A Smalltalk implementation of Exil, a Component-based Programming Language

Petr Spacek   Christophe Dony
Chouki Tibermacine
LIRMM, CNRS and Montpellier II University
161, rue Ada
34392 Montpellier Cedex 5 France
{spacek,dony,tibermacin}@lirmm.fr

Luc Fabresse
Université Lille Nord de France
941 rue Charles Bourseul
59508 DOUAI Cedex France
luc.fabresse@mines-douai.fr

Abstract

The component-based development paradigm brings various solutions for software reusability and better modular structure of applications. When applied in programming language context it changes the way dependencies and connections between software pieces are expressed. In this paper we present the Smalltalk implementation of "Exil", a component-based architecture description and programming language that makes it possible to use component related concepts (ports, interfaces, services, ...) at design and if wished at programming time. This proposal enables Smalltalk users to develop their applications in the component-oriented style.

Categories and Subject Descriptors D.1.0 [*Programming techniques*]: General—Component oriented programming technique; D.2.11 [*Software Architectures*]: Languages—Component Oriented Language

General Terms Component-based Programming

Keywords Component, Inheritance, Architectures, Programming, Substitutability

1. Introduction

In this work, we consider a software component as a piece of software which is a unit of deployment and composition with contractually specified interfaces and explicit dependencies. A component interacts with other ones only by well declared communication channels.

We have designed a programming language where a component is a basic concept to encapsulate data and functionality in similar way as an object does, but with respect to component design ideas such as independence, explicit requirements and architectures. The goal is to develop a language, in which an expert programmer can develop independent components, design for reuse [5], and a non expert programmer can develop applications by connecting previously developed components, design by reuse [5].

A component can be seen as a black-box which provides functionalities and explicitly expresses what it requires to be able to provide them. With OOP, objects usually require some other objects to be able provide services, for example a compiler class usually requires a parser, see Figure 1. This dependency is expressed by the public attribute p and by Compiler's constructor, see lines 2 and 4 of figure 1). From reuse and deployment point of view, object dependencies are not very well observable from the outside (except by using reflective means or somehow by reading a documentation if there is one). With Component-based programming (CBP), where component dependencies are explicit, it is clear what the deployment environment will need to provide so that the components can function, as illustrated on Figure 2, lines 2 and 14.

```
01 class Compiler {
02   public Parser p;
03
04   public Compiler(Parser p) {}
05   ...
06 }
07 class App {
08   void main(string[] args) {
09     Compiler c = new Compiler();
10     c.p = new SpecialParser();
11     // or Compiler c = new Compiler(new SpecialParser());
12     ...
13   }
14 }
```

Figure 1. The `Compiler` class declares `Parser` attribute, from the black-box viewpoint, this requirement is hidden to the user

When a component is connected to another one, generally to satisfy requirements, this defines a software architecture. A software architecture is an overall design of software system [10]. The design is expressed as a collection of components, connections between the components, and constraints on how the components interact. Describing architecture explicitly at the programming language level can facilitate the implementation and evolution of large software systems and can aid in the specification and analysis of high-level designs. For example, a system's architecture can show which components a module may interact with, help to identify the components involved in a change, and describe system invariants that should be respected during software evolution.

The work we present in this paper aims at proposing a Smalltalk implementation of a dynamically typed component-based programming language named Exil. Exil is based on Scl [4, 5] and extends it towards the explicit and declarative expression of requirements and architectures. It is based on the descriptor/instance dichotomy where components are instances of descriptors. It also provides an original inheritance system [9] which is not the scope of this paper.

[Copyright notice will appear here once 'preprint' option is removed.]

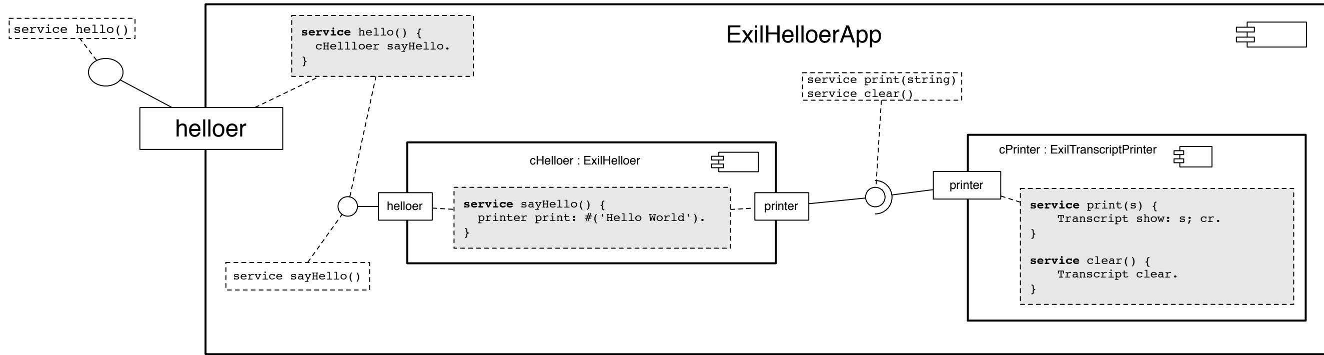

Figure 3. Diagram of the architecture of the *ExilHelloerApp* component

```
01 component Compiler {
02   require Parser p ;
03   ...
04 }
05 ...
06 component App {
07   provide service main(string[] args);
08
09   internalComponents {
10     Compiler c;
11     Parser p;
12   }
13   connections {
14     connect c.requirements.p to p.default
15   }
16
17   service main(string[] args) {
18     ...
19   }
20 }
```

Figure 2. Component-oriented languages explicitly express dependencies, i.e. component `Compiler` require a `Parser`

The paper is organized as follows. Section 2 proposes an overview of the Exil component model and programming language. Section 3 gives some basic clues on the Smalltalk implementation. Before concluding and discussing the future work, we briefly present in Section 4 the related work.

2. Exil Overview

This section presents the key concepts and structure of Exil.

In Exil, every component is an instance of a *descriptor*. A descriptor defines the behavior of its instances by *services* and their structure through *ports*, *internal components* and *connections*. The diagram of a simple descriptor is presented on Figure 3. *ExilHelloerApp* is a descriptor of a component saying hello, the Exil code of such a component is shown on Figure 4. It defines that the `ExilHelloerApp` component will maintain only one provided port called `helloer` providing service `hello`. The architecture of this component (field *internalComponents*) declares 2 internal components: `cHelloer` and `cPrinter`, and in the field *internalConnections* it defines how they are connected.

A descriptor may extend another descriptor, such a descriptor is called a *sub-descriptor*. A sub-descriptor inherits all ports, internal components and connection of super-descriptor (its parent), it may add new ports, new internal components and new connections or it may specialize them. A sub-descriptor may specialize an inherited port by modifying its interface. It may specialize an inherited internal component by modifying its default descriptor and finally may specialize connections by combination of `connect` and `disconnect` statements. A sub-descriptor may specialize an inherited service. Using classical mechanism of inheritance we do have common problems such as encapsulation violation between the components. More over, allowing additional requirements in a sub-descriptor the substitutability becomes more complicated [9]. However, modeling assets brought by inheritance are one major cornerstone of the success of object-oriented languages (1) for the ability it gives developers to organize their ideas on the base of concept classification (a list is a kind of collection, such architecture is a kind of visitor, ...) which is itself one key of human abstraction power and (2) for the calculus model introduced by object-oriented programming that makes it possible to execute an inherited code in the correct context (the receiver environment). In the off-the-shelf components context, as pointed by [6], a set of available black-box components cannot cover all possible scenarios of usage, and therefore an adaptation mechanism is needed. Inheritance can be the mechanism, which enables programmers to easily extend or specialize such components.

Services A descriptor may introduce services to specify functionality of its instances. When a service is listed in a provided port description, then the service is public. Each service has a signature given by the following template: `<service-name> (<argument1-name>, <argument2-name>, ... )`. A definition of a service consist of the `service` keyword followed by the *service signature* and a source code written in brackets after the signature, for example service `hello` in Figure 4.

The syntax of Exil is a mix of java-like syntax, used for specifying descriptors, and Smalltalk syntax used for service bodies implementation - this dichotomy is motivated by the fact that our language is currently implemented in the Pharo-Smalltalk environment [1], but we consider java-like syntax more readable and expressive for structural descriptions.

Components communicate by *service invocations* through their ports. A service invocation consists of a port, a selector (the name of the requested service) and arguments. Arguments are treated as in Scl, by temporary connections between arguments and ports temporary created for each argument. In case of argument incompatibility an exception is thrown.

A component c_1 can invoke a service of a component c_2 if a provided port p_2 of c_2 is connected to a required port p_1 of c_1. In this case, the service invocation is emitted via p_1 of c_1 and received via p_2 of c_2. When a component sends a service invocation i via one of its required ports r, the component checks if port r is connected to provided port p and if yes, then it transmits i via p, else a *does-not-understood* error is thrown.

```
component ExilHelloerApp {
  provide { helloer->{hello()} }
  internalComponents {
    cHelloer->ExilHelloer;
    cPrinter->ExilTranscriptPrinter;
  }

  internalConnections {
    connect cHelloer.printer to cPrinter.printer
  }

  service hello() { cHelloer sayHello }
}
```

Figure 4. Exil code of the ExilHelloerApp descriptor providing and implementing the hello service and having four internal components cHelloer and cPrinter inter-connected by connections specified in the *internalConnections* field

Interface An interface in Exil is a named list of service signatures. An interface is created by a statement with the following template: interface <interface-name> { <service-signature-1>; <service-signature-2>; ...}. We have introduced interfaces in Exil compared to Scl for convenience and reuse purposes. That means, we want to be able to reuse a list of services, used as a contract description by one component, as a contract description of another component.

Port A port is an unidirectional communication point described by a list of services signatures or by an interface reference (which makes it possible to reuse such a description) and by a role (*required* or *provided*). A provided port describes what is offered by a component. A required port describes what is demanded by a component. For example, a definition of two provided ports named A and B looks like: provide {A->{service1()}; B->ISecond}, where the A port provides a service called service1 and the B port is described by an interface ISecond.

Connection A connection between two components is performed by a connection between their ports. A descriptor lists all connections in the field *internalConnections*. A connection is specified by a statement described by the template
connect <an-emittor-port-address> to
<a-receiver-port-address>. By *the port-address* it is meant the expression <component-name>.<port-name>, for example see the field internalConnections in Figure 4. A component can act as an adapter between various other components and then, it is called a *connector*.

Internal component A component can own internal components. Such a component is then called *a composite*. The owning component references an internal component by a variable.

A list of internal components is defined in the descriptor's field *internalComponents*, see Figure 4. The Exil code of the cHelloer internal component is in Figure 5. Internal components are initialized during instantiation of the owning composite. By default all internal components are initialized with NilComponent component, a developer should implement an init service or optionally may specify a default descriptor in the internal components list, i.e. use the following statement internalComponents {cPrinter->Printer}, which is equivalent to the cPrinter := Printer new. line in the init service.

An internal component is encapsulated by the owning composite and it is not accessible from outside of the composite. Services defined by a composite can use internal components to implement a desired behavior (service invocation redirect).

```
component ExilHelloer {
  provide { helloer->{sayHello()} }
  require { printer->{print(string); clear()} }

  service sayHello() {
    printer print: #('Hello World').
  }
}
```

Figure 5. Exil code of the ExilHelloer descriptor providing and implementing the sayHello service and requiring services print and clear via required port printer.

3. Implementation

Exil is implemented in the Pharo Smalltalk environment [1] as an extension of Scl. We chose Smalltalk because of its reflective capabilities, which are necessary for mechanisms like the service invocation mechanism. And we chose Pharo environment for its rich set of support tools and frameworks like PetitParser [8] framework or Mondrian [7], which we use or will use for the Exil implementation.

The Exil implementation contains core classes representing Exil component model, then there are parser and compiler classes responsible for source code processing and classes implementing Exil GUI shown on Figures 6 and 7.

3.1 Parser & Compiler

Exil has custom syntax and therefore a special parser is required. We use PetitParser framework base for our parser which is represented by the ExilParser class, which inherits ExilGrammar class and extends it with AST building code. We have chosen PetitParser framework, because it allows us to maintain Exil grammar as easily as common Smalltalk code and because we can smoothly compose it with the PetitSmalltalk parser. The PetitSmalltalk parser is used for service bodies parsing. The result of our parser is an AST made from subclasses of the ExilParseNode class.

ExilCompiler transform the AST made by the parser to Exil core classes representation. The compiler is designed as a visitor of AST nodes.

3.2 Core

In our proposal, we use Smalltalk classes to implement component descriptors. All classes implementing component descriptors are subclasses of the base class called ExilComponent, the base class the mechanism to store information about provided and required ports, internal components and their interconnections contains in the class-side. This information is used at instantiation time by descriptors (initialization methods) to create components (descriptors instances).

Internally, for each port and each internal component, an instance variable is created in the class implementing the descriptor to hold references to port instances and internal components instances. Ports are implemented as objects. There is one class hierarchy for provided ports and one for required ports, all of them are subclasses of ExilPort base class. Ports are described by interfaces, which are implemented by arrays or by classes.

An interface is implemented as an array of service signatures. When an interface is defined as the named interface, for example interface IMemory { load(); save(data); }, then the IMemory sub-class of the class ExilInterface with methods load and save_data: is created. These methods are having empty bodies. A sub-interface is then implemented as a sub-class of the class representing its parent.

Services are represented by methods. An automatic (and transparent for the user) mapping of a service signature from Exil to

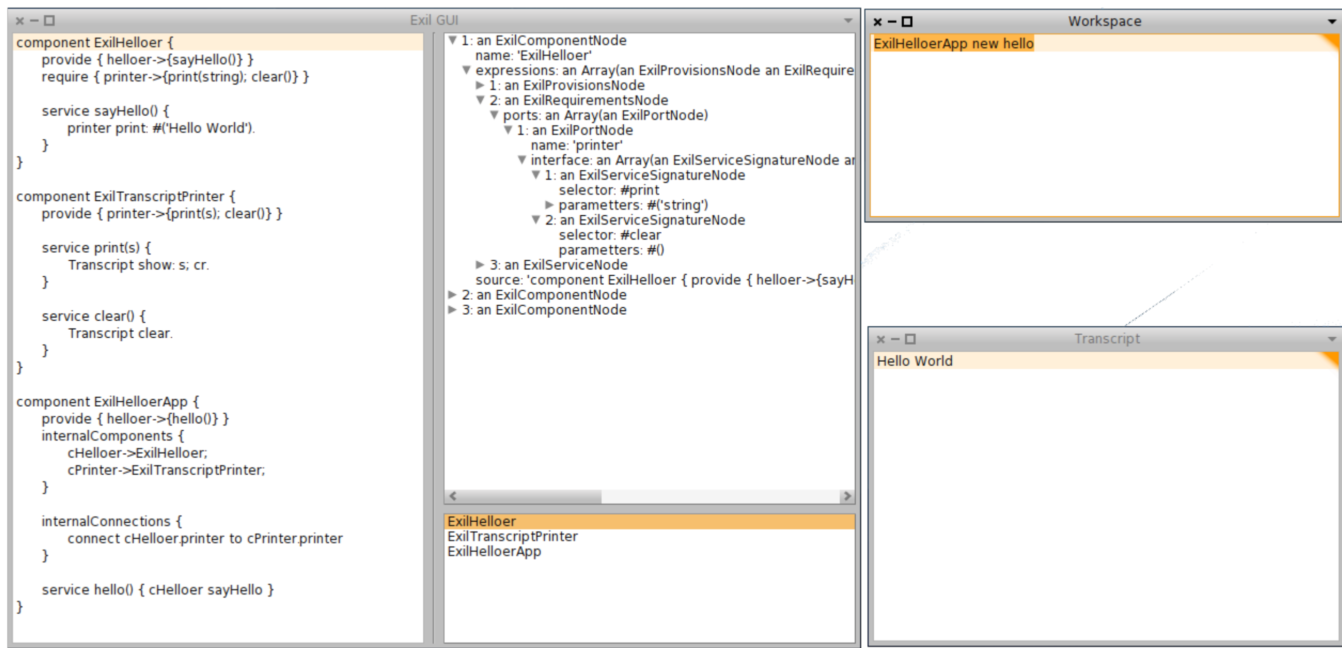

Figure 6. Exil GUI

a Smalltalk's method selector is based on naming convention, for example service signature `sum(numberA,numberB)` is mapped into a Smalltalk method with selector `sum_numberA:numberB:`. Since service bodies are written in Smalltalk, a call of the `sum` service performed in a body of an another service looks like `<receiver> sum:{ 1. 2 }`. The call is automatically dispatched to `sum_numberA:numberB:` method, according to parameters arity.

For each internal component and port there is an instance variable having the same name. Information about ports and associated interfaces or about default descriptors of internal components are stored as class side methods which return an array of associations, i.e. pairs of port name (resp. internal component name) and interface (resp. descriptor). Connections are stored similarly, as a class-side method which returns an array of associations. An association is a pair of port-addresses. We call these methods *the description methods*. For example the `ExilHelloerApp` descriptor shown in Figure 4 is implemented as a sub-class of the `ExilComponent` class named `ExilHelloerApp` having 3 instance variables, one named `helloer` representing the port *helloer* and two others representing internal components *cHelloer, cPrinter* and named in the same way as the internal components. The `ExilHelloerApp` metaclass implements four description methods called `providedPortsDescription`, `requiredPortsDescription`, `internalComponentsDescription` and `connectionsDescription`. Source code of the class is in Figure 8.

3.3 Inheritance implementation

A sub-descriptor is implemented as a sub-class of the class representing its super-descriptor. All these specializations are implemented as modifications of the description methods.

A service specialization is equal to the method overriding in Smalltalk. When a sub-descriptor specialize an inherited service, the corresponding method is then overridden in the sub-class realizing the sub-descriptor.

```
ExilComponent subclass: #ExilHelloerApp
  instanceVariableNames: 'helloer cHelloer cPrinter'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Exil-Examples-Helloer'.
!

ArithEvaluator class>>providedPortsDescription
  ^ #( #helloer->#(#hello) ).
!
ArithEvaluator class>>requiredPortsDescription
  ^ #().
!
ArithEvaluator class>>internalComponentsDescription
  ^ #( #cHelloer->ExilHelloer #cPrinter->ExilTranscriptPrinter ).
!
ArithEvaluator class>>connectionsDescription
  ^ #((#cHelloer->#printer)->(#cPrinter->#printer).
!
```

Figure 8. The ExilHelloerApp class implementing the `ExilHelloerApp` descriptor showed in Figure 4.

Service invocations Service invocations, in the context of inheritance, fully benefits from Smalltalk's message sending system. When a port receives a service invocation which is valid according to the contract specified by the interface of the port, it translates the service signature into a Smalltalk selector and the standard *method look-up* is performed. Since descriptors and sub-descriptors are realized by classes and subclasses, not extra mechanism is needed, the standard *method look-up* works perfectly.

Substitution and initialization/compatibility support Exil users are responsible for the `init` method implementation of a descriptor to achieve dependency injection, that is to say, for initializing variables referencing internal components used in the internal architecture. The first support comes in case when a sub-descriptor has an additional required port. Then our inheritance system automatically generates two new class instantiation methods (and two corresponding `init` methods, not described here), one `newCompatible` without parameter and `newCompatible:` having as unique param-

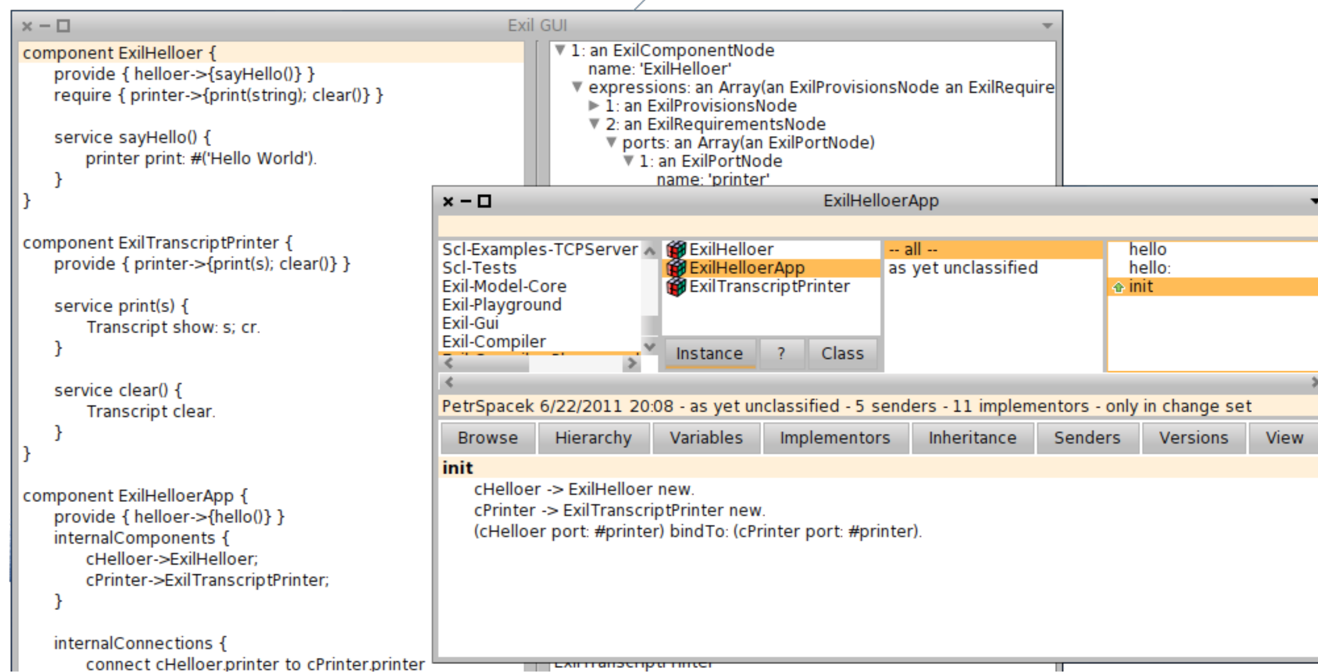

Figure 7. Component classes in browser

eter an array of pairs *port-component*. The first one is able to create an instance, compatible for substitution with instances of super-descriptors, for which all additional requirements are satisfied by connections to default instances of the required component. The second one does the same thing but uses its argument for connecting additional required ports.

The second support is a method to achieve substitutions safely. Substitution is achieved by a method `replace:with:` on instance-side of the base class `ExilComponent`. This method takes two arguments, the first one is the name of the internal component variable referencing the component which should be replaced and the second argument is the replacement component. `replace:with:` checks for original and replacement components descriptors compatibility then it checks if all requirements would be satisfied after substitution. If the descriptor of the new component is compatible with the descriptor of the original one and if all requirements of the new component are about to be satisfied, the replacement is performed otherwise an exception is thrown. `replace:with:` reconnects all ports of the original component to corresponding ports of the new component and change internal component reference of the composite to reference the new component.

Readers can download a Pharo image of Exil implementation here: http://www.lirmm.fr/∼spacek/exil/

4. Related work

We give in this section an overview of existing component models implemented on top of Smalltalk, and discuss their limitations.

CLIC Clic [2], an extension of Smalltalk to support full-fledged components, which provides component features such as ports, attributes, or architecture. From the implementation point of view, it fully relied on Smalltalk reflective capabilities. Thus, from the Smalltalk virtual machine point of view, CLIC components are objects and their descriptors are extended Smalltalk classes. Because of this symbiosis between CLIC and Smalltalk, the use of CLIC allows taking benefit from modularity and reusability of components without sacrifice performance. CLIC model allows components to have only one provided port. The idea of a single provided port is based on the observation that developers do not know beforehand, which services will be specified by each required port of client component. Therefore it is hard to split component functionality over multiple ports. CLIC also support explicit architecture description and inheritance. It does not need any additional parser or compiler.

FracTalk FracTalk[1] is a Smalltalk implementation of the Fractal hierarchical component model [3]. In FracTalk, a primitive component is implemented as a plain object. As in Exil, every port is implemented as a single object in order to ensure that every port allow invoking only declared operations. Therefore, a single component materializes as multiple objets. In opposite to Exil, the description of a component is scattered over multiple classes. A component in FracTalk is described by implementation class and factory class. Another limitation of FracTalk is the the difficulty to make use of Smalltalk libraries. Smalltalk objects aren't full fledged components since they do not have a membrane and then does not provide expected non-functional ports. Therefore, the only mean to use a Smalltalk object in a FracTalk application is to encapsulate it in the content of some component.

5. Conclusions

In this paper, we propose a Smalltalk implementation for a dynamically-typed component-based language. The language brings benefits of the component-paradigm closer to the Smalltalk users and it also provides solid soil for experiments in component software area. Exil allows programmers to express architectural structure and then seamlessly fill in the implementation with Smalltalk code, resulting in a program structure that more closely matches the designer's conceptual architecture. Thus, Exil helps to promote

[1] http://vst.ensm-douai.fr/FracTalk

effective architecture-based design, implementation, program understanding, and evolution.

We plan to work in the near future on the integration of our previous work on architecture constraint specification and architecture description structural validation [11, 12]. In this way, we can specify conditions on the internal structure of a component (its internal components and connections between them) or on its ports. This will help developers in better designing their systems by making more precise architecture descriptions. There are here some interesting issues that we foresee to study, as for example, architecture constraint inheritance.

In the future, we would like to switch from Smalltalk syntax used for services implementation to Ruby syntax, which is more similar to the syntax used for component structure description. For this purposes we would like to port SmallRuby[2] project into Pharo and develop Ruby parser using PetitParser [8] framework. We are also interested in visual programming and we plan to use the Mondrian [7] framework to enhance our user interface with auto-generated component/architecture diagrams.

References

[1] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example*. Square Bracket Associates, 2009. ISBN 978-3-9523341-4-0. URL `http://pharobyexample.org`.

[2] N. Bouraqadi and L. Fabresse. Clic: a component model symbiotic with smalltalk. In *Proceedings of the International Workshop on Smalltalk Technologies*, IWST '09, pages 114–119, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-899-5.

[3] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36: 1257–1284, September 2006. ISSN 0038-0644. doi: 10.1002/spe.v36:11/12. URL `http://portal.acm.org/citation.cfm?id=1152333.1152345`.

[4] L. Fabresse. *From decoupling to unanticipated assembly of components: design and implementation of the component-oriented language Scl*. PhD thesis, Montpellier II University, Montpellier, France, December 2007.

[5] L. Fabresse, C. Dony, and M. Huchard. Foundations of a simple and unified component-oriented language. *Comput. Lang. Syst. Struct.*, 34:130–149, July 2008. ISSN 1477-8424. doi: 10.1016/j.cl.2007.05.002. URL `http://portal.acm.org/citation.cfm?id=1327541.1327717`.

[6] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995. ISBN 978-0-201-63361-0.

[7] M. Meyer and T. Gîrba. Mondrian: Scripting visualizations. European Smalltalk User Group 2006 Technology Innovation Awards, Aug. 2006. URL `http://scg.unibe.ch/archive/reports/Meye06cMondrian.pdf`. It received the 2nd prize.

[8] L. Renggli, S. Ducasse, T. Gîrba, and O. Nierstrasz. Practical dynamic grammars for dynamic languages. In *4th Workshop on Dynamic Languages and Applications (DYLA 2010)*, Malaga, Spain, June 2010. URL `http://scg.unibe.ch/archive/papers/Reng10cDynamicGrammars.pdf`.

[9] P. Spacek, C. Dony, C. Tibermacine, and L. Fabresse. Reuse-oriented inheritance in a dynamically-typed component-based programming language. Technical report, LIRMM, University of Montpellier 2, May 2011.

[10] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002. ISBN 0201745720.

[11] C. Tibermacine, R. Fleurquin, and S. Sadou. A family of languages for architecture constraint specification. *In the Journal of Systems and Software (JSS), Elsevier*, 83(5):815–831, 2010.

[12] C. Tibermacine, S. Sadou, C. Dony, and L. Fabresse. Component-based specification of software architecture constraints. In *Proceedings of the 14th International ACM SIGSOFT Symposium on Component Based Software Engineering (CBSE'11)*, Boulder, Colorado, USA, June 2011. ACM Press.

[2] https://swing.fit.cvut.cz/projects/smallruby