A Formal Framework for a Functional Language with Adaptable Components

Pascal Coupey, Christophe Fouqueré

A Formal Framework for a Functional Language with Adaptable Components

P. Coupey C. Fouqueré

LIPN – UMR7030
CNRS – Université Paris 13
99 av. J-B Clément, F–93430 Villetaneuse, France
firstname.lastname@lipn.univ-paris13.fr

Abstract

We propose a component programming language called FLAC, *Functional Language for Adaptable Components*, on top of a functional programming language which authorizes full adaptability of components while ensuring type safety. The langage is given together with a type system that offers a complete static type ckecking of any programs (including adaptations) to ensure error-free run-time adaptations. Dynamic adaptability and static type checking might seem at first sight paradoxical, but our approach allows it because, first, we use a single language for traditional services and control services (i.e., services for adaptations), and secondly, a specific merge operation takes care of adaptations.[1]

1. Introduction

Component-based programming is like a construction game: programs are assembled out of black boxes called components. The term 'black box' is justified by the fact that only the interfaces of components are publicly available. Such an interface characterizes the signatures of services the component offers or needs. The basic construct *plug* assembles a component that provides a service to a component that needs it. The domain of component-based systems (CBS) is mature enough to actually endorse the benefits of such a paradigm [12]. *Safety* and *adaptativity* are the two main requirements for such systems. Safety is ensured when execution is error-free. For that purpose, most CBSs expect each required service used by a component to be satisfied by a service provided by another component (soundness) and any service offered by a component to be concretely defined (completeness) [7, 9]. Obviously, type checking must be done accordingly. Adaptative capabilities are required to fit the evolution of the environment, software or hardware. Hence it is often given as an important goal for CBSs [3]. It may be minimal as in works of Aldrich [1] or Sreedhar [11] or first works of Costa Seco et al. [8] or more developed as in works of Dowling et al. [5] or Batista et al. [2], Bruneton et al. [4] or last works of Costa Seco et al. [9, 10]. A CBS is *dynamic* if adaptations can be made at runtime as reactions for contextual events. If adaptations are made by an external actor (e.g., a programmer) via a declarative or procedural interface, adaptations are called *external* although they are called *internal* when the code and the prerequisites are described in the program itself. A CBS is called *closed* when all adaptations are internal. Current dynamic closed CBSs provide a meta-model by incorporating a specific adaptation language different[2] from the language allowing component description and assemble. They also have mechanisms ensuring integrity and consistency of the system during dynamic reconfiguration. The meta-model manages a configuration graph of components and connections that can be inspected and modified at runtime. For example in the Fractal framework [4], component controllers allow to predict component changes at runtime (deleting or replacing a component, adding/removing functionalities, changing links with other components, ...) and dynamic adaptation is achieved by introspecting and reconfiguring the internal structure of components. The implementation of adaptability is frequently done by modifying the byte-code at runtime. However, in most existing CBSs, adaptability is limited if type checking is in use: the type of the new component should be a subtype of the type of the old component. The type of a component is roughly defined in current CBSs as the set of the signatures of its provided and requested services, and subtyping is covariant with provided services and contravariant with requested services. Furthermore usual target languages, e.g., JAVA [1, 4], are not as strongly typed as one can expect as type conversion is allowed. Note also that types are not automatically inferred in these languages. Thus, if the language is not constrained, the programmer could easily override typing constraints, giving rise to unexpected results or software crashes. Costa Seco et al. [9, 10] propose a component-oriented programming language that authorizes adaptations of component objects thanks to *configurators*, i.e., reconfiguration scripts that describe architectural operations modifying the structure of a component. Type control is ensured by configurator type declaration using a specific language. A configurator type describes the pre and post conditions for the application of the configurator. However, first the architectural operations in the configurator description is outside the static context of the components on which they are applied, hence they are white-box operations. Consequently, the type of a configurator cannot be automatically inferred: a configurator definition requires an explicit type. Second, precondition type mismatches are not visible to the type system: the programmer has to prevent them by means of conditional structures. To summarize, two drawbacks prevent a plain use of CBSs:

- Adaptability is constrained by the way the component type is defined: adaptations cannot alter the signature of required or provided services. However, the environment may simultaneously evolve in such a way that the final system is still correctly typed.
- Casting and byte-code modification forbid static analysis of a code, hence safety of program execution.

The language FLAC, *Functional Language for Adaptable Components*, presented in this paper allows dynamic internal adaptations without the two previous drawbacks: dynamic adaptations

[1] This work is supported by the Marie Curie action n. 29849 Websicola.

[2] Even if the difference is sometimes more or less subjective [4].

may remove or add requested or provided services, while types are statically checked:

1. Adaptation constructs are first-class entities: there is no meta-language. Hence, type specification integrates adaptations declared in the code: the type of a component is not only given by the actual signatures of requested and provided services but also by those resulting from its adaptations. To the best of our knowledge it is the first proposal where the type of a component integrates adaptation services. It means that in our language, the type of a component without adaptation capabilities is not identical to the "same" component with such adaptation properties. Indeed, how a simple car toy can be considered as having the same type than a car toy which can become a robot thanks to an adaptation property? Consequently, type checking may be completely done statically.
2. Executing an adaptation *creates* a new component in the memory, whereas current approaches change bytecodes in place. Hence components have only one type for the whole run of the program. A garbage collector mechanism, concretely the one given with OCaml [6], allows for freeing unused components.
3. The language is strongly-typed (as well as OCaml) ensuring error-free run-time adaptations.

In the next section we present the language FLAC. We focus on adaptations capabilities with various examples. Its operational semantics is described in section 3 and we give in section 4 a typing system that ensures safeness.

2. The FLAC Language

We extend a functional programming language[3] mainly by means of a **Component** data structure that integrates an interface (requested and provided services) and a functional part intended to define service codes (see Fig. 1.). A **Service name** is a string. Return types of service expressions are automatically inferred. Component parameters may be any kind of expressions, including services as well as components. Services are referenced by Uniform Resource Services (URS). A **URS** is identified by a component followed by a service name, optionally with a signature. The syntax for expressions is augmented to take care of these new structures: `call`, `plug` and `merge` operations are sufficient to illustrate adaptation capability.

In Fig. 2, two components are declared: a database `Server` offers a service `u` dedicated to process database requests, a `Client` asks for a database service `p`. The program creates a client c_1 and a server s_1 and plugs them together. The *plug* expression creates a call indirection from `p` in c_1 to `u` in s_1. Note that such a plug sends back a new component, i.e., it does not modify in place the component c_1.

Adaptations in FLAC are given by *control services*, i.e., services that add/delete services, hence create new components. In figures, control services appear on top of components. Control services contribute to the definition of components without reference to a meta-language: not only they could be required or provided, but also they are undistinguishable from standard services. In fact, control services are those that return a component, contrarily to services that return basic data. The type system of FLAC is able to infer and check control services types. A `call` to a control service followed by a `merge` operation constructs a new component adapted from the old one. Two modes are proposed in expression (`merge`#e_3 e_1 e_2): `add` returns a new component by adding the contents of e_2 to e_1 while `sub` removes the contents of e_2 from e_1.

[3] OCaml [6] is used as a core functional language.

Component *default value is the empty set*
C ::= `Component`
`[Requested` {s_mtlist}]
`[Provided` {s_mtlist}]
`[Services` {S_list}]
`Cend;`
Service Name
s ::= *a string*
Service Declaration
s_mtlist ::= *empty* | $s[(t)]$; s_mtlist
Service Expression
S ::= $[t]$ $s[(t_p)]$ $(p) \Rightarrow e$
t is the service type, inferred by default, s is a service name,
e is the body of the service, with parameter p of type t_p,
if t_p is inferred by the type checker then it is added to the code.
S_list ::= *empty* | S ; S_list
Uniform Resource Service (URS)
urs ::= $[e_1]\#e_2[(t)]$
e_1 is a component (in the scope by default),
e_2 is a service name possibly with a type t.

Mode
m ::= `add` | `sub`
Expression
e ::= `expression of a functional language`
| (`call` $e_1[?e_3]$) *call of URS e_1 [with arg. e_3].*
Return: value of $e_1[?e_3]$
| (`plug` e_1 e_2) *plug in URS e_1 the URS e_2.*
Return: copy of the first param., the service plugged
| (`unplug` e_1) *unplug URS e_1.*
Return: copy of the param., the service unplugged
| (`merge`[#e_3] e_1 e_2) *merge comp. e_1 with comp. e_2.*
Return: copy of e_1 merged in mode e_3, `add` by def., with e_2
| C | s | urs

Fig. 1: Grammar of FLAC

```
let Client = Component
               Requested { p(string);}
             Cend;;

let Server = Component
               Provided  { u(string);}
               Services  { u(string req)⇒...;}
             Cend;;

(let c1=Client and s1=Server in
  (let c2=(plug c1#p  s1#u) in ...))
```

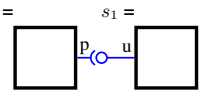

Fig. 2: Plugging two components

For readability purposes, we use the following aliases (where m is explicitly either `add` or `sub`):

- (m e_1 e_2) in place of (`merge`#m e_1 e_2)
- (`mself` e_1#s) in place of (`merge`#m e_1 (`call` e_1#s))

Note that in this last case, the result of `call` should be a component, i.e., s is a control service.

In Fig. 3, a database server component `Server1` may evolve by mixing up with an administration component `Admin`: The control service `r` of `Server1` returns a (copy of) component `Admin` that offers a service `a` dedicated to administration request processing. Executing on it `merge` in mode `add` results in component s_2 that is an adaptation of s_1. As s_2 provides itself the control service `r`, the administration service `a` may be removed from s_2, as illus-

trated when computing s_3. In the same way, it is obvious to add or to remove a requested service. For example, our server could evolve toward a securized server which needs a service `k` to encrypt or decrypt a string. To do this, simply add a control service which returns a component including a requested encryption/decryption service `k`.

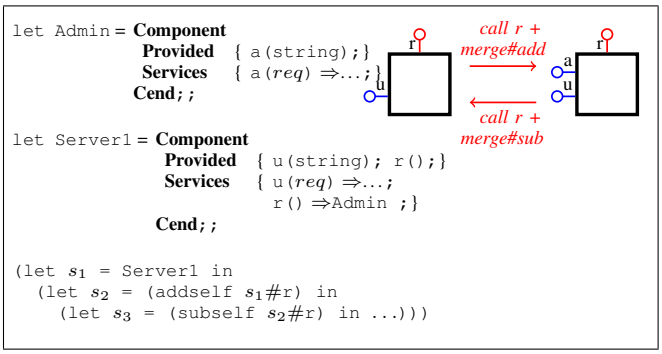

Fig. 3: Using control service: adding and removing a service

Example in Fig. 4 illustrates a succession of adaptations from a simple client-server couple plugged together to a securized client-server couple, that are plugged via both simple and administration database request services. We add to previous figures component `Client2`, a database client that can evolve toward one that needs an encryption/decryption service and/or one that requires an administration service `p`, and component `Server2`, that contains a control service `v` able to transform it to a securized server that needs an encryption/decryption service. Component `Crypt` proposes a service `v` for encrypting/decrypting strings. Note the difference between the way clients c_5 and c_7 are created: client c_5 shares the same encryption/decryption protocol with server s_3 (hence also s_4), whereas client c_7 is allowed to call the service `a` declared in server s_4.

Next two sections are devoted to the formal aspects of FLAC. Next section presents an operational semantics that gives the meaning of any valid program in FLAC while section 4 highlights the type system.

3. Operational Semantics

The operational semantics follow standard functional programming operational semantics: it is given as an evaluation judgment on programs and expressions to be computed with respect to a given environment. An environment is an *evaluation environment* together with a *handler environment*. An evaluation environment is a partial function from the set of variable names and component locations to values, either ground values or handlers to such values (supposing a domain of handlers). The evaluation environment has a special variable name '_self' whose value is a handler, it is supposed to be the handler of the component defined in the current context. A handler environment is a partial function from the set of handlers to values. Handlers are used to denote component values. The evaluation judgment for expressions is of the following form:

$$\mathcal{E}, \mathcal{H} \vdash e \Downarrow v, \mathcal{H}'$$

read as: the evaluation of expression e in an evaluation environment $\mathcal{E}$ with a handler environment $\mathcal{H}$ leads to a value v together with a handler environment $\mathcal{H}'$. A handler value is given as a function over the domain $\{req, prv, serv\}$. Values for req and prv selectors are sets of requested or provided services, i.e., signatures of services. The value of $serv$ selector is a map from service names to functional closures. We do not present the operational semantics

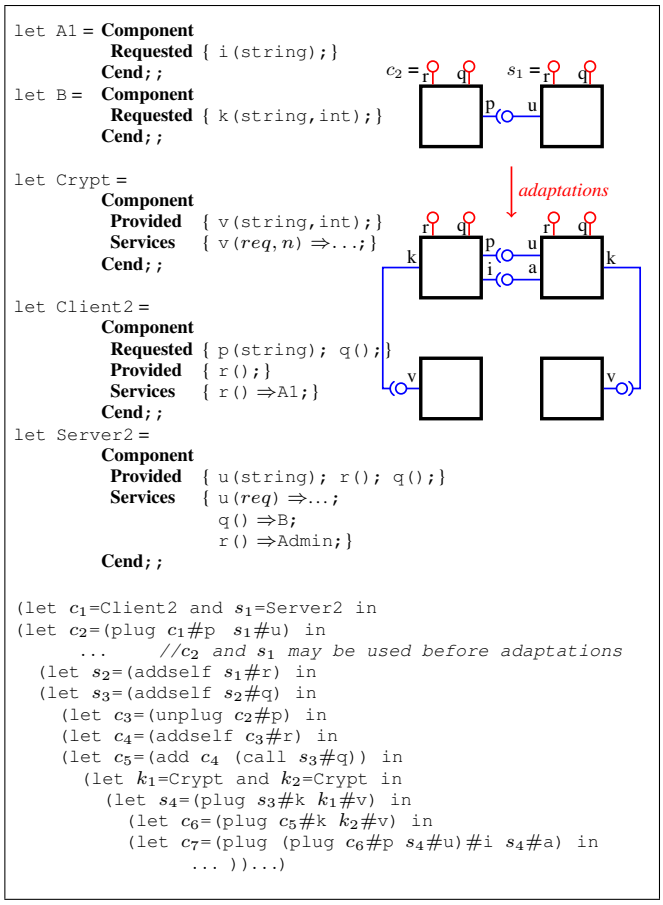

Fig. 4: Adapting components

of the functional part of the language. We extend a domain of basic (functional and data) values by the following kinds of values:

- h, value of a component, i.e., a *handler value*.
- (h, s, t_p), value of an URS where h is the value of a component, s is a service name, i.e., a string, t_p is a type (for discriminating services with the same name, it may be empty).

The operational semantics of Components is given in Fig. 5. The semantics corresponding to a Component declaration is straightforwardly given by lists of requested or provided services, and a closure similar to the treatment of functions: the rule is nothing else but a new value given for the reference. The lists of requested or provided services may be used if one extends the language by expressions asking for the interface of components. The semantics of plugging a service is similar to

```
Component
 Services   {s1(p) ⇒ (call e2?p);}
Cend
```

except that $\_self$ should refer to the component given in the first argument. The operation noted '$\overset{+}{\leftarrow}$' (resp. '$\overset{-}{\leftarrow}$') adds (resp. deletes) from its first argument the second argument if present. The semantics of an URS follows its syntactical structure. Rules are given in Fig. 6. A call to a service stipulates a service name and possibly a type, and as usual parameter values if needed.

There are various ways of merging components in FLAC. This gives the user full control over the model of components to have at the end and full modularity with respect to specification. Informally, two merge modes are given here:

$$\frac{h \ \texttt{fresh} \quad \mathcal{E}' = \mathcal{E} \overset{+}{\leftarrow} (\_self \mapsto h) \quad \mathcal{E}' \vdash Sl_3 \Downarrow L}{\mathcal{E}, \mathcal{H} \vdash \begin{array}{l}\texttt{Component}\\ \texttt{[Requested} \quad \{sl_1\}]\\ \texttt{[Provided} \quad \{sl_2\}]\\ \texttt{[Services} \quad \{Sl_3\}]\\ \texttt{Cend}\end{array} \Downarrow h, \mathcal{H} \cup \{h \mapsto \left\{\begin{array}{l}[req \mapsto sl_1,]\\ [prv \mapsto sl_2,]\\ [serv \mapsto L]\end{array}\right\}\}}$$

$$\frac{\begin{array}{c}\mathcal{E}, \mathcal{H} \vdash e_1 \Downarrow (h_1, s_1, t_{p1}), \mathcal{H}_1 \quad h_2 \ \texttt{fresh}\\ \mathcal{E}' = \mathcal{E} \overset{+}{\leftarrow} (\_self \mapsto h_2)\\ \mathcal{H}_2 = \mathcal{H}_1 \cup \{h_2 \mapsto \mathcal{H}_1(h_1) \cup \{serv[(s_1, t_{p1})] \mapsto cval(\mathcal{E}', (p) \Rightarrow (call\ e_2 ?p))\}\}\end{array}}{\mathcal{E}, \mathcal{H} \vdash (\texttt{plug}\ e_1\ e_2) \Downarrow h_2, \mathcal{H}_2[h_2(req) \overset{-}{\leftarrow} (s_1, t_{p1})]}$$

$$\frac{\begin{array}{c}\mathcal{E}, \mathcal{H} \vdash e_1 \Downarrow (h_1, s_1, t_{p1}), \mathcal{H}_1 \quad h_2 \ \texttt{fresh}\\ \mathcal{E}' = \mathcal{E} \overset{+}{\leftarrow} (\_self \mapsto h_2)\\ \mathcal{H}_2 = \mathcal{H}_1 \cup \{h_2 \mapsto \mathcal{H}_1(h_1)\}\end{array}}{\mathcal{E}, \mathcal{H} \vdash (\texttt{unplug}\ e_1) \Downarrow h_2, \mathcal{H}_2[h_2(prv) \overset{-}{\leftarrow} (s_1, t_{p1}), h_2(req) \overset{+}{\leftarrow} (s_1, t_{p1})]}$$

In the following rule, the expression defining the parameter has value `()` by default.

$$\frac{\begin{array}{c}\mathcal{E}, \mathcal{H} \vdash e_1 \Downarrow (h_1, s_1, t_{p1}), \mathcal{H}_1 \quad [\mathcal{E}, \mathcal{H}_1 \vdash e_2 \Downarrow v, \mathcal{H}_2]\\ h_1(serv)[(s_1, t_{p1})] = cval(\mathcal{E}_1, (p) \Rightarrow e_0) \quad matchPatt(v, p) = \mathcal{E}_0\end{array}}{\begin{array}{c}\mathcal{E}_1 \backslash (dom(\mathcal{E}_0) \cup h_1) \cup \mathcal{E}_0 \cup h_1, \mathcal{H}_2 \vdash e_0 \Downarrow v_0, \mathcal{H}_0\\ \hline \mathcal{E}, \mathcal{H} \vdash (\texttt{call}\ e_1 [?e_2]) \Downarrow v_0, \mathcal{H}_0\end{array}}$$

($matchPatt(\_,\_)$ is a standard pattern matching function returning the bound variables)
In the following rule, the expression defining mode has value `add` by default.

$$\frac{\begin{array}{c}\mathcal{E}, \mathcal{H} \vdash e_1 \Downarrow h_1, \mathcal{H}_1 \quad \mathcal{E}' = \mathcal{E} \overset{+}{\leftarrow} (\_self \mapsto h_1) \quad \mathcal{E}', \mathcal{H}_1 \vdash e_2 \Downarrow h_2, \mathcal{H}_2\\ [\mathcal{E}', \mathcal{H}_2 \vdash e_3 \Downarrow m, \mathcal{H}_3]\end{array}}{\mathcal{E}, \mathcal{H} \vdash (\texttt{merge}[\#e_3]\ e_1\ e_2) \Downarrow h_1 \leftarrow_m h_2, \mathcal{H}_3}$$

Fig. 5: Operational semantics for Components

$$\frac{h = \mathcal{E}(\_self) \quad \mathcal{E}, \mathcal{H} \vdash e_2 \Downarrow s, \mathcal{H}_1}{\mathcal{E}, \mathcal{H} \vdash \#e_2[(t)] \Downarrow (h, s, t), \mathcal{H}_1} \qquad \frac{\mathcal{E}, \mathcal{H} \vdash e_1 \Downarrow h, \mathcal{H}_1 \quad \mathcal{E}, \mathcal{H}_1 \vdash e_2 \Downarrow s, \mathcal{H}_2}{\mathcal{E}, \mathcal{H} \vdash e_1 \#e_2[(t)] \Downarrow (h, s, t), \mathcal{H}_2}$$

$$\frac{}{\mathcal{E}, \mathcal{H} \vdash empty \Downarrow [\,]} \qquad \frac{\mathcal{E}, \mathcal{H} \vdash S \Downarrow v \quad \mathcal{E}, \mathcal{H} \vdash S\_list \Downarrow L}{\mathcal{E}, \mathcal{H} \vdash S\ ;\ S\_list \Downarrow [v|L]}$$

$$\frac{}{\mathcal{E}, \mathcal{H} \vdash [t]\ s(t_p)\ (p) \Rightarrow e \Downarrow (s, t_p) \rightarrow cval(\mathcal{E}, (p) \Rightarrow e)}$$

Fig. 6: Operational semantics for URS and services

- The *add* mode is the default one. It consists in superposing the new component over the old one. In case two nodes exist, the sets of services are merged with a priority to the new component.
- The *sub* mode deletes from the old component declarations given in the second parameter of the call.

Formally, we define below the merge operations that serve for the operational semantics and the typing system. A component c is considered in the following definition as a partial function $\mathcal{N} \longrightarrow \mathcal{V}$ where $\mathcal{N}$ is a set of (service) names, $\mathcal{V}$ is a set of values (e.g., service codes). We note $dom(c)$ the domain of the partial function c.

Definition 1. *Let c_1, c_2 be two components, $c_1 \leftarrow_{op} c_2$, where $op \in \{add, sub\}$, is a component defined by:*

- $\forall n \in dom(c_2)$, *if* $op =$*add*, $(c_1 \leftarrow_{op} c_2)(n) = c_2(n)$,
- $\forall n \in dom(c_1) \backslash dom(c_2)$, $(c_1 \leftarrow_{op} c_2)(n) = c_1(n)$,
- *otherwise functions are undefined.*

Components

t_c	$::= (\rho,\rho,\rho)$	*component type*
ρ	$::= \epsilon$	*end of the sequence of service types*
	$\mid \ s \rightarrow [t_p\ [\rightarrow t]],\rho$	*service type*

Fig. 7: Type Language

4. Typing System

One adds, to a standard language of functional types, types t_c for component expressions. Such a type t_c is a triple of sequences of service types. A service type is a mapping from a service name to a type of the service code. The type of the service code may not be given: this case corresponds to a requested service. Otherwise it is a mapping from a pattern type t_p to a type. The system we present below is a type checker, however it requires only standard methods to derive from it a type inference system. The type environment is a partial function from a set of variable names and a special name `self` to types. Typing judgments are of one of the following forms[4] where Δ is a type environment and t is a type:

- $\Delta \vdash e : t$ where e is an expression
- $\Delta \vdash_s e : s$ where e is a service name (string)
- $\Delta \vdash_{\text{URS}} e : (t_c,s,t_p)$ where e is an URS
- $\Delta \vdash_{\texttt{S\_list}} e : t$ where e is a list of services
- $\Delta \vdash_{\texttt{mtlist}} e : t$ where e is a list of services, either requested or provided
- $\Delta \vdash_p p : t_p$ where p is a pattern

As components are not modified in-place, there is no specific difficulties in the typing system as soon as one considers that a component is a partially defined object. Hence its expected services are marked as partially defined function types, and a plug operation completes accordingly the type of the provided service in the component. Conversely, an unplug operation partially undefines part of the component type. Finally, as in the operational semantics, in a call operation, the special name `self` is defined to be the type of the first argument to be reused during the typing of the second argument of the call operation if the component is omitted. Note that this special name `self` is not part of the (user) type language.

Types for service names and URS follow the data structure (in URS rule, t_p is empty if t is not given):

$$\frac{s \ \texttt{string}}{\Delta \vdash_s s : s} \qquad \frac{t_c = \Delta(\texttt{self}) \quad \Delta \vdash_s e_2 : s \quad [\Delta \vdash_p t : t_p]}{\Delta \vdash_{\text{URS}} \#e_2[(t)] : (t_c,s,t_p)}$$

$$\frac{\Delta \vdash e_1 : t_c \quad \Delta \vdash_s e_2 : s \quad [\Delta \vdash_p t : t_p]}{\Delta \vdash_{\text{URS}} e_1 \#e_2[(t)] : (t_c,s,t_p)}$$

Service declarations, respectively service interfaces, are typed with $\vdash_{\texttt{S\_list}}$, respectively $\vdash_{\texttt{mtlist}}$, judgment rules. It consists mainly of mappings from service names to code types. The $\vee$ operation merges two lists: if a service name is present in each list then the code type should be equal[5] or at most one is defined (see $\vdash_{\texttt{mtlist}}$ judgment rules below), otherwise the service name is just added.

$$\frac{}{\Delta \vdash_{\texttt{S\_list}} :} \qquad \frac{\begin{array}{c}\Delta \vdash_s s : s \quad \Delta \vdash_p p : t_p \quad \Delta \vdash e : t\\ \Delta \vdash_{\texttt{S\_list}} S\_list : \rho \quad [t' = s \rightarrow t_p \rightarrow t]\\ \rho' = \rho \vee (s \rightarrow t_p \rightarrow t)\end{array}}{\Delta \vdash_{\texttt{S\_list}} [t']\ s[(t_p)]\ (p) \Rightarrow e\ ;\ S\_list \ : \ \rho'}$$

[4] We omit type judgments and type rules for the functional language.

[5] We do not consider subtyping in this paper.

Provided and requested service declarations are typed with $\vdash_{\texttt{mtlist}}$ judgment rules. Note that requested services are given service types without code types.

$$\frac{}{\Delta \vdash_{\texttt{mtlist}} :} \qquad \frac{\Delta \vdash_{\texttt{s}} s : s \quad [t \text{ a type}] \quad \Delta \vdash_{\texttt{mtlist}} s\_mtlist : \rho \quad \rho' = \rho \vee (s \rightarrow t \rightarrow)}{\Delta \vdash_{\texttt{mtlist}} s[(t)] \,;\, s\_mtlist : \rho'}$$

A component type merges together service types as declared in the three parts of a component definition.

$$\frac{\Delta, \vdash_{\texttt{mtlist}} sl_1 : \rho_r \quad \Delta, \vdash_{\texttt{mtlist}} sl_2 : \rho_p \quad \Delta, \vdash_{\texttt{s\_list}} Sl_3 : \rho_s}{\Delta \vdash \begin{array}{l}\texttt{Component}\\ \quad[\texttt{Requested} \quad \{sl_1\}]\\ \quad[\texttt{Provided} \quad \{sl_2\}] \quad : (\rho_r,\rho_p,\rho_s)\\ \quad[\texttt{Services} \quad \{Sl_3\}]\\ \texttt{Cend}\end{array}}$$

For typing the plug operation, a virtual component is created that serves as a go-between to send the call from a component to another one. The unplug operation is typed accordingly: a virtual component is created containing only an undefined service type for what is unplugged.

$$\frac{\begin{array}{c}\Delta \vdash_{\texttt{URS}} e_1 : (t_{c1},s_1,t_{p1}) \quad t_{c1} = (\rho_{r1},\rho_{p1},\rho_{s1})\\ \Delta, \texttt{self} \mapsto t_{c1} \vdash_{\texttt{URS}} e_2 : (t_{c2},s_2,t_{p2})\\ \Delta, v \mapsto t_{p1} \vdash (\texttt{call}\ e_2?v) : t_2\end{array} \quad \rho'_r = \rho_{r1} \overleftarrow{\leftarrow} s_1 \rightarrow t_{p1} \quad \rho'_s = \rho_{r1} \overleftarrow{\leftarrow} s_1 \rightarrow t_{p1} \rightarrow t_2}{\Delta \vdash (\texttt{plug}\ e_1\ e_2) : (\rho'_r,\rho_{p1},\rho'_s)}$$

$$\frac{\Delta \vdash_{\texttt{URS}} e_1 : (t_{c1},s_1,t_{p1}) \quad t_{c1} = (\rho_{r1},\rho_{p1},\rho_{s1}) \quad \rho'_r = \rho_{r1} \overleftarrow{\leftarrow} s_1 \rightarrow t_{p1} \quad \rho'_s = \rho_{r1} \overleftarrow{\leftarrow} s_1 \rightarrow t_{p1} \rightarrow}{\Delta \vdash (\texttt{unplug}\ e_1) : (\rho'_r,\rho_{p1},\rho'_s)}$$

The `call` result type is the result type of the service.

$$\frac{\Delta \vdash_{\texttt{URS}} e_1 : (t_{c1},s_1,t_{p1}) \quad t_{c1} = (\rho_{r1},\rho_{p1},\rho_{s1}) \quad [\Delta \vdash e_2 : t_p] \quad s_1 \mapsto t_p \mapsto t' \in \rho_{s1} \quad s_1 \mapsto t_p \in \rho_{p1}}{\Delta \vdash (\texttt{call}\ e_1[?e_2]) : t'}$$

The type language is extended by a mode value $\mu \subset \{\texttt{add}, \texttt{sub}\}$. The type of a component expression is a sequence of service types. A service type is a mapping from a service name to a set of modes and a code type (the type of the service code). The set of modes declares the available ways to call the service. Finally, as in object functional programming, component locations appear as types. This generates a finite set of types for a finite program. The `merge` operation merges the two component types with respect to the mode. In rule below, $m = \texttt{add}$ if e_3 is empty:

$$\frac{\Delta \vdash e_1 : t_{c1} \quad \Delta, \texttt{self} \mapsto t_{c1} \vdash e_2 : t_{c2} \quad [\Delta \vdash_{\mu} e_3 : m]}{\Delta \vdash (\texttt{merge}[\#e_3]\ e_1\ e_2) : t_{c1} \leftarrow_m t_{c2}}$$

Type safety follows from verifying standard type preservation properties. Hence well-typed expressions are evaluable, i.e., there cannot be evaluation errors (provided for the functional language part an operational semantics safe with respect to a classic typing). E.g., if expressions typable in the underlying functional langage are always reducible to values:[6]

Theorem 1. *Let e be an expression of the language, if $\vdash e : t$ is provable, then there exist $v, \mathcal{H}$ such that $\vdash e \Downarrow v, \mathcal{H}$ is provable.*

[6] A more general statement may be given: if an expression is well-typed then it is either a value or it is reducible with a small-step version of the operational semantics.

The proof is standard and the more general property is checked: let e be an expression of the language, if $\Delta \vdash e : t$ is provable, then there exist $v, \mathcal{E}, \mathcal{H}, \mathcal{H}'$ such that $\mathcal{E}, \mathcal{H} \vdash e \Downarrow v, \mathcal{H}'$ is provable, furthermore if t is a component type then v is a handler with value in $\mathcal{H}'$ such that the structure of this value follows the structure of t, finally if $\Delta(x)$ is defined then also $\mathcal{E}(x)$ and if $\mathcal{E}(x)$ is a handler then $\mathcal{H}(\mathcal{E}(x))$ has a value. Similar properties for other type inference systems, i.e., $\vdash_{\texttt{URS}}, \ldots$ The proof is straightforward if we notice that there is no in-place adaptation and the underlying programming language is supposed to be functional, thus adaptations do not lead to type changes. Note that there is no necessity for a component to have values for all its required services: the typing system ensures that a call is correct as soon as what is needed for the call to be executed is present in the component, and only that. However, in case of distributed systems or web services, a dynamic type-checking has to be added as one cannot be sure that requests are well-formed with respect to the program.

5. Conclusion

The FLAC programming language deals with adaptable components. Its main feature concerns dynamic internal adaptations in a strongly-typed language. It is facilitated by the fact that adaptations are described in the same language as the component description language. This language may be developed in several directions we currently study. Among them, structuring components is in practice highly expected as it increases the modularity of the language. This may be done by adding named parameters to component definitions. Such named parameters not only allow assignments of (sub)components but they may be used for denoting them. It is then possible, for the programmer, to write control services such that the part of the component not involved in the evolution is automatically rebuilt. This may be implemented by abstractly manipulating the structure of components, i.e., addressing each subcomponent by its logical named path.

References

[1] Jonathan Aldrich. Using types to enforce architectural structure. In *WICSA '08: Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)*, pages 211–220, Washington, DC, USA, 2008. IEEE Computer Society.

[2] Thaís Vasconcelos Batista, Ackbar Joolia, and Geoff Coulson. Managing dynamic reconfiguration in component-based systems. In *Software Architecture, 2nd European Workshop, EWSA*, pages 1–17, Pisa, Italy, 2005.

[3] Jan Bosch, Clemens A. Szyperski, and Wolfgang Weck, editors. *Proceedings of the Third International Workshop on Component-Oriented Programming*, Brussels, Belgium, 1998. Turku Centre for Computer Science.

[4] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The FRACTAL component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Software: Practice and Experience*, 36(11-12):1257–1284, 2006.

[5] Jim Dowling and Vinny Cahill. The K-component architecture meta-model for self-adaptive software. In *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 81–88, London, UK, 2001. Springer-Verlag.

[6] Objective CAML homepage. http://caml.inria.fr/ocaml/index.en.html.

[7] Piriquito Margarida. *Type System for the ComponentJ Programming Language*. PhD thesis, Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia, 2009.

[8] João Costa Seco and Luís Caires. A basic model of typed components. In *ECOOP '00: Proceedings of the 14th European Conference on*

Object-Oriented Programming, pages 108–128, London, UK, 2000. Springer-Verlag.

[9] João Costa Seco and Luís Caires. Types for dynamic reconfiguration. In Peter Sestoft, editor, *ESOP*, volume 3924 of *Lecture Notes in Computer Science*, pages 214–229. Springer, 2006.

[10] Piriquito Margarida Seco João Costa, Silva Ricardo. ComponentJ: A component-based programming language with dynamic reconfiguration. *Computer Science and Information Systems*, 5(2):63–86, 2008.

[11] Vugranam C. Sreedhar. Mixin'up components. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 198–207, New York, NY, USA, 2002. ACM.

[12] Clemens Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, 2nd edition, 2002.

6. Appendix

Theorem 1. *Provided the underlying functional language evaluates typable expressions, let e be an expression of the language, if $\vdash e : t$ is provable, then there exist $v, \mathcal{H}$ such that $\vdash e \Downarrow v, \mathcal{H}$ is provable.*

Proof. We prove the following: Let e be an expression of the language, if $\Delta \vdash e : t$ is provable, then there exist $v, \mathcal{E}, \mathcal{H}, \mathcal{H}'$ such that $\mathcal{E}, \mathcal{H} \vdash e \Downarrow v, \mathcal{H}'$ is provable, furthermore if t is a component type then v is a handler with value in $\mathcal{H}'$ such that the structure of this value follows the structure of t, finally if $\Delta(x)$ is defined then also $\mathcal{E}(x)$ and if $\mathcal{E}(x)$ is a handler then $\mathcal{H}(\mathcal{E}(x))$ has a value. Similar properties for other type inference systems, i.e., $\vdash_{\text{URS}}$, ... The proof is done by considering each rule of the typing system.

- $\dfrac{s \ \texttt{string}}{\Delta \vdash_{\text{s}} s : s}$ then for all $\mathcal{H}$, $\dfrac{s \ \texttt{string}}{\mathcal{E}, \mathcal{H} \vdash s \Downarrow s, \mathcal{H}}$

- $\dfrac{t_c = \Delta(\texttt{self}) \quad \Delta \vdash_s e_2 : s \quad \Delta \vdash_p t_p : t_p}{\Delta \vdash_{\text{URS}} \#e_2(t_p) : (t_c, s, t_p)}$ then, given the hypothesis of the typing rule, there exist $\mathcal{E}, \mathcal{H}, h, \mathcal{H}_1, s, \mathcal{H}_2$, that justify the hypothesis of the operational rule:
$\dfrac{h = \mathcal{E}(\_self) \quad \mathcal{E}, \mathcal{H} \vdash e_2 \Downarrow s, \mathcal{H}_1}{\mathcal{E}, \mathcal{H} \vdash \#e_2(t_p) \Downarrow (h, s, t_p), \mathcal{H}_1}$

- $\dfrac{\Delta \vdash e_1 : t_c \quad \Delta \vdash_s e_2 : s \quad \Delta \vdash_p t : t_p}{\Delta \vdash_{\text{URS}} e_1 \# e_2(t_p) : (t_c, s, t_p)}$ then, given the hypothesis of the typing rule, there exist $\mathcal{E}, \mathcal{H}, h, \mathcal{H}_1, s, \mathcal{H}_2$, that justify the hypothesis of the operational rule:
$\dfrac{\mathcal{E}, \mathcal{H} \vdash e_1 \Downarrow h, \mathcal{H}_1 \quad \mathcal{E}, \mathcal{H}_1 \vdash e_2 \Downarrow s, \mathcal{H}_2}{\mathcal{E}, \mathcal{H} \vdash e_1 \# e_2(t_p) \Downarrow (h, s, t_p), \mathcal{H}_3}$

- $\dfrac{}{\Delta \vdash_{\text{s\_list}}}$: then for all $\mathcal{E}, \mathcal{H}$: $\dfrac{}{\mathcal{E}, \mathcal{H} \vdash \mathit{empty} \Downarrow [\,]}$

- $\dfrac{\Delta \vdash_{\text{s}} s : s \quad \Delta \vdash_p p : t_p \quad \Delta \vdash e : t \quad \Delta \vdash_{\text{s\_list}} S\_list : \rho \quad [t' = s \to t_p \to t] \quad \rho' = \rho \vee (s \to t_p \to t)}{\Delta \vdash_{\text{s\_list}} [t'] \ s[(t_p)] \ (p) \Rightarrow e \ ; S\_list \ : \ \rho'}$ then, given the hypothesis of the typing rule, there exist $\mathcal{E}, \mathcal{H}, v, L$, that justify the hypothesis of the operational rules:
$\dfrac{\mathcal{E}, \mathcal{H} \vdash [t] \ s(t_p) \ (p) \Rightarrow e \Downarrow (s, t_p) \to \mathit{cval}(\mathcal{E}, (p) \Rightarrow e)}{\mathcal{E}, \mathcal{H} \vdash S \Downarrow v \quad \mathcal{E}, \mathcal{H} \vdash S\_list \Downarrow L}$
and $\dfrac{}{\mathcal{E}, \mathcal{H} \vdash S \ ; S\_list \Downarrow [v|L]}$

- $\dfrac{}{\Delta \vdash_{\text{mtlist}}}$: This rule as well as the following one are used to present a list of available or requested services. The operational semantics is nothing else but the list itself.

- $\dfrac{\Delta \vdash_{\text{s}} s : s \quad [t \ a \ type] \quad \Delta \vdash_{\text{mtlist}} s\_mtlist : \rho \quad \rho' = \rho \vee (s \to t \to)}{\Delta \vdash_{\text{mtlist}} s[(t)] \ ; s\_mtlist : \rho'}$

- $\dfrac{\Delta, \vdash_{\text{mtlist}} sl_1 : \rho_r \quad \Delta, \vdash_{\text{mtlist}} sl_2 : \rho_p \quad \Delta, \vdash_{\text{s\_list}} Sl_3 : \rho_s}{\Delta \vdash \begin{array}{ll} \texttt{Component} & \\ \texttt{[Requested} & \{sl_1\}] \\ \texttt{[Provided} & \{sl_2\}] \\ \texttt{[Services} & \{Sl_3\}] \\ \texttt{Cend} & \end{array} : (\rho_r, \rho_p, \rho_s)}$
then, given the hypothesis of the typing rule, there exist $\mathcal{E}, \mathcal{H}, L$, that justify the hypothesis of the operational rule:
$\dfrac{h \ \texttt{fresh} \quad \mathcal{E}' = \mathcal{E} \overset{+}{\leftarrow} (\_self \mapsto h) \quad \mathcal{E}' \vdash Sl_3 \Downarrow L}{\mathcal{E}, \mathcal{H} \vdash \begin{array}{ll} \texttt{Component} & \\ \texttt{[Requested} & \{sl_1\}] \\ \texttt{[Provided} & \{sl_2\}] \\ \texttt{[Services} & \{Sl_3\}] \\ \texttt{Cend} & \end{array} \Downarrow h, \mathcal{H} \cup \{h \mapsto \left\{ \begin{array}{l} [req \mapsto sl_1,] \\ [prv \mapsto sl_2,] \\ [serv \mapsto L] \end{array} \right\}\}}$

- $\dfrac{\Delta \vdash_{\text{URS}} e_1 : (t_{c1}, s_1, t_{p1}) \quad t_{c1} = (\rho_{r1}, \rho_{p1}, \rho_{s1}) \quad \Delta, \texttt{self} \mapsto t_{c1} \vdash_{\text{URS}} e_2 : (t_{c2}, s_2, t_{p2}) \quad \Delta, v \mapsto t_{p1} \vdash (\texttt{call} \ e_2 ?v) : t_2 \quad \rho'_r = \rho_{r1} \overset{-}{\leftarrow} s_1 \to t_{p1} \quad \rho'_s = \rho_{r1} \overset{+}{\leftarrow} s_1 \to t_{p1} \to t_2}{\Delta \vdash (\texttt{plug} \ e_1 \ e_2) : (\rho'_r, \rho_{p1}, \rho'_s)}$ then, given the hypothesis of the typing rule, there exist $\mathcal{E}, \mathcal{H}, \mathcal{H}_1, h_1, s_1, t_{p1}$, that justify the hypothesis of the operational rule:
$\dfrac{\mathcal{E}, \mathcal{H} \vdash e_1 \Downarrow (h_1, s_1, t_{p1}), \mathcal{H}_1 \quad h_2 \ \texttt{fresh} \quad \mathcal{E}' = \mathcal{E} \overset{+}{\leftarrow} (\_self \mapsto h_2) \quad \mathcal{H}_2 = \mathcal{H}_1 \cup \{h_2 \mapsto \mathcal{H}_1(h_1) \cup \{serv[(s_1, t_{p1})] \mapsto \mathit{cval}(\mathcal{E}', (p) \Rightarrow (\mathit{call} \ e_2 ?p))\}\}}{\mathcal{E}, \mathcal{H} \vdash (\texttt{plug} \ e_1 \ e_2) \Downarrow h_2, \mathcal{H}_2[h_2(req) \overset{-}{\leftarrow} (s_1, t_{p1})]}$

- $\dfrac{\Delta \vdash_{\text{URS}} e_1 : (t_{c1}, s_1, t_{p1}) \quad t_{c1} = (\rho_{r1}, \rho_{p1}, \rho_{s1}) \quad \rho'_r = \rho_{r1} \overset{+}{\leftarrow} s_1 \to t_{p1} \quad \rho'_s = \rho_{r1} \overset{-}{\leftarrow} s_1 \to t_{p1} \to}{\Delta \vdash (\texttt{unplug} \ e_1) : (\rho'_r, \rho_{p1}, \rho'_s)}$ then, given the hypothesis of the typing rule, there exist $\mathcal{E}, \mathcal{H}, \mathcal{H}_1, h_1, s_1, t_{p1}$, that justify the hypothesis of the operational rule:
$\dfrac{\mathcal{E}, \mathcal{H} \vdash e_1 \Downarrow (h_1, s_1, t_{p1}), \mathcal{H}_1 \quad h_2 \ \texttt{fresh} \quad \mathcal{E}' = \mathcal{E} \overset{+}{\leftarrow} (\_self \mapsto h_2) \quad \mathcal{H}_2 = \mathcal{H}_1 \cup \{h_2 \mapsto \mathcal{H}_1(h_1)\}}{\mathcal{E}, \mathcal{H} \vdash (\texttt{unplug} \ e_1) \Downarrow h_2, \mathcal{H}_2[h_2(prv) \overset{-}{\leftarrow} (s_1, t_{p1}), h_2(req) \overset{+}{\leftarrow} (s_1; t_{p1})]}$

- $\dfrac{\Delta \vdash_{\text{URS}} e_1 : (t_{c1}, s_1, t_{p1}) \quad t_{c1} = (\rho_{r1}, \rho_{p1}, \rho_{s1}) \quad [\Delta \vdash e_2 : t_p] \quad s_1 \mapsto t_p \mapsto t' \in \rho_{s1} \quad s_1 \mapsto t_p \in \rho_{p1}}{\Delta \vdash (\texttt{call} \ e_1 [?e_2]) : t'}$ then, given the hypothesis of the typing rule, there exist $\mathcal{E}, \mathcal{H}, \mathcal{H}_1, h_1, s_1, t_{p1}$, that justify the hypothesis of the operational rule:
$\dfrac{\mathcal{E}, \mathcal{H} \vdash e_1 \Downarrow (h_1, s_1, t_{p1}), \mathcal{H}_1 \quad [\mathcal{E}, \mathcal{H}_1 \vdash e_2 \Downarrow v, \mathcal{H}_2] \quad h_1(serv)[(s_1, t_{p1})] = \mathit{cval}(\mathcal{E}_1, (p) \Rightarrow e_0) \quad \mathit{matchPatt}(v, p) = \mathcal{E}_0 \quad \mathcal{E}_1 \backslash (dom(\mathcal{E}_0) \cup h_1) \cup \mathcal{E}_0 \cup h_1, \mathcal{H}_2 \vdash e_0 \Downarrow v_0, \mathcal{H}_0}{\mathcal{E}, \mathcal{H} \vdash (\texttt{call} \ e_1 [?e_2]) \Downarrow v_0, \mathcal{H}_0}$

- $\dfrac{\Delta \vdash_{\text{URC}} e_1 : t_1 \quad \Delta, \texttt{self} \mapsto t_1 \vdash e_2 : t_2 \quad [\Delta \vdash_{\mu} e_3 : m]}{\Delta \vdash (\texttt{merge} [\#e_3] \ e_1 \ e_2) : t_1 \leftarrow_m t_2}$
then, given the hypothesis of the typing rule, there exist $\mathcal{E}, \mathcal{H}, \mathcal{H}_1, h_1, s_1, \mathcal{H}_2, h_2$, that justify the hypothesis of the operational rule:
$\dfrac{\mathcal{E}, \mathcal{H} \vdash e_1 \Downarrow h_1, \mathcal{H}_1 \quad \mathcal{E}' = \mathcal{E} \overset{+}{\leftarrow} (\_self \mapsto h_1) \quad \mathcal{E}', \mathcal{H}_1 \vdash e_2 \Downarrow h_2, \mathcal{H}_2 \quad [\mathcal{E}', \mathcal{H}_2 \vdash e_3 \Downarrow m, \mathcal{H}_3]}{\mathcal{E}, \mathcal{H} \vdash (\texttt{merge} [\#e_3] \ e_1 \ e_2) \Downarrow h_1 \leftarrow_m h_2, \mathcal{H}_3}$

□