A Language for Behavioural Modelling of Architectural Patterns

Alejandro Sanchez
Departamento de Informática
Univ. Nacional de San Luis
San Luis, Argentina
asanchez@unsl.edu.ar

Luis S. Barbosa
DI - HASLab
Universidade do Minho
Braga, Portugal
lsb@di.uminho.pt

Daniel Riesco
Departamento de Informática
Univ. Nacional de San Luis
San Luis, Argentina
driesco@unsl.edu.ar

ABSTRACT

The complexity of interactions governing the coordination of loosely-coupled services, which forms the core of current software, brought behavioural issues up to the front of architectural concerns. This paper takes such a challenge seriously by lifting typical behaviour modelling techniques to the specification of both types and instances of architectural patterns in which the later ones are connected by ports that behave according to a water flow metaphor. A specific language is introduced for this purpose as well as a translator to mCRL2 so that the simulation and analysis techniques available in the corresponding toolset can be used to reason about (the behavioural layer of) software architectures. The approach is illustrated in a few examples.

Categories and Subject Descriptors

D.2 [**Software Engineering**]: Design tools and techniques, Software architectures, Software/program verification

General Terms

Software architecture

Keywords

Software architecture, Behaviour modelling, Architectural Pattern

1. INTRODUCTION

Continuous evolution towards very large, distributed, heterogeneous, highly dynamic computing systems has shifted both the focus and the method of Software Architecture as a generic design discipline. On the one hand the diversity and complexity of interactions governing the coordination of loosely-coupled services, which forms the core of current software, brought behavioural issues up to the front of architectural concerns. Actually, it is consensual to recognise that *the overall structure of software systems* [8], whose study was defined as the specific domain of this discipline in its founding papers, is essentially the structure of their internal interactions and the emergent macro behaviour they entail. On the other hand, informal representations of architectural designs, often relying on graphical notations with poor semantics, are unsuitable for high-assurance software development, precluding any form of rigorous analysis and verification.

In this context the motivation for this paper is twofold: emphasising behaviour modelling in architectural design and doing so in a rigorous, mathematically sound way. Process algebra [12, 4], broadly defined as *the study of the behaviour of parallel or distributed systems by algebraic means* [3], 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.

This paper introduces a language for specification of architectural patterns, on top of a mature and well-known Process Algebra framework — mCRL2 [10, 9]. The language provides mechanisms for defining types as well as instances of them. A type in the language can be defined as a basic architectural element (typically a service/component or connector) with an associated behaviour, or as a set of elements constituting an architectural pattern. In both cases, ports, interaction points that behave according to the intuitive water flow metaphor, can be defined for them. The language also supports hierarchical composition of architectural patterns, allowing the definition of configurations by indifferently attaching ports of pattern or element instances.

A translator of architectural specifications to the mCRL2 modelling language was developed, which brings up a collection of powerful tools for their simulation, graphical visualisations and analysis. mCRL2 is 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 multiactions 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.

BM-FA'11, Jun 06 2011, University of Birmingham, UK

After a brief background summary in section 2, the modelling language is introduced in section 3. Section 4 discusses how behavioural analysis of architectural patterns can be performed in this setting. After a presentation of the translation process, analysis in the mCRL2 framework is illustrated through a number of examples and the basis of a patterns calculus is discussed. Section 5 concludes with a comparison with related work and points some directions for future work.

2. BACKGROUND

A language for software design at the architectural level - Architectural Description Language (ADL), must provide a minimum of modelling capabilities and tool support. It is required to allow the explicit specification of both the structural and behavioural dimensions of components, connectors and configurations, and it must also provide tool support to develop and evolve systems in an architecture-based way. A survey describing these requirements in detail can be found in [11].

Among the modelling capabilities, we mention the ability to specify semantics and types for components and connectors — elements, and compositionality for configurations. The semantics of an element is a description of its behaviour in a formal language that enables tool-supported analysis. Then, types allow to define reusable blocks of semantics. Compositionality makes it possible to regard an architecture as a single element in another architectural specification.

An ADL should also include a language feature for the Architectural Pattern concept. The concept is largely applied in the software architecture community and it refers to an architectural abstraction of a design solution to recurring problems. Available catalogs, such as [6] and [15], propose architectural design and evolution by their application. However, its characterisations remain largely informal.

Among the features that tools must provide we mention multiple-views and analysis. The first refers to the possibility of both a textual and a graphical representation. For the second, we focus our attention in the possibility of two checks: refinement — whether an architecture materialising an architectural pattern is a refinement of an element in another pattern (the relevance of this verification is highlighted in [7]); and equivalence — whether an architecture is behaviourally equivalent to another. The mCRL2 language and tool set is a suitable platform to provide these features.

The mCRL2 language [10] is a specification language for describing communication behaviour among systems, which combines a process algebra [4], for the dynamic component, with higher-order abstract equational data types, for the data part. The language is supported by a toolset [9] enabling simulation, visualisation, behavioural reduction and verification. A number of analysis techniques are implemented; their execution resorting, in some cases, to human guidance.

As usual in process algebras, behaviours are described by means of process terms which compose elementary actions (events, communications) through a number of combinators. The reader is referred to [10] for a complete description of mCRL2 and its semantics. For the moment, we limit ourselves to recall the informal meaning of the main process combinators and operators used in the sequel.

Among the combinators of the language we mention: alternative composition $p + q$ — indicates that either of two behaviours p or q can occur; sequential composition $p.q$ — denotes that the process behaves as p and when p finishes it behaves as q; parallel composition $p \parallel q$ — indicates that the actions in p occur independently of actions in q; conditional $c \rightarrow p \diamond q$ — represents a process that behaves as p if c holds or as q if it does not; multiactions — denote a set of actions that occur simultaneously; and summand (shown below) — indicates a choice among the processes $p(c)$ for c ranging over the possible values in the datatype D.

$$\sum_{c:D} p(c)$$

Parallel processes can also be influenced by communication operators. We mention four of them: communication, allow, rename, and hide. The communication operator $\Gamma_c(p)$ is used to indicate which actions communicate and synchronise within a parallel process expression p. The set C contains communication rules of the form $a_1|...|a_n \rightarrow c$ with $n > 0$ and a_i and c actions. Each communication rule triggers the replacement of $a_1(d)|...|a_n(d)$ with $c(d)$ in p. The allow operator $\nabla_V(p)$ is used to enforce multi-actions in V and block undesired ones. The rename operator $\rho_R(p)$, with R a set of renames of the form $a \rightarrow b$, is used to rename action names in p. The hide operator $\tau_I(p)$ removes action names in the set I from multi-actions and replace them with the invisible action τ.

3. A LANGUAGE FOR ARCHITECTURAL SPECIFICATIONS

An architectural specification is built with two basic constructs: `pattern` and `architecture`. The first is a formalisation of the concept of architectural pattern and contains elements (connectors and components) showing some behaviour. The second is an instance of a pattern, and describes a particular configuration of (element or pattern) instances. A pattern can be specified without describing any particular architecture. However, in order to be verified, a description of the form $Desc$ needs to contain an architecture and a non empty set of patterns.

$$Desc = \langle Set(Pat), Arch \rangle, \text{ with } patterns = \pi_1 \qquad (1)$$

We use projection functions to access tuple components. A projection π_i allows us to retrieve the i^{th} tuple component. As a convention, each projection is given a name: the component type name in lower cases. For instance, in (1), π_2 is named $arch$. When convenient, we also explicitly name projections as we do in the case of π_1 as $patterns$.

3.1 Pattern

The `pattern` construct groups descriptions of components and connectors in a single unit, and adopts the form Pat shown in (2). The first component of a tuple pat of this type is a unique identifier and the second is a list that contains its formal parameters. Each fp in $fpars(pat)$ is a tuple with the two components shown in (3): an identifier and an mCRL2 data type. The example in Listing 1 describes a pattern with `ClientServer` as id and no parameters.

$$Pat = \langle Id, List(Fp), Set(E) \rangle, \text{ with } fpars = \pi_2 \wedge elements = \pi_3 \quad (2)$$

$$Fp = \langle Id, DataType \rangle \quad (3)$$

Listing 1: Client Server Pattern

```
1  pattern ClientServer()
2  elements
3  element Client()
4    act prcs, sreq, rres;
5    proc Client() = prcs.sreq.rres.Client();
6    interface
7      in rres;
8      out sreq;
9  element Server()
10   act rreq, sres, cres;
11   proc Server() = rreq.cres.sres.Server();
12   interface
13     in rreq;
14     out sres;
15 end
```

3.1.1 Element

The language construct `element`, allows us to define each element in $elements(pat)$. We use this construct to explicitly characterise both components and connectors. An element e is a tuple that includes, as shown in (4), six components: an identifier, a list of formal parameters $fpars(e)$; a set $actions(e)$ of actions; a tuple $mpd(e)$; a set $procs(e)$ of process declarations; and a set $ports(e)$ of ports (its interface). Each fp in $fpars(e)$ adopts the same form as in (3), and each act in $actions(e)$ the one in (5).

$$E = \langle Id,\ List(Fp),\ Set(Act),\ Mpd, Set(Prc),\ Set(Prt) \rangle, \text{ with } fpars = \pi_2 \wedge actions = \pi_3 \wedge procs = \pi_5 \wedge ports = \pi_6 \quad (4)$$

$$Act = \langle Id,\ List(Datatype) \rangle \text{ with } datatypes = \pi_2 \quad (5)$$

The element's behaviour is described using a slightly modified subset of mCRL2. In particular, no `init` section is considered and process expressions cannot include summands, parallel process combinators, nor communication operators. The content of $actions(e)$ are the actions declared in `act` sections, and the $mpd(e)$ and $procs(e)$ correspond with process declarations in `proc` sections. The example in Listing 1 shows for each element a section `act` declaring actions and a section `proc` defining a process that uses these actions.

The first declared process within the element is the *main process declaration* (MPD). It constitutes the entry point for its behaviour, and is the one to be started when the element is instantiated. We defer an explanation of what do we mean with "instantiation" until section 4.1, where we describe how instance behaviours are generated. For now, we limit to mention that initial values must be specified for all its formal parameters. The MPD form, shown in (6), has as components: an identifier; a list of formal parameters $fpars(mpd)$; a list $ivalues(mpd)$ of data expressions that match in order and type $fpars(mpd)$, and constitute their initial values; and a process expression $body(mpd)$. The data expressions in $ivalues(mpd)$ are defined with constants and identifiers declared in $fpars(e)$.

$$Mpd = \langle Id,\ List(Fp),\ List(Iv),\ Body \rangle, \text{ with } fpars = \pi_2 \wedge ivalues = \pi_3 \quad (6)$$

Note that the declaration of initial values in formal parameters is not part of mCRL2. Also that sections `const`, `sort`, `var`, `eqn`, and `map` can be part of an specification but we do not include tuple components in e for them.

Each $proc$ in $procs(e)$ is a process declaration that may be invoked from the main process body $body(mpd(e))$. Their tuple components are of the form shown in (7), similar to Mpd, with the difference that they do not have a list of initial values for their formal parameters.

$$Prc = \langle Id, List(Fp), Body \rangle, \text{ with } fpars = \pi_2 \quad (7)$$

3.1.2 Interface

The `interface` keyword marks the declaration of the set of ports $ports(e)$ that constitute the element's interface (see interfaces for Client and Server respectively starting at lines 6 and 12). We adopt the underlying metaphor of water flow in [2] 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. Each port constitutes a tuple $port$ of the form shown in (8) with a direction (either `in` or `out`), and an action as components.

$$Prt = \langle Direction, Act \rangle \quad (8)$$

3.2 Architecture

An `architecture` construct describes a configuration that instances adopt. Its declaration is a tuple $arch$ of the form shown in (9). Its tuple components are: an identifier; a pattern reference that must match a declared pattern, i.e., an id of a tuple pat in $patterns(desc)$; a list $apars(arch)$ of actual parameters that match in order and type $fpars(pat)$; a set of instances $instances(arch)$, a set of attachments $atts(arch)$, and a set of port renames $rens(arch)$. An example architecture is shown below line 15 of Listing 2.

$$Arch = \langle Id, Id, List(Ap), Set(Inst), Set(Att), Set(Ren) \rangle, \text{ with } idPat = \pi_2 \wedge apars = \pi_3 \wedge instances = \pi_4 \wedge atts = \pi_5 \wedge rens = \pi_6 \quad (9)$$

Listing 2: Pipe and Filter Architecture

```
1  pattern PipeFilter()
2  elements
3  element Filter()
4    act rec, trans, send;
5    proc Filter() = rec.trans.send.Filter();
6    interface
7      in rec;
8      out send;
9  element Pipe()
10     act accept, forward;
11     proc Pipe() = accept.forward.Pipe();
12   interface
13     in accept;
14     out forward;
```

```
15 end
16 architecture clientServer = ClientServer()
17 instances
18   architecture s = PipeFilter()
19     instances
20       f1 = Filter(); f2 = Filter();
21       p1 = Pipe();
22     attachments
23       from f1.send to p1.accept;
24       from p1.forward to f2.rec;
25     interface
26       f1.rec as rreq;
27       f2.send as sres;
28   end
29   c1 = Client();
30   c2 = Client();
31 attachments
32   from c1.sreq to s.rreq;
33   from c2.sreq to s.rreq;
34   from s.sres to c1.rres;
35   from s.sres to c2.rres;
36 end
```

3.2.1 Instances

After the keyword `instances` the set $instances(arch)$ is defined. An instance $inst$ in this set can be either an element instance or an architecture. In the former case, see (10), the instance has an identifier, a reference to an element $idElem(inst)$, and a list $apars(inst)$ of actual parameters that coincide in order and type the element's formal ones $fpars(e)$. In the later case, a new `architecture` construct can be nested as an instance entry. This allows instances to have an architecture as well, building grounds for hierarchically composing specifications. As an example, between lines 18 and 28, an instance is specified as an architecture.

$$\begin{aligned} Inst =& Ainst \mid Einst \\ Ainst =& Arch \\ Einst =& \langle Id, Id, List(Ap) \rangle \\ & \text{with } (idElem = \pi_2 \wedge apars = \pi_3) \end{aligned} \tag{10}$$

3.2.2 Attachments

The description of how instances are connected begins after the keyword `attachments`. Each line in the section defines an attachment, a tuple att in $atts(arch)$ of the form shown in (11). Each attachment includes two port references: one is the out port $from(att)$, and the other the in port $to(att)$. They are respectively indicated with the `from` and `to` keywords. For instance, the attachment in line 23 indicates that the out port `rec` of `f1` communicates with the in port `accept` of `p1`.

$$Att = \langle Pr, Pr \rangle, \text{ with } from = \pi_1 \wedge to = \pi_2 \tag{11}$$

3.2.3 Interface

The architecture's interface is the set $rens(arch)$ of port renames declared upon the keyword `interface`. Each rename ren contains a port reference $pr(ren)$ and a new name $id(ren)$. Ports not included in $rens(arch)$ are not visible from the outside. Note that including the same port in an attachment and in the interface is incorrect. An example rename is shown in line 26. It indicates that the port `rec` of `f1` becomes the port `rreq` for the architecture `s`.

Finally, a port reference (used in (11) and (12)) is a tuple pr adopting one of the two forms shown in (13). The first is used when the reference is to an element instances, and the second when it is to a pattern instances.

$$Ren = \langle Id, Pr \rangle \tag{12}$$

$$Pr = \langle Inst, Prt \rangle \mid \langle Inst, Ren \rangle \tag{13}$$

4. ARCHITECTURAL ANALYSIS

4.1 Generation of mCRL2 Specifications

In this section we briefly describe how a process specification is automatically generated from an architectural one. A detailed description can be found in [14].

A generated specification has three main mCRL2 sections: `act`, `proc` and `init`. We do not mention other sections concerning data expressions since they do not require processing and are just copied. Each of the concatenated functions below formats one of the three sections from a parameter.

$$\begin{aligned} & gen(ainst : Ainst) \triangleq \\ & \quad wrtActs(acts) +\!\!+ wrtPds(pds) +\!\!+ wrtInit(conf) \end{aligned} \tag{14}$$

The first function $wrtActs$ writes the declaration of all actions from a set $acts$ of tuples of the form Act. The second writes the process declarations from a string pds. The third does a similar thing for the initialisation of the system.

We generate the parameters for these functions from an architectural declaration $ainst$ by applying the recursive function $genPA$. The initial call to this function receives $ainst$ as parameter. Then, it is recursively called for each instance i of $ainst$ and a result r_i is obtained. Therefore, we need two versions, one for architectures (15), and one for element instances (16). In both cases the result of its evaluation is a four-component tuple: $acts$ — is the set of actions to be placed in the `act` section; pds — is a string containing all process declarations and is to be placed in the `proc` section; $conf$ — is a string with the configuration the system adopts and is to be placed in the `init` section; and $allows$ — is a set of strings representing action ids and is used for intermediate calculations during recursive calls. In the next subsections we describe how each of these values is calculated for element instances first and then for pattern instances.

$$\begin{aligned} & genPA(ainst : Ainst) \triangleq \langle acts, pds, conf, allows \rangle \\ & \quad \forall\, i \in instances(ainst),\ r_i = genPA(i, ainst) \end{aligned} \tag{15}$$

$$genPA(einst : Einst) \triangleq \langle acts, pds, conf, allows \rangle \tag{16}$$

4.1.1 Generation — Element instance

The generation of each instance $einst$ of an element e is calculated from the results of processing the bodies of all process declarations defined in e with respect to $einst$. The function $genStep$, invoked within $genPA$ for each process body p, carries out this processing and returns a tuple containing: $acts_p$ — the set of actions to be placed in the `act` section; pds_p — a string with the resulting process expression; $allows_p$ — a set of strings used for intermediate calculations; and $hides_p$ — a set of strings used to build the $conf$ string for the element instance.

We need three rules for $genStep$. The first is for the alternative composition (17) of two process expressions p and q.

The result is the combination calculated by *combineAC* of the tuples returned from the recursive call with p and q. The function *combineAC* returns a tuple with the concatenation of the respective pds_p, re-assembling the alternative composition expression, and the union of the respective $allows_p$, $acts_p$ and $hides_p$ sets. The second *genStep* is for conditionals (18). It works in a similar way as the one for alternative composition with the difference that it reassembles a conditional. The third one is for sequential composition (19) and uses another function (20) that processes the individual id.

$$genStep(\text{"\$}p\text{\$ + \$}q\text{\$"}, einst : Einst) \quad (17)$$
$$\triangleq combineAC(genStep(p, einst), genStep(q, einst))$$
$$genStep(\text{"\$}c\text{\$ } \rightarrow \text{ \$}p\text{\$ } \diamond \text{ \$}q\text{\$"}, einst : Einst) \quad (18)$$
$$\triangleq combineCond(c, genStep(p, einst), genStep(q, einst))$$
$$genStep(\text{"\$}id\text{\$(\$}x\text{\$).\$}p\text{\$"}, \ einst : Einst) \quad (19)$$
$$\triangleq combineSC(genStep(id, x, einst), genStep(p, einst))$$
$$genStep(i : Id, x : List(Ap), einst : Einst) \quad (20)$$
$$\triangleq \langle acts_p, pds_p, allows_p, hides_p \rangle$$

The *genStep* for processing an *id* requires distinguishing which of the three roles shown below is *id* assuming.

- *Process Call.* In this case pds_p becomes the concatenation of the *id* (the process name) with $id(einst)$ and with its actual parameters if present, and $allows_p$, $hides_p$, $acts_P$ are empty sets.

- *Port.* In this case we first need to obtain the list of attached port references $attPrts$. If the port is locally attached, i.e. at the container architecture *ainst*, we get it from the set of attachments $atts(ainst)$. If the port is attached at the upper level, and assuming that *gparent* is the architecture containing *ainst*, we obtain the set from $atts(gparent)$, after considering the corresponding rename in $rens(ainst)$ (this may need to be repeated). Then we generate the set of actions $acts_p$ by creating a new action for each *pr* in $attPrts$. The identifier for an action derived from a *pr* is the concatenation of *id* with $id(einst)$ and $id(inst(pr))$. Since ports behave differently according their direction, we need to generate the process expression pds_p according the cases as follows:

  - *In.* The process expression is the alternative composition of all actions in $acts_p$. When an action receives parameters, we also need to surround the expression with a summand as the one shown below for each parameter d of datatype D not bound to any value.

    $$\sum_{d:D} pds_p$$

  - *Out.* The process expression is the alternative composition of the multiactions generated from the actions in each set of the power set $P(acts_p)$ (the empty set must be removed first). An example of the generated expression is shown in Listing 3, for the port in line 27 of Listing 2.

  The $allows_p$ is the set of the action ids taken from $acts_p$, and $hides_p$ is the empty set.

- *Internal Action or Unattached Port.* Then pds_p is an action named as the concatenation of *id* with $id(einst)$. The $acts_p$ is a set with this action, and $allows_p$ and $hides_p$ are sets containing only the generated id.

Listing 3: Out Port Example

```
1 send_f2_c1 + send_f2_c2 +
2 send_f2_c2 | send_f2_c1
```

Note that the tactic for generating identifiers is a naive solution in the sense that it ensures different ones when it may not be necessary. An optimised solution aiming at the minimisation of the LTS state number is part of future work.

Once we obtained a result (containing $acts_p$, pds_p, $allows_p$, and $hides_p$) from each process body, we combine them. With the exception of the body corresponding to the main process, all string expressions pds_p are concatenated into *pds* (previously adding their process headers). The sets ($allows_p$, $acts_p$ and $hides_p$) are position-wise combined by applying set union to become the sets *allows*, *acts* and *hides*.

Subsequently, we generate the *conf* string for the element instance. For this, we concatenate the element name $id(e)$, with the instance name $id(einst)$, and the actual parameters $apars(einst)$. For example, the *conf* generated from the line 29 in Listing 2 is *Client_c*1. In the case that the element instance has actual parameters, they are appended after the instance id; e.g. $c1 = Client(3)$ becomes *Client_c*1(3). In the case that we want to carry out behavioural comparisons and the set *hides* is not empty, we generate another expression as shown in (21).

$$conf_h = \tau_{hides} conf \quad (21)$$

We still need to concatenate to the *pds* the MPD $mpd(e)$. In order to use the initialisation values $ivalues(mpd(e))$, two process declarations are required. The example MPD in Listing 4 and the generated code in Listing 5 illustrate how it is done. Subsequently, the process declaration for the MPD can be concatenated to *pds*.

Listing 4: Example MPD

```
1 element Client(idC:Nat)
2 proc
3   Client(id:Nat = idC) = ...
```

Listing 5: Example MPD - Generated mCRL2

```
1 Client_c1(idC:Nat) = Client_mp_c1(idC);
2 Client_mp_c1(id:Nat)  = ...
```

4.1.2 Generation — Pattern Instance

The result of evaluating *genPA* with an architecture *ainst* is a tuple r of the form $\langle acts, pds, conf, allows \rangle$ as shown in (15). The tuple components in r are generated upon the set of results r_i of recursively applying *genPA* with each instance i in $instances(ainst)$.

The tuple component *conf* in the result is a string with the mCRL2 operators ρ, ∇, and Γ, nested in that order. The communication operator Γ takes C as the set of communications and the parallel combination of the π_3 of each r_i (which is the respective $conf_i$ of each instance i). The allow operator ∇, taking as parameter the set V of action ids, is used

to enforce communications and to rule out undesired action combinations. The rename operator ρ is applied then with the set R of renamings. The $conf_h$ string is calculated as the hiding operator τ applied to $conf$ taking H as argument. As we will mention in Section 4.2, it is used when specifications are generated to carry out behavioural comparisons.

$$conf = \rho_R(\nabla_V(\Gamma_C(\prod_{ei \in insts(ainst)} \pi_3(r_i)))) \quad (22)$$

$$conf_h = \tau_H(conf) \quad (23)$$

Each rule in the set C of communication rules is generated from each attachment in $atts(ainst)$. Identifiers are concatenated as it is shown in example (24), which is the generated rule for the attachment in line 32 of Listing 2.

$$cr = \langle sreq\_c1\_s,\ rres\_s\_c1,\ sreq\_c1\_rres\_s \rangle \quad (24)$$

The calculation of the set R of rename rules requires more work. For each ren in $rens(ainst)$ we need the set $attPrts$ of attached port references in the upper level. Then, for each prt in $attPrts$ we generate a rename rule rr as shown in (25). An example of the generated rename rules for the lines 26 and 27 in Listing 2 is shown in Listing 6

$$rr = \langle id_{old}, id_{new} \rangle \quad (25)$$
$$id_{old} = id(pr(ren)) +\!\!+ id(inst(pr(ren))) +\!\!+ id(inst(prt))$$
$$id_{new} = id(ren) +\!\!+ id(inst(prt)) +\!\!+ id(inst(pr(ren)))$$

Listing 6: Rename Rules - Generated mCRL2

```
rename({
  send_f2_c1 -> sres_s_c1,
  send_f2_c2 -> sres_s_c2,
  rec_f1_c1 -> rreq_s_c1,
  rec_f1_c2 -> rreq_s_c2 }, ...)
```

The set V for the allow operator is calculated as the union of three sets: the set of action ids resulting from the projection $\pi_3(cr)$ for each synchronisation rule cr in C; the set of action ids resulting from the projection $\pi_1(rr)$ of each rename rule in R; the set resulting from the union of each set $\pi_4(r_i)$, i.e., the $allows_i$ set obtained from each instance i of the architecture.

The set H for the hiding operator contains all ids of synchronisation actions in C, i.e. $\{sid \mid sid = \pi_3(cr)\ \forall cr \in C\}$.

The last tuple component of r, $acts$, is the union of the three sets as follows: the set of actions resulting from the projection $\pi_3(cr)$ of each cr in C; the set of actions resulting from the projection $\pi_1(rr)$ of each rr in R; and the set resulting from the union of each set $\pi_1(r_i)$ of each r_i.

4.2 Analysis of Architectural Specifications

Once translated to mCRL2, an architectural specification can be analysed and compared resorting to tools available in the corresponding toolsuit. The envisaged analysis is based on the following definition of architectural equivalence:

$$a \equiv b \Leftrightarrow gen_h(a) \approx_B gen_h(b) \quad (26)$$

where gen_h refers to the same as gen in (14) but with each occurrence of $conf$ replaced by $conf_h$,i.e., the set of actions not involved in the interface are hidden; and $\approx_B$ denotes branching bisimilarity. Branching bisimilarity [4] relates behaviours differing in the amount of internal activity but exhibiting similar branching structure. This equivalence relation allows us to determine whether two architectures are interchangeable with respect to their interface behaviour.

A weaker form of comparing architectural specifications resorts to

$$a \sqsubseteq b \Leftrightarrow gen_h(a) \sqsubseteq_W gen_h(b) \quad (27)$$

where $\sqsubseteq_W$ is weak trace inclusion. A weak trace is a sequence of observable actions performed by a behaviour. This relation allows us to answer if an architecture b has an interface language allowing it to replace an architecture a.

Both $\approx_B$ and $\sqsubseteq_W$ are supported by the mCRL2 toolset. In the sequel we illustrate the sort of envisaged analysis that can be carried out in our framework.

In order to start our comparisons we define a *base* architecture. We use the one in Listing 2 with the nested architecture `s` replaced with `s = Server()` an instance of the Server element defined in Listing 1.

Now suppose we want to charge the client each time the server is used. We define a new element `PaidServer` that after computing a response, calculates a cost for the service (Listing 7). Then we define the architecture *paid* by replacing `s` with an instance of `PaidServer`, and we use the tool to check that both $base \sqsubseteq paid$ and $paid \equiv base$.

Listing 7: Paid Server

```
element PaidServer()
  act rreq, sres, cres, ccost;
  proc Server() =
    rreq.cres.ccost.sres.Server();
  interface
    in rreq;
    out sres;
```

A new version of the server, represented by the element `BuggyServer`, is available. We define an architecture *buggy* by replacing `s` with an instance of `BuggyServer`. Then we use the tool to establish that $base \sqsubseteq buggy$ but $buggy \not\equiv base$. The second result is because while *base* responds to every request, *buggy* stops working in a non-deterministic way.

Listing 8: Buggy Server

```
element BuggyServer()
  act rreq, sres, cres;
  proc Server() =
    rreq.(cres.delta + cres.sres.Server());
  interface
      in rreq;
      out sres;
```

Now we want to compare *base* with an architecture in which the server does not perform any processing but can receive more than one request before responding. We name this architecture $buffered$, and we obtain it by replacing `s` in *base* with the `s` in Listing 9. Now we verify that $base \sqsubseteq buffered$ but $buffered \not\equiv base$. This is because `s` in *base* has to respond to any received request before receiving a subsequent one, but the `s` in $buffered$ can receive up to 3 requests without responding to any.

Listing 9: Server with a 3-Position-Buffer

```
pattern Buffered()
elements
element Buffer()
  act inb, outb;
  proc Buffer() = inb.outb.Buffer();
  interface
    in inb;
    out outb;
end
architecture s = Buffered()
  instances
    b1 = Buffer(); b2 = Buffer();
    b3 = Buffer();
  attachments
    from b1.outb to b2.inb;
    from b2.outb to b3.inb;
  interface
    b1.inb as rreq;
    b3.outb as sres;
end
```

We define a different version of the server as a materialisation of the Pipe and Filter Pattern. It is the one shown in Listing 2 and we refer to such architecture as pf. Using the tool we are able to establish that $base \sqsubseteq buffered \sqsubseteq pf$ and that $pf \equiv buffered \not\equiv base$. This means that the server in pf can replace the server in $base$, and if we assume that the server in $buffered$ is behaviourally correct, we can safely assume that the server in pf also is.

We can add to the system the ability to acknowledge every communication. The architecture shown in Listing 10, named $ackBase$, reflects this behaviour. The server receives a request, computes a response, sends the response to the client, waits for an acknowledge indicating that the response was received, and then sends an acknowledge of receiving a request. As expected, $base \not\equiv ackBase$ and $base \not\sqsubseteq ackBase$.

Listing 10: Server — Acknowledge

```
pattern AckClientServer()
elements
element Server()
  act rreq, arreq, sres, asres, cres;
  proc Server() =
    rreq.cres.sres.asres.arreq.Server();
  interface
  in rreq; out arreq;
  out sres; in asres;
element Client()
  act prcs, sreq, asreq, rres, arres;
  proc Client() =
    prcs.sreq.rres.arres.asreq.Client();
  interface
    out sreq; in asreq;
    in rres; out arres;
end
architecture ackBase = AckClientServer()
instances
  s = Server();
  c1 = Client(); c2 = Client();
attachments
  from c1.sreq to s.rreq;
  from s.arreq to c1.asreq;
  from c2.sreq to s.rreq;
  from s.arreq to c2.asreq;
  from s.sres to c1.rres;
  from c1.arres to s.asres;
  from s.sres to c2.rres;
  from c2.arres to s.asres;
end
```

A further improvement is to replace s in $ackBase$ with a server defined in terms of the Pipes and Filters pattern in a similar way as done in pf. We call such configuration $ackPf$ and we show in Listing 11 the elements to create it. We expect $ackBase \sqsubseteq ackPF$ and $ackBase \not\equiv ackPF$. The later because we assume that, in a similar way as with pf, $ackPF$ can receive up to three requests before answering any. However, the result we obtain is $ackBase \equiv ackPF$. The reason lies in the order in which events take place. In the particular case of Filter (see line 6 in Listing 11) the reception of a request, represented by rec, is only acknowledged at the end of the process by $arec$. Since this is also happening in Pipe and in Client (the initial communication is acknowledged last, just before the recursive call), the first filter in the chain is not free to receive a second request until the client that issued the request receives his response. As a consequence, the emergent behaviour turns out to be similar to the one shown by $ackBase$. This can be easily resolved by acknowledging a communication as soon as it occurs.

Listing 11: Pipes and Filters — Acknowledge

```
pattern AckPipeFilter()
elements
element Filter()
  act rec, arec, trans, send, asend;
  proc Filter() =
    rec.trans.send.asend.arec.Filter();
  interface
    in rec; out arec;
    out send; in asend;
element Pipe()
  act accept, forward, aaccept, aforward;
  proc Pipe() =
    accept.forward.aforward.aaccept.Pipe();
  interface
    in accept; out aaccept;
    out forward; in aforward;
end
```

5. CONCLUSIONS

5.1 Related Work

We can distinguish two approaches in the design of languages that provide support for both the behavioural and structural dimensions in architectural design. One is to extend a structure-based language with a behavioural model, and the other is to build the architectural language on top of the behavioural model, by upgrading it with architectural constructs.

The Architectural and Analysis Description Language — AADL [13], widely adopted in the aeronautic and automotive industry, is an example of the first approach. AADL was mostly focused on the structural dimension but a behavioural annex [5] was included later. Subsequently, the annex was precisely defined under different behavioural models and the corresponding tools were developed.

The work in [1] is an example of the second approach. The authors propose a set of guidelines to transform a process algebra into a fully fledged architectural description language. They refer to the resulting language as a Process Algebraic Architecture Description Language — PADL. The guidelines describe how to include component-orientation into a

process algebra and in this way allow precise and analysable architectural descriptions. They also argue that this approach enhances usability of process algebras.

AADL, PADL and the language we propose in this paper have language constructs to specify components and configurations as first-class-citizens. In both PADL and the later it is also possible to explicitly declare connectors. Moreover, the same language construct is used for this.

Both PADL and AADL provide a variety of ports to indicate different sorts of interactions. In contrast, we only consider in and out ports, and resorts to the water-flow metaphor, to informally indicate their semantics.

The three languages allow the specification of types. Since AADL does not provide a language construct for connectors, it also lacks a mean to declare connector types.

The three languages support patterns as a mechanism to develop specifications from existing ones. PADL and our own language differ from AADL in that they particularly aim at formalising the concept of Architectural Pattern [6][15]. PADL provides a mechanism to restrict possible configurations that a system specified as an instance of a pattern may adopt. It basically limits how configurations can be built.

The three languages have specific tool support. Our language inherits the existing mCRL2 toolset, PADL is supported by a tool named Two-Towers, and AADL has an extensive tool support covering a wide range of needs.

5.2 Conclusions and future work

In this paper we presented an ADL that allows the behavioural modelling of architectural patterns and the compositional construction of architectures. We describe a translator from architectural specifications to mCRL2, enabling in this way, tool-supported analysis of the formers. The analysis considers two envisaged relations: equivalence and refinement; which are illustrated with examples. As future work we consider the development of a refinement calculus for architectural patterns, exploring relations $\equiv$ and $\sqsubseteq$ respectively defined in (26) and (27). We also expect to improve the translator in order to obtain process algebra specifications better suited for tool analysis.

Acknowledgements

This research was partially supported by Fct (the Portuguese Foundation for Science and Technology) under contract PTDC-/EIA-CCO/108302/2008 — the Mondrian project, and by QREN (the Portuguese National Strategy Reference Chart) project 1621 — Evolve.

6. REFERENCES

[1] A. Aldini, M. Bernardo, and F. Corradini. *A Process Algebraic Approach to Software Architecture Design*, volume 54. Springer London, London, 2010.

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

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

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

[5] J. P. Bodeveix, M. Filali, P. Gaufillet, and F. Vernadat. The AADL real-time model A behavioural annex for the AADL. In *Proc. of the DASIA 2006 – DAta Systems In Aerospace – Conference*, number May, Berlin, Germany, 2006.

[6] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Wiley, 1996.

[7] D. Garlan. Style-based refinement for software architecture. *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops* -, pages 72–75, 1996.

[8] D. Garlan and M. Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering (volume I)*. World Scientific Publishing Co., 1993.

[9] 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.

[10] 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.

[11] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering, IEEE*, 26(1):70–93, 2002.

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

[13] SAE Development Team. AADL homepage, 2011.

[14] A. Sanchez, L. S. Barbosa, and D. Riesco. A Language for Behavioral Modeling of Architectural Patterns - Generation and Analysis in the mCRL2 Framework. Technical Report DI-CCTC-11-02, Universidade do Minho, Departamento de Informática, Campus de Gualtar 4710-057, Braga, Portugal, 03 2011.

[15] M. Shaw and D. Garlan. *Software architecture: perspectives on an emerging discipline*. Prentice Hall, 1996.