Architectural Specifications
for the Development
of CommUnity Programs

Artur Zawłocki

with

Antónia Lopes
José Luis Fiadeiro
Andrzej Tarlecki

Last modification: 9 September 2003
Overview

• What are (CASL-style) architectural specifications good for?
• Semantics of parametric units
• Architectural specification in CommUnity: a (longish) example
• Conclusions and future work
spec Numbers =

sort Nat

ops 0 : Nat;

succ : Nat → Nat;

-- + -- : Nat × Nat → Nat;

fibb : Nat → Nat;

gcd : Nat × Nat → Nat

axioms ∀n : Nat ∙ n + 0 = n;

∀n, m : Nat ∙ n + succ(m) = succ(n + m);

...
What are (CASL-style) Architectural Specifications for?

Assume one is given a specification to implement. The one on the last slide is not very impressive, but for the sake of example let it represent large specification of a complex system.

Probably it is not feasible to implement such a big specification in one go. One would rather break the task of implementing it into smaller, independent and more tractable subtasks. Then this subtasks could be given to different implementors and the modules they produce would be then put together to form complete implementation.

One may also hope to reuse certain modules.
spec Nat =
  sort  Nat
  ops   0 : Nat, succ : Nat → Nat
       _ + _ : Nat × Nat → Nat;
  axioms ...
end

arch spec ArchNumbers =
  units  N : Nat
         F : Nat → spec Nat  then op fibb : Nat → Nat...
         G : Nat → spec Nat  then op gcd : Nat × Nat → Nat...
  result F(N) and G(N)
*ArchNumbers* is an architectural specification. It says, that in order to implement it one must provide three units (think of them as of modules in some programming language) named $N$, $F$ and $G$. Each unit has a specification associated with it: $N$ satisfies $Nat$ whereas $F$ satisfies

$$Nat \rightarrow \text{spec} \ Nat \ \text{then op} \ fibb : Nat \rightarrow Nat \ldots$$

(… stand for the axioms of $fibb$) which means, that when applied to some unit satisfying $Nat$, $F$ will give a unit satisfying the specification in the target of $\rightarrow$.

Thus $F$ and $G$ are parametric units (think of them as of functors in some expressive enough programming language) and one may apply them to $N$. They both independently extend $N$ with two different functions. The last line:

\[ \text{result } F(N) \text{ and } G(N) \]

says that the final implementation is obtained by taking the amalgamation of the two extensions.
But what is $F : SP \rightarrow SP'$?
But what is $F : SP \to SP'$?

$F : SP \sigma \rightarrow SP'$ iff

- $F : |\text{Mod}(\Sigma)| \rightarrow |\text{Mod}(\Sigma')|$, 
- $F(M) \in \text{Mod}[SP']$ for all $M \in \text{Mod}[SP]$, and 
- $F(M)|_{\sigma} = M$ for all $M \in \text{Dom}(F)$

where $\Sigma = Sig[SP], \Sigma' = Sig[SP']$.

Why do we need these requirements?
Why do we need these requirements?

First two are just to ensure that (the semantics of) \( F \) is a (possibly partial) function taking every model \( M \) of \( SP \) to a model \( F(M) \) of \( SP' \).

The third one expresses the property of persistency — the argument model may still be retrieved from the result of the operation. That is, \( F \) preserves its argument and possibly extends it.
Semantics of parametrised units

\[
\text{arch spec } AS = \\
\text{units } A : SP_G \\
F : SP \overset{\sigma}{\rightarrow} SP' \\
A' = F(A \text{ fit } \gamma : \Sigma \rightarrow \Sigma_G) \\
\ldots
\]
In general one may want to apply the parametric unit accepting arguments satisfying \( SP \) to
a unit satisfying some other specification \( SP_G \). This is possible provided that there exists a
fitting morphism \( \gamma : \Sigma \rightarrow \Sigma_G \) that identifies components of \( \Sigma_G \) that correspond to compo-
nents of \( \Sigma \).

For every model \( A \) over \( \Sigma_G \), its reduct \( A|_\gamma \) along \( \gamma \) is a model over \( \Sigma \). If \( \gamma \) is a specification
morphism from \( SP \) to \( SP_G \) then the reduct \( A|_\gamma \) of a model \( A \) of \( SP_G \) is a model of \( SP \).
Semantics of parametrised units

\[ \text{arch spec } AS = \]
\[ \text{units } A : SP_G \]
\[ F : SP \xrightarrow{\sigma} SP' \]
\[ A' = F(A \text{ fit } \gamma : \Sigma \to \Sigma_G) \]
\[ \ldots \]

To define \( F_G \) we need:

- the existence of pushouts in \( \text{Sig} \),
- the persistency of \( F \) and
- the amalgamation property
Since $F$ is persistent, $F(A|\gamma)$ reduces to $A|\gamma$. Using the amalgamation property one obtains a unique model $F_G(A)$ over $\Sigma'_G$, that extends both $A$ and $F(A|\gamma)$.

This construction defines the semantics of $F : SP \xrightarrow{\sigma} SP'$ as a function from $\Sigma_G$ to $\Sigma'_G$.

Verifying the correctness of this function requires checking that $\sigma$ and $\gamma$ are specification morphisms. If so, the diagram of signatures may be replaced by a diagram of corresponding specifications and the resulting model $F_G(A)$ will satisfy the pushout specification $SP'_G = SP_G \cup SP'$.
In CommUnity we have **Architectural Connectors**:

*(Taken from J. L. Fiadeiro, A. Lopes, M. Wermelinger A Mathematical Semantics for Architectural Connectors)*

Do we need another way of describing architectures?
A semantics of CommUnity architectural connector is a function on CommUnity designs: its arguments are the refinements of connector’s roles, its result is the refinement of the colimit of the connector.

When refinements of a design are viewed as models of a specification an analogy between connectors and parametric units emerges.
Architectural specifications for CommUnity

- **Designs** will be used as specifications.

```plaintext
design Counter is
    out val: Nat
    do incr [val]: True, val = 0 --> val' > val
       reset [val]: True, False  --> val' = 0
```
• **Designs** will be used as specifications.

```plaintext
design Counter is
    out val: Nat
    do incr [val]: True, val = 0 --> val' > val
        reset [val]: True, False --> val' = 0
```

• **Programs** are fully specified designs.

```plaintext
program P is
    in  a: Nat
    out x: Nat
    do incr_1: True --> x := x + 1
        incr_a: a > 0 --> x := x + a
        reset: a = 7 --> x := 0
```
Architectural specifications for CommUnity

- **Designs** will be used as specifications.

```plaintext
design Counter is
    out val: Nat
    do incr [val]: True, val = 0 --> val' > val
        reset [val]: True, False --> val' = 0
```

- **Programs** are fully specified designs.

```plaintext
program P is
    in a: Nat
    out x: Nat
    do incr_1: True --> x := x + 1
        incr_a: a > 0 --> x := x + a
        reset: a = 7 --> x := 0
```

- **Refinement morphisms** describe removing of underspecification.

\[(P, \rho)\] is a **model** of \(D\) iff \(\rho : D \rightarrow P\) is a refinement morphism
To make the analogy clear: CommUnity designs play the role of the specifications. A design describes a class of programs (fully specified designs), just like a specification describes a class of models.

Since there may be many ways a program refines a design, a model is given by a program together with a refinement morphism.

Refinement morphisms between designs are specification morphisms: for any model \((P, \mu)\) of a design \(D'\), its reduct \((P, \rho; \mu)\) along a refinement morphism \(\rho : D \rightarrow D'\) is a model of \(D\). Note, that the reduct operation does not change the program itself.
An example

design VendingMachine is

    out cr: Nat,
    dr : Drink,

    do increase_credit [cr]: True, False --> cr’ > cr
        choose_coffee [cr,dr]: cr > price(Coffee), False
            --> dr’ = Coffee \ cr’ <= cr - price(Coffee)
        choose_tea [cr,dr]: cr > price(Tea), False
            --> dr’ = Tea \ cr’ <= cr - price(Tea)
        give_drink [dr]: dr /= Nothing --> dr’ = Nothing
VendingMachine is a specification of a drink-selling machine.

The attribute \( cr \) records the amount of money the user has put into machine before choosing a drink, \( \text{increase\_credit} \) is used to increase this amount. Guard of this action is fully unspecified.

\( \text{choose\_coffee} \) and \( \text{choose\_tea} \) are issued to require a drink, they set the attribute \( dr \) and decrease the credit. Their guards are required to imply that the credit is larger than the price of the drink.

Finally \( \text{give\_drink} \) represents an action of giving a drink to the user. The guard of this action is fully specified, as well as its result on the attribute \( dr \).
An example

arch spec VendingMachine is
  prog A : Automaton
  func C : CountArg -> CountRes
  prog CA = C(A) : CountingAutomaton
  func D : DrinkArg -> DrinkRes
  func DA = D(CA) : DrinkingAutomaton
  func K : CoffeeArg -> CoffeeRes
  func T : TeaArg -> TeaRes
  prog KA = K(DA) : CoffeeAutomaton
  prog TA = T(DA) : TeaAutomaton
  prog KTA = KA + TA : VendingMachine
result KTA
The implementation of `VendingMachine` may be described by an architectural specification showing how to construct a program being a refinement of `VendingMachine`, given a program `A` satisfying some simpler specification `Automaton` and four “parametric units”: `C`, `D`, `K` and `T`.

Note, that in case of parametric units applications, the actual arguments does not satisfy the formal arguments specification, so the fitting morphisms (not shown on the previous slide) are needed to perform the application.
design Automaton is
  out st : Ready | Busy
  do insert []: True, st = Ready --> True
     choose [st]: st = Ready, False --> True
     give [st]: st = Busy --> st' = Ready

program A is
  out s : Ready | Busy
     b : Bool
  do in: s = Ready --> b := True
     ch: s = Ready --> if b then s := Busy else skip
     gv: s = Busy --> s := Ready || b := False

\( (A, \{in \mapsto insert, ch \mapsto choose, gv \mapsto give\}, \{st \mapsto s\}) \) is a model of Automaton
The design *Automaton* describes a two-state automaton.

Action *insert* must be enabled whenever the automaton is *Ready*, but in some implementations it may also be enabled when the automaton is *Busy*. *insert* may not change the *st* attribute. Action *choose* may only be enabled when the automaton is *Ready*, its effect on *st* is unspecified. Both the guard and the effect of *give* are fully specified.

Program *A* is an example implementation of *Automaton* that will be used through the example to construct the implementation of *VendingMachine*. $\rho$ will denote a refinement morphism from *Automaton* to *A*. 
design CountArg is
  do incr []: True, False --> True
  zero []: True, False --> True

design CountRes is
  out cr
  do incr [cr]: True, False --> cr’ > cr
  zero [cr]: cr > 0, False --> cr’ = 0

superposing functor C (P : CountArg) : CountRes is
program
  in n : Nat+
  out cr : Nat
  incr : P.G(incr) --> P.F(incr) || cr := cr + n
  zero : P.G(reset) \ cr > 0 --> P.F(reset) || cr := 0
Parametric unit $C$ is used to augment the basic automaton with the capabilities of increasing and resetting the amount of credit.

This extension (written in an ad-hoc notation as a body of functor $C$) amounts to adding one input and one output attribute, adding assignments to the effects of two actions and strengthening the guard of one of them — resulting program is thus a superposition of the argument program and the body of a functor defines superposition morphism between the argument design and its extension:

$$\sigma_C : \text{CountArg} \to C(\text{CountArg})$$

It is well known that such a superposition may be implemented by an architectural connector (and conversely — an application of a connector computes a superposition of an argument).

The connector implementing parametric unit $C : D \to D'$ will have the single role $D$ and the glue $C(D)$ refining $D'$.
An example — semantics of parametric units

program C(A) is
    out s : Ready | Busy
    b : Bool
    n : Nat+
    c : Nat
    do
        in: s = Ready --> b := True || c := c + n
        ch: s = Ready \ cr > 0 --> if b then s := Busy else skip || cr := 0
        gv: s = Busy --> s := Ready || b := False

AGILE Workshop, Pisa, July 2003
Result of an application of parametric unit $C$ to the model $(P, \mu)$ is obtained by applying the connector implementing $C$ to the program $P$ (this is the program component of the model) and composing $\tau$ with the refinement morphism $\mu'$ resulting from the connector application (this is the morphism component of the model).

To perform the application of $C$ to the program $A$, the latter must be first reduced to a model of $CountArg$. This is done by providing the fitting morphism $\gamma : CountArg \rightarrow Automaton$ and taking $(A, \gamma; \rho)$ as an argument for $C$ (as shown on the next slide).

The result of applying functor $C$ to program $A$ in the example is shown on the previous slide. The pair

$$(C(A), \{\text{in} \mapsto \text{insert}, \text{ch} \mapsto \text{choose}, \text{gv} \mapsto \text{give}\}, \{\text{cr} \mapsto c\})$$

is a model of $CountRes$. 
An example — semantics of parametric units

\[
\begin{array}{c}
E & \xrightarrow{\rho} & P \\
\downarrow & & \downarrow \\
E + D' & \xrightarrow{\gamma} & D \\
\downarrow & & \downarrow \\
D' & \xrightarrow{\sigma} & D' \\
\downarrow & & \downarrow \\
C(P) & \xrightarrow{\tau;\mu'} & C(P) \\
\end{array}
\]

design CountingAutomaton is

\[
\begin{align*}
\text{out } & st : \text{ Ready | Busy, cr : Nat} \\
\text{do } & \text{ insert } [] : \text{ True, False } \rightarrow \text{ cr' > cr} \\
& \text{ choose } [st] : \text{ st = Ready \wedge cr > 0, False } \rightarrow \text{ cr' = 0} \\
& \text{ give } [st] : \text{ st = Busy } \rightarrow \text{ st' = Ready}
\end{align*}
\]

\((C'(A), \{\text{in } \mapsto \text{ insert, ch } \mapsto \text{ choose, gv } \mapsto \text{ give}\})\) is a model of CountingAutomaton
The specification of the application of parametric unit $C : D \to D'$ to the model $(P, \rho : E \to P)$ is given by the pushout (in a category of CommUnity designs and superposition morphisms) of $D'$ and $E$.

In the case of our example the specification resulting from the application of $C$ to $A$ is isomorphic to $CountingAutomaton$, the declared specification of unit $CA$ in the architectural specification. In general, the refinement morphism may be provided to fit the declared specification into the computed one.

Further development is shown in less detail on the following slides. Parametric unit $D$ adds the attribute $dr$ that can be modified by the action $choose$. Units $TA$ and $KA$ specialise general $choose$ to two more specific actions: $choose\_tea$ and $choose\_coffee$ respectively.
program D(C(A)) is
  out s : Ready | Busy
    b : Bool
    n : Nat+
    c : Nat
    d : Drink
  do in: s = Ready --> b := True || c := c + n
    ch: s = Ready \ cr > 0 --> if b then s := Busy else skip || cr := 0
    gv: s = Busy --> s := Ready || b := False

design DrinkAutomaton is
  out st : Ready | Busy, cr : Nat,
    dr : Drink
  do insert []: True, False --> cr’ > cr
    choose [st,dr]: st = Ready \ cr > 0, False --> cr’ = 0
    give [st]: st = Busy --> st’ = Ready
design DrinkAutomaton is

out st : Ready | Busy, cr : Nat,
    dr : Drink

  do insert []: True, False --> cr' > cr
  choose [st,dr]: st = Ready \ cr > 0, False --> cr' = 0
  give [st]: st = Busy --> st' = Ready

design CoffeeAutomaton is

out st : Ready | Busy, cr : Nat,
    dr : Drink

  do insert []: True, False --> cr' > cr
  choose_coffee [st,dr]: st = Ready \ cr >= price(Coffee), False
                                 --> cr' = 0 || dr' = Coffee
  give [st]: st = Busy --> st' = Ready
design CoffeeAutomaton is
    out st : Ready | Busy, cr : Nat,
        dr : Drink
    do  insert []: True, False --> cr’ > cr
        choose_coffee [st,dr]: st = Ready \ cr >= price(Coffee), False
                                --> cr’ = 0 || dr’ = Coffee
        give [st]: st = Busy  --> st’ = Ready

design TeaAutomaton is
    out st : Ready | Busy, cr : Nat,
        dr : Drink
    do  insert []: True, False --> cr’ > cr
        choose_tea [st,dr]: st = Ready \ cr >= price(Tea), False
                                --> cr’ = 0 || dr’ = Tea
        give [st]: st = Busy  --> st’ = Ready
At this moment there are two superpositions of *DrinkAutomaton*, developed independently from some common machinery.

The last step is to put this two superpositions together obtaining the design *VendingMachine*:

\[
\text{prog } KTA = KA + TA : \text{VendingMachine}
\]

Note, that this operation is not a *synchronisation* of *KA* with *TA* which would result in having one version of *choose_coffee* only. *VendingMachine* is neither a superposition of *TeaAutomaton* nor of *CoffeeAutomaton* but it is a superposition of *DrinkAutomaton*. 
Conclusions

• CASL-style architectural specifications deal with refinement, CommUnity-style — with superposition

• Similar pragmatics but semantics based on different properties

• CommUnity-style architectural specifications are more general?
The example we presented showed how to structure the superposition of CommUnity programs rather than the refinement. In CASL-style specifications one has that $F(A)$ extends and preserves $A$. Analogous requirement in CommUnity would be that $C(P)$ is a refinement of $P$.

But when both $P$ and $C(P)$ are programs, a refinement $\rho : P \rightarrow C(P)$ is also a superposition, thus one may consider parametric units defined in terms of superposition as more general than the ones defined in terms of a refinement.

One may also ask whether the requirement that parametric units describe a superposition (as is the case for all the units in the example) is indeed essential — this will be the subject of further work.
Future work

• Application to one of the case studies involving mobility

• Considering more general class of parametric units, not only the ones based on the superposition

• Better model theory for CommUnity: definition of institution with refinement morphisms as specification morphisms