional reasoning and abstraction with respect to internal activity of services or components a system is composed of. In particular, mCRL2 a process algebra, incorporating data and time information, with a number of features which turn it suitable for flexible modelling of behaviour. For example, the introduction of multi-actions enables the specification of (not necessarily related) actions that are to be executed together. Most process algebras only allow a single action to be executed atomically thus forcing an order on the execution of actions. They also allow the separation of parallelism and communication: a multiaction simply represents the simultaneous execution of a set of actions, action synchronisation being specified separately.

In this guide we describe how to write Archery specifications and how to carry out their analysis. The language defines relations among specification that can be proved with the mCRL2 tool-set. The tools also allow animating and visualising the specifications.

2 Syntax

2.1 Patterns and elements

An Archery model comprises global data specifications, one or more patterns and a main architecture (Figure 1). Global data specifications follow the mCRL2 syntax. We include in Figure 2 part of such syntax in order to express Archery's. However, we do not aim to replicate the extensive documentation describing it and available, for instance, in the toolset website[1].

Spec ::= *DataSpecs Pat+ Var*

Figure 1: Archery Specification

Domain ::= *SortExpr* (**#** *SortExpr*)*
DataExprs ::= *DataExpr* (**,** *DataExpr*)*
IdDecl ::= ID **:** *SortExpr*
IdsDecl ::= *Ids* **:** *SortExpr*
Ids ::= ID(**,** ID)*

Figure 2: Data Specifications

A pattern (see Figure 3 for the syntactic structure) has a unique identifier and includes an optional list of formal parameters, and one or more architectural elements. Two example patterns (`ClientServer`

[1] `http:\\www.mcrl.org`

and `PipeFilter`) are shown in Listing 1. The list of formal parameters responds to the mCRL2 syntax for declaring identifiers: a list of identifiers ending with a colon, followed by a sort expression.

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 **;**

Figure 3: Archery pattern

Each architectural element in a pattern is described by an identifier, an optional list of formal parameters, a description of its *behaviour* and an *interface*. The former consists of a list of actions and a list of process expressions whose head is the pattern's initial behaviour. Behaviour is specified in a slightly modified subset of mCRL2. An example list of action definitions and a process expression are respectively shown in lines 3 and 4 of Listing 1.

The interface, on the other hand, contains one or more ports. Each *port* is defined by a direction, 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 inspired in [1] 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.

Listing 1: Patterns – Client-Server and Pipes and Filters

```
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 prcs, sreq, rres;
8      proc Client() = prcs.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
```

2.2 Instances

A variable has an identifier and a type that must match an element or pattern name (figure 4). Allowed values are, of course, instances of elements or patterns.

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

Figure 4: Archery instance

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 between lines 3 and 14 of the example.

Each attachment includes a port reference to an `out` port, and another one to an `in` 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 `out` port communicates with which `in` port — see e.g. `f1.send` with `p1.accept` in line 9.

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. Ports not included in this set are not visible from the outside. Including the same port in an attachment and the interface is incorrect. An example interface with two renamings is shown between lines 12 and 13.

Listing 2: An example architecture

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

3 Analysis

3.1 Translation

Function $\mathcal{T}$ describes the translation of a pattern instance referenced by a variable. The result depends on whether interleave semantics are used or not, and whether it is intended for animating a specification or whether for comparing two. Clause (1) considers the former condition, and clause (2) considers the latter.

$$\mathcal{T}_{interleave}(Var) = \begin{cases} \rho_R(\nabla_A(\Gamma_C(\prod_{j \in Instances} \mathcal{T}_{interleave}(Var_j)))) & \Leftarrow interleave = true \\ \rho_R(\partial_B(\Gamma_C(\prod_{j \in Instances} \mathcal{T}_{interleave}(Var_j)))) & \Leftarrow otherwise \end{cases} \quad (1)$$

$$\mathcal{T}_{hide,interleave}(Var) = \begin{cases} \tau_H(\mathcal{T}_{interleave}(Var)) & \Leftarrow hide = true \\ \mathcal{T}_{interleave}(Var) & \Leftarrow otherwise \end{cases} \quad (2)$$

Clause (1) combines mCRL2 operators ρ, either ∇ or ∂ , and Γ, nested in that order. The *communication* operator Γ takes C as the set of communication rules and the parallel combination of the translation $\mathcal{T}$ of each variable Var_j in the architecture. Each rule *cr* in C is generated from each attachment in *Attachments*. The *allow* operator ∇ is used for interleave semantics. It takes the set A of action ids as parameter to allow only those in the set and restrict any other. Set A is computed as the union of the set of action identifiers obtained for each instance j in *Instances*, and the synchronisation actions in each cr of C. The *block* operator ∂, taking as parameter the set B of action ids, is used to enforce communications and to rule out undesired actions occuring independently. Set B contains the generated action ids to represent port's behaviour. The *rename* operator ρ is applied then with the set R of renamings. The calculation of the set R of rename rules is a little more tricky. For each *Ren* in *ArchInterface* seek the set *prs* of attached port references in the upper level. Then, for each *pr* in *prs* generate a rename rule *rr*.

Clause (2) shows the translation for comparing specifications. It surrounds clause (1) with the hiding operator τ. The argument, set H, contains the actions of element instances and the ids of synchronisation actions in C.

Element instances are translated as sequential processes by processing the behaviour defined in the element with respect to the instance information. The resulting process expression renames actions and replaces ports with suitable process expressions that reflects their behaviour.

3.2 Relations

Architectural models in Archery can be compared through the behavioural equivalences and preorders defined in mCRL2. Actually, rooted branching bisimilarity, $\approx_{RB}$, provides a basis for establishing architectural interchangeability with respect to interface behaviour. Branching bisimilarity [3] relates behaviours differing in the amount of internal activity but exhibiting similar branching structure. Rooted branching bisimilarity adds a rootedness condition: initial internal transitions are never inert. Formally,

$$a \equiv b \Leftrightarrow \mathcal{T}(a) \approx_{RB} \mathcal{T}(b) \quad (3)$$

$$a \sqsubseteq b \Leftrightarrow \mathcal{T}(a) \sqsubseteq_{RB} \mathcal{T}(b) \quad (4)$$

Coarser relationships are sometimes necessary to compare Archery models. Weak trace equivalence, $\approx_{WT}$, and refinement, which abstract from the internal branching structure, are used to define the architectural relations $\equiv_{WT}$ and $\sqsubseteq_{WT}$, respectively.

4 Tool support

4.1 Translating to mCRL2

4.1.1 Distribution

The translator is distributed in a compressed file with the contents as follows:

- `translator.jar` – a java implementation of the translator Archery to mCRL2
- `translator.sh` – a shell script for invoking the translator on Linux platforms
- `README.txt`
- `examples` – a folder with the example specifications in Section 5
- `translate-examples.sh` – a shell script to translate some of the examples

4.1.2 Command line

On Linux platforms the translator can be invoked with `translator.sh` according to the format in line 1. On other platforms it can be executed with the command on line 2

```
1 translate.sh [OPTIONS] [filename]
2 java -jar translator.jar [OPTIONS] [filename]
```

The translator supports a few options, each indicated in the command line by a leading double dash. The options are:

- `help` – text explaining format and options
- `version` – translator's version
- `compare` – (default) translation for comparing specifications
- `simulate` – translation for analysing a single specification
- `interleave` – (default) translated according to interleave semantics
- `non-interleave` – (default) translated allowing actions simultaneity

Example invocations you can find in file `translate-examples.sh` are shown below.

```
1 ./translator.sh --simulate --non-interleave examples/base.archery
2 ./translator.sh --compare examples/base.archery
3 ./translator.sh --simulate --interleave examples/buggy.archery
4 ./translator.sh examples/buggy.archery
```

The translation generates a file with extension `mcrl` that can be used for the analysis supported by the mCRL2 tool-set.

4.2 Generating a Linear Process Specification

In order to carry out any analysis we first need to create a Linear Process Specification (LPS) from the generated `mcrl` file. This is carried out with the `mcrl2lps` command according to the example that follows.

```
1 mcrl22lps --lin-method=stack --rewriter=jitty --delta --no-constelm --no-deltaelm --
      no-sumelm --verbose file.mcrl2 file.mcrl2.lps
```

Note that this process can take long, in particular if non interleave semantics are indicated when the mCRL2 file is generated by the translator (see Section 4.1.2).

A detailed description of the options can be found on mCRL2 website [2].

4.3 Simulating a specification

A specification can be animated with the command `lpsxsim`. The tool requires an LPS version of the original specification, which is preferably translated to mCRL2 with the `simulate` option (see Section 4.1.2). The command can be invoked as follows.

```
1 lpsxsim file.mcrl2.lps
```

A detailed description of the options can be found at the mCRL2 website [3] and a screen capture is shown in Section 5.2.

[2] http://www.mcrl2.org/release/user_manual/tools/mcrl22lps.html
[3] http://www.mcrl2.org/release/user_manual/tools/lpsxsim.html

4.4 Generating a Labelled Transition System

Some analysis can only be performed if the corresponding Labelled Transition System (LTS) is generated. This can be done with `lps2lts` command according to the example that follows.

```
1 lps2lts --rewriter=jitty --state-format=tree --strategy=breadth --verbose file.mcrl2.
      lps file.con.mcrl2.lts
```

An explanation of the command is available at the mCRL2 website [4].

4.5 Visualising the Labelled Transition System

The corresponding LTS of an specification can be visualised with the `ltsgraph` tool. It is recommended to use an mCRL2 file generated with the `simulate` option (see Section 4.1.2). The visualisation can be started with a command similar to the one below.

```
1 ltsgraph file.mcrl.lts
```

A detailed description of the command can be found in the mCRL2 website [5].

4.6 Proving equivalence and refinement relations

Equivalence and refinement relations, as described in Section 3.2, can be proved with the `ltscompare` tool. The example commands ending in lines 2, 4 and 6 respectively illustrate how to prove $spec1 \equiv spec2$, $spec1 \equiv_{WT} spec2$ and $spec1 \sqsubseteq_{WT} spec2$. The current version of the tool does no support proving $spec1 \sqsubseteq spec2$.

```
1 ltscompare --equivalence=branching-bisim --counter-example --verbose
2   spec1.mcrl.lts spec2.mcrl.lts
3 ltscompare --equivalence=weak-trace --counter-example --verbose
4   spec1.mcrl.lts spec2.mcrl.lts
5 ltscompare --preorder=weak-trace --verbose
6   spec1.mcrl.lts spec2.mcrl.lts
```

In the case of proving equivalence, the rootedness condition needs to be manually verified after branching bisimilarity is proved.

5 Examples

The Client-Server pattern prescribes configurations arranged by instances of two main element types: client and server. The main design principle of the pattern is that clients can only connect to servers and vice-verse. An element type for connectors is also part of the pattern, but we omit for simplicity in our example characterisations.

5.1 A single server

Listing 3 shows a Client-Server pattern characterisation and an instance of the server. The server receives a request `rreq`, computes a response `cres`, sends the response back `sres` and iterates `Do()`. We will refer to this configuration as $simple$ in the sequel.

Listing 3: A single server

```
1 pattern ClientServer()
2 element Server()
3   act rreq, sres, cres;
4   proc Do() = rreq.cres.sres.Do();
5   interface in rreq; out sres;
6 element Client()
```

[4] http://www.mcrl2.org/release/user_manual/tools/lps2lts.html
[5] http://www.mcrl2.org/release/user_manual/tools/ltsgraph.html

```
  act prcs, sreq, rres;
  proc Do() = prcs.sreq.rres.Do();
  interface in rres; out sreq;
end
s : Server = Server();
```

The LTS for the specification can be visualised with the `ltsgraph` tool. The visualisation for this example is shown in Figure 5. It requires translating the specification to mCRL2 for simulation (Section 4.1.2), generating the LPS (Section 4.2), and then the LTS (Section 4.4). Note that the ids for the action and the unconnected ports combine the action id with the instance id.

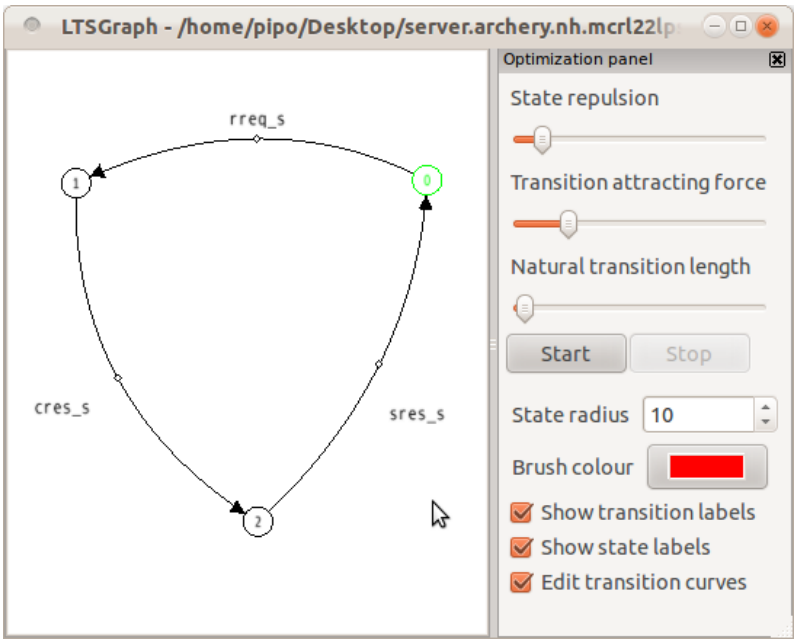

Figure 5: Server LTS

5.2 A server and two clients

An architecture consisting of two clients connected to a server is shown in Listing 4. In the sequel, we will refer to this configuration as *base*.

Listing 4: Base - a server and two clients

```
pattern ClientServer()
element Server()
  act rreq, sres, cres;
  proc Do() = rreq.cres.sres.Do();
  interface in rreq; out sres;
element Client()
  act prcs, sreq, rres;
  proc Do() = prcs.sreq.rres.Do();
  interface in rres; out sreq;
end
base : ClientServer = architecture  ClientServer()
  instances
    c1 : Client = Client(); c2 : Client = Client();
    s  : Server = Server();
  attachments
    from c1.sreq to s.rreq; from c2.sreq to s.rreq;
    from s.sres to c1.rres; from s.sres to c2.rres;
```

```
18 end;
```

The configuration can be animated as it is shown in Figure 6. It shows the two possible transitions at the beginning of the behaviour: `prcs_c1` and `prcs_c1`. The user can select which one to execute.

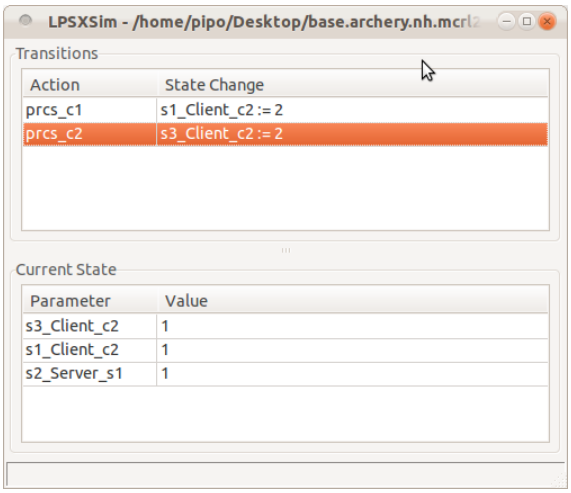

Figure 6: A server and two clients - Simulation

The LTS for the configuration, generated upon a translation using interleave semantics, is shown in Figure 7. Note the labels generated by the translator. We describe the ones for transitions leaving and reaching state 0. The former ones, `prcs_c1` and `prcs_c2`, represent the respective first actions in the behaviour of instances `c1` and `c2`. The latter ones, `synch_sres_s_rres_c1` and `synch_sres_s_rres_c2`, represent the respective interactions among the server instance and the two clients instances receiving the response from the server.

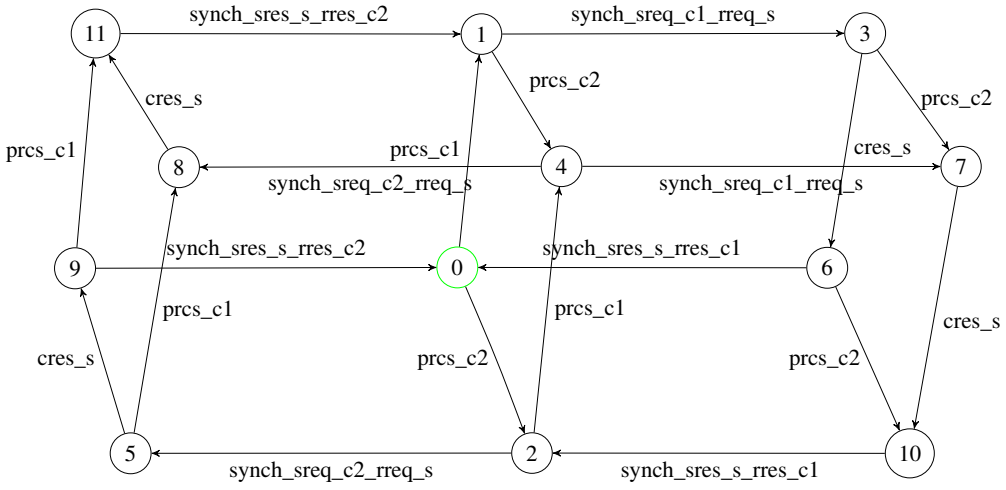

Figure 7: A server and two clients - LTS

5.3 A paid server

A pattern in which the server calculates a price for responding some request is shown in Listing 5. The behaviour of the server changes, as it adds in a non-deterministic way an extra action after `cres` that

represents the calculation of the price. In the sequel, we will refer to this configuration as $paid$.

Listing 5: A paid server

```
pattern ClientPaidServer()
element PaidServer()
  act rreq, sres, cres, cpri;
  proc Do() = rreq.(tau.cres.cpri + tau.cres).sres.Do();
  interface in rreq; out sres;
element Client()
  act prcs, sreq, rres;
  proc Do() = prcs.sreq.rres.Do();
  interface in rres; out sreq;
end
s : PaidServer = PaidServer();
```

We can compare configurations $simple$ and $paid$ and establish if whether they are interchangeable or not. Using the tool as described in Section 4.6 we prove $simple \equiv paid$.

5.4 A buggy server

We characterise a Client-Server pattern in which the server may stop working in a non-deterministic way as it is shown in Listing 6. We will refer to this configuration as $buggy$ in the sequel.

Listing 6: A buggy server

```
pattern ClientServer()
element BuggyServer()
  act rreq, sres, cres;
  proc Server() = rreq.(tau.delta + tau.cres).sres.Server();
  interface in rreq; out sres;
element Client()
  act prcs, sreq, rres;
  proc Client() = prcs.sreq.rres.Client();
  interface in rres; out sreq;
end
s : BuggyServer = BuggyServer();
```

We can study how configuration $buggy$ relates to $simple$ using the comparison tool. We prove that $buggy \not\equiv simple$, but we do have that $buggy \equiv_{WT} simple$.

5.5 A fault tolerant server

Assume that the server can detect errors, and inform the client that the processing of the request has failed (see 7). This is represented by adding a parameter that includes a boolean value to the actions for exchanging the response. We refer to this configuration as $tolerant$

Listing 7: A fault tolerant server

```
sort
  Data;
  Comm = struct msg(success:Bool,data:Data);
cons d, e: Data;
pattern ClientFaultTolerantServer()
element FaultTolerantServer()
  act rreq, cres; sres: Comm;
  proc Do() = rreq.cres.(tau.sres(msg(false,d)) + tau.sres(msg(true,e))).Do();
  interface in rreq; out sres;
element Client()
  act prcs, sreq; rres:Comm;
  proc Do() = prcs.sreq.rres(res).Do();
  interface in rres; out sreq;
end
s  : FaultTolerantServer = FaultTolerantServer();
```

We observe that none of the relations is valid: $simple \not\equiv tolerant$, $simple \not\equiv_{WT} tolerant$, and $simple \not\equiv_{WT} tolerant$. This is because the language generated by the actions of the interface has changed; they have a parameter in $tolerant$ not present in $simple$.

5.6 A composite server

It is possible to hierarchically combine patterns. The internal structure of a server can be defined in terms of pattern Pipe and Filter (see Listing 8). We refer to this configuration as $composite$.

Listing 8: A composite server

```
pattern ClientServer()
element Server()
  act rreq, sres, cres;
  proc Server() = rreq.cres.sres.Server();
  interface in rreq; out sres;
element Client()
  act prcs, sreq, rres;
  proc Client() = prcs.sreq.rres.Client();
  interface in rres; out sreq;
end
pattern PipeFilter()
element Pipe()
  act accept, forward;
  proc Pipe() = accept.forward.Pipe();
  interface in accept; out forward;
element Filter()
  act rec, trans, send;
  proc Filter() = rec.trans.send.Filter();
  interface in rec; out send;
end
s : Server = architecture PipeFilter()
instances
  f1 : Filter = Filter(); f2 : Filter = Filter();
  p1 : Pipe = Pipe();
attachments
  from f1.send to p1.accept; from p1.forward to f2.rec;
interface
  f1.rec as rreq; f2.send as sres;
end
```

Configuration $composite$ is not equivalent, nor weak trace equivalent to $simple$, *i.e.*, $composite \not\equiv simple$ and $composite \not\equiv_{WT} simple$. However, it is possible to check that $simple \sqsubseteq_{WT} composite$.

References

[1] F. Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, June 2004.

[2] J. Baeten. A brief history of process algebra. *Theoretical Computer Science*, 335(2/3):131–146, 2005.

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

[4] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice (2nd Edition)*. Addison-Wesley Professional, second edition, Apr. 2003.

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

[6] J. F. Groote, J. Keiren, A. Mathijssen, B. Ploeger, F. Stappers, C. Tankink, Y. Usenko, M. Weerdenburg, W. Wesselink, T. Willemse, and J. Wulp. The mcrl2 toolset. In *Proc. International Workshop on Advanced Software Development Tools and Techniques (WASDeTT 2008)*, 2008.

[7] 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.

[8] R. Milner. *Communication and Concurrency*. Series in Computer Science. Prentice Hall, 1989.

[9] A. Sanchez, L. S. Barbosa, and D. Riesco. Bigraphical modelling of architectural patterns (to appear). In *Post-proceedings of the 8th International Symposium on Formal Aspects of Component Software*, FACS 2011. Springer, 2011.

[10] 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, pages 17–24, New York, NY, USA, 2011. ACM.

[11] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Apr. 1996.