SPIC: A System of Pure Interconnected Components

Rushikesh K Joshi and Neeraj Bhope

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay,
Mumbai, India, 400 076
Email: rkj@cse.iitb.ac.in

Abstract. Object-oriented software consists of interacting objects. For an interaction, objects need to know the *identity* and *interface* of each other. In contrast, in a real life component-oriented system such as an electronic circuit, components are independent and unaware of each other. We investigate a programming system of pure components, in which the interacting components need not be aware of each other's identity. The pure component system is implemented in CLOS based on a metaobject protocol. The language and its implementation is described.

1 Introduction

A *pure component* is defined as an independent entity unaware of the identity and interface abstractions of its peers. Objects are not pure components since peer objects need to depend on each other's interface abstractions while realizing an interobject relationship. In terms of implementation, an object maintains a handle to its peer object, through which, it invokes member functions on the peer. Thus the caller remains aware of the identity and interface abstraction of the callee. An interesting question is that can a system of pure components be wired to work collaboratively? If each object is to be unaware of its peers, how can one realize a collborative task? (A trivial solution with unacceptable engineering properties is to embedd all behaviour into one object.) In this work, we propose a system of pure components called *SPIC* and demonstrate a methodology to program collaborative tasks through it.

A *pure component* in SPIC does not send messages directly to its peers, but defines its behaviour in a self-contained fashion. A motivation behind SPIC is to investigate the feasibility of this approach and its consequences to program design and development. SPIC has been implemented using the CLOS metaobject protocol provided in Closette. As demonstrative eamples, two design patterns of Gamma [1] have been implemented in SPIC.

2 Motivation and Approach

Our approach towards *pure components* is based on observations of real-life components such as automobile component assemblies, electronic circuites and piping systems. Individual components in these systems are designed as complete units providing pluggable ports. Realizing a collaborative system of components involves coupling components together in right manner.

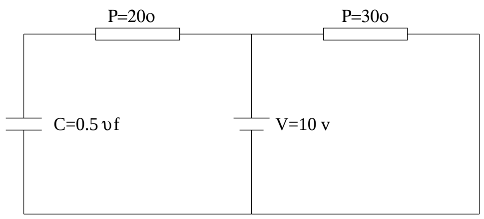

Fig. 1. Components in Real Life

Consider an electrical circuit shown in Fig. 1. The resistors, capacitors and the battary are pure components connected together forming a system: the circuit. The system can be evolved by replacing components upon aging, or for meeting a new performance criterion. Any change with an identity or interface of an involved component does not affect the neighbouring components. In other words, component replacements or configurations are carried out transparently. This has been possible since connectors are separated from component implementations. As noted by Peter Gabriel in his work on patterns of software [5], too much abstraction could be dangerous since any change to an abstraction's interface directly affetcs all its usage. Component implementations in object object oriented programming rely on peer components' abstractions, In SPIC, the decoupling of abstractions is achieved by separating connectors from component implementations.

2.1 Properties of SPIC Components

A component implementation does not know of the identities of its peer components since it does not maintain handles to its external peers. However, it provides a *connection* interface. Components are externally hooked to each other through their connection interfaces. The following properties are associated with *SPIC components.*

- Components do not directly interact with each other. They do not maintain handles to each other. It can be noted that a handle to intermediate entity (such as a proxy in a distributed object scenario) is also treated as a handle to external component due to the abstraction coupling.
- A system is realized by connecting components together for collaboration.
- Reconfiguration at connection level does not affect the components.
- Implementation changes within a component does not affect the external connections.

These properties are achieved through the following traits:

– Individual components export *ports*.
– External *Connectors* specify port connections.
– *Dispatchers* (per-class and per-object) associate method implementations with ports.
– A component can *drop* information into a port that it owns.

The above traits have been supported in SPIC through Closette MOP. The subsequent sections discuss the design, implementation and applications of SPIC.

3 Design of SPIC

This section describes the SPIC model of components and the related programming language features. At the core of the SPIC model, we have ports, dispatchers and connectors. Ports can carry untyped information from within a component to an external connector, which delivers the information to a peer component through its port hooked to the connector. Dispatcher dispatches the information to a method.

3.1 Ports

Ports are used to offer or request functionality. A method attached to gets invoked on receipt of information *dropped* on it by some other component through a connector. The dropped information is treated as argument list which is passed to the method.

Figure 2 shows two interconnected pure components. *Component a* obtains the services of an external component through a port connected to *component b*. However, *component a* remains unaware of its peer component since the former only communicates with its local port. Method *export1* is *attached* to *port2* in *component b*. In method *myfunc* in *component a*, the arguments are *dropped* on *port1*. The functionality provider component remains anonymous to *component a*.

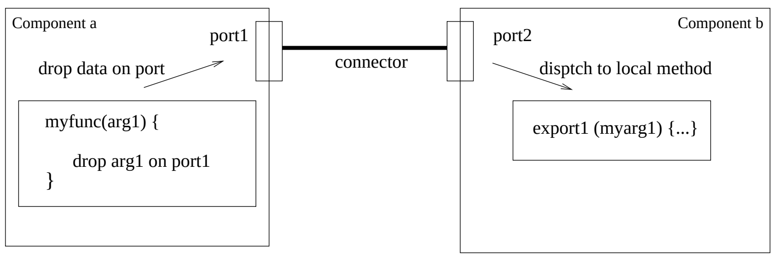

Fig. 2. Pure component interaction

3.2 Method Attachment

A method may be *attached* with a port at method definition time. The *attachment* of a method with a port can also be dynamic. This is done through a dispatcher protocol.

In Fig. 3, method `export1` is replaced by a new method `export2`, which is also defined on *component b*. Dynamic attachment and detachment allows facilitates dynamic changes in functionality offered at a port.

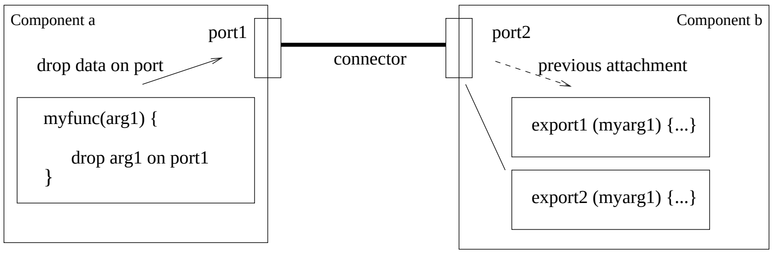

Fig. 3. Changing methods associated with a port

Method attachments can be applied on per-class or per-object basis. A *per-class attachment* affects all the component instances of a particular class while a *per-instance attachment* is effective only a particular instance.

While interactions with other components is achieved through ports, direct method invocation is supported within a component. This allows a component to be realized by means of a private implementation which may possibly include mixin classes.

3.3 Dynamically Reconfigurable Connectors

Connector space encapsulate interaction among components. Connections between components are stored by the system in a default connector space. It is also possible to extend or change connector configuration and hence the semantics of interaction at run time. Connector semantics manifest through a metaobject protocol implementation for CLOS.

In Figure 4, a connection switch results in an invocation of a method in *component c*, instead of a method in *component b*, as defined by the state change of the connector. The connector may change its state dynamically, i.e. it may switch its end virtices resulting in a different configuration. The configuration is changed without involving any of the connected components.

More than one components can be connected to a single port resulting in a *multicasting* interaction. A *drop* on the hub port results in method invocations on all the connected components through their respective dispatchers or method attachments.

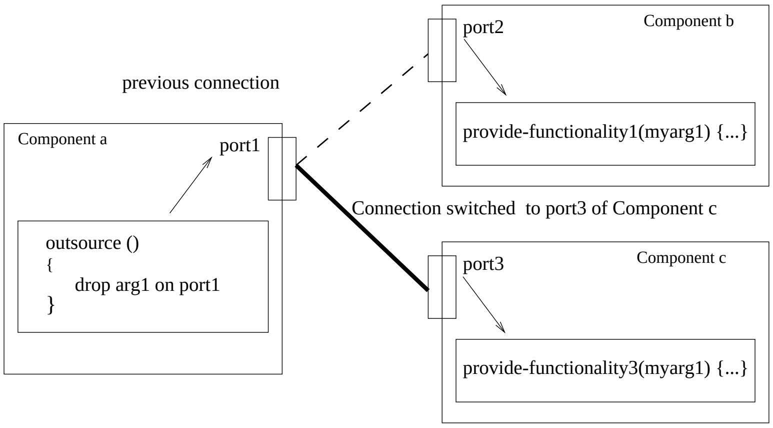

Fig. 4. Connecting components

3.4 A Simple Application.

An example is illustrated to demonstrate the applicability of the paradigm. Consider a web-browser application as shown in Fig. 5. Let's assume that this is the first version of the browser is equipped with the basic functionality and protocol handlers. In its new version, it is planned to introduce local in memory caching as a performance booster. The figure shows a high level model of the application abstracting interacting interfaces.

———— done till here june 11 sept 5 sept 24 ——————-

between the **user interface**, **URL-parser**, **protocol handlers** (http, ftp etc) and the **page parser** components. They encapsulate the computational aspects they perform and the interactions between these components forms the structural aspect of the application which can be reconfigured. Using our implementation of the *pure component* paradigm we can code these components and then *configure* or *connect* them to get the application. The actual program will contain component instances with connections replacing the arrows in the figure. Individual components only exchange data and particular **methods** and **instances** are **hidden** across the components.

In Fig. 6 the browser is augmented with caching. The change involves only changing the connections between the **user interface** and the **URL-parser** components. The process does not require any change in the component code as they are unaware of other components and their methods. Only code for the **cache** component, creating its instance and modifying the *connections* between the required components is added to the existing system.

Thus pure components can be developed without knowledge of the identity and interface of other pure components. Any pure component which satisfies the

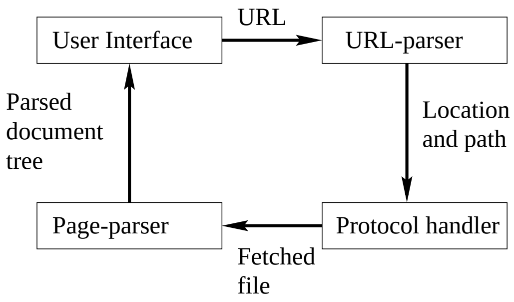

Fig. 5. A simple browser.

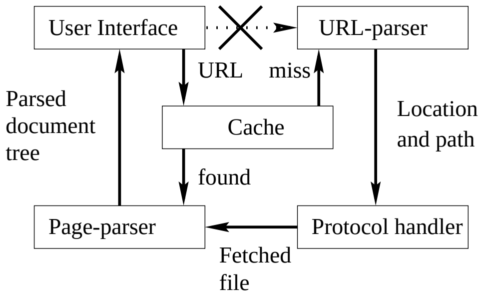

Fig. 6. The reconfigured browser.

functionality requirement can be connected to it. The connection mechanism also allows changes to be made to the software easily by simple re-wiring of components. This design is aimed only to exhibit the pure component paradigm and does not enforce method invocation only through ports.

This is the concept of pure components. Next we discuss some of the features that the paradigm offers.

4 Modeling with Pure Components

This section describes a modeling technique comprising of a static and a dynamic models involving instances of *pure components*. The *publisher-subscriber* design pattern of Gamma et al.[1] is modeled through pure components.

4.1 Publisher-Subscriber

The publisher in our paradigm has a port *notify* which is connected to a port for getting notification in any subscriber instance. This port, due to the multi-casting semantics of ports, invokes appropriate methods on all its subscribers. The subscribing is affected as a result of the *connect* statements for connecting the publisher and subscriber objects using their ports.

```
(defcomponent publisher ()
   ((mylist :initform (list ()) :initarg :mylist))
   (:metaclass puritan)
   (:port notify someport))

(defgeneric notify-all-subscribers ((pub publisher))
   (:method-class component-method))

(defcomponent-method notify-all-subscribers someport ((pub publisher))
   (drop pub 'notify 456))

(defcomponent subscriber1 ()
   ((foo :initform 420 :initarg :foo))
   (:metaclass puritan)
   (:port getnotification))

(defgeneric do-action1 (sub)
   (:method-class component-method))

(defcomponent-method do-action1 getnotification1 ((sub subscriber1) a)
   (format t "I am subscriber1"))
```

```
(defcomponent subscriber2 ()
    ((foo :initform 45 :initarg :foo))
    (:metaclass puritan)
    (:port subscribe getnotification))

(defgeneric do-action2 (sub)
    (:method-class component-method))

(defcomponent-method do-action2 getnotification2 ((sub subscriber2) a)
    (format t "I am subscriber2 and i got  s from the publisher" a))

(setq sub1a (make-instance 'subscriber1 :foo 20)) ;instances
(setq sub2a (make-instance 'subscriber2 :foo 21))
(setq pub (make-instance 'publisher))


(connect pub 'notify sub1a 'getnotification1) ;connections
(connect pub 'notify sub2a 'getnotification2)

(notify-all-subscribers pub) ;notify all
```

Component Diagram. The component diagram with notations for showing ports and attachment between methods and ports is given in Fig. 7.

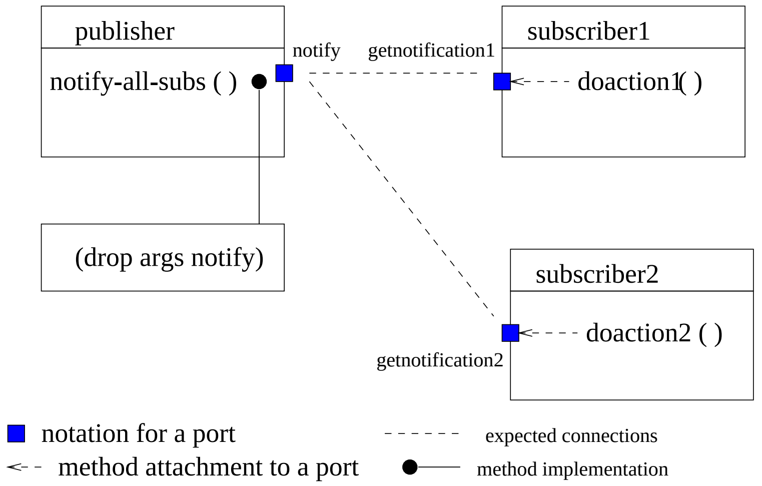

Fig. 7. Component diagram for publisher-subscriber pattern

Interaction Diagram. The interaction diagram for the above example is shown in Fig. 8. The diagram uses notation to show the interaction of *pure components* through the *ports* and *connections*. The thick vertical lines represent ports of specific component instances. Component instances are represented as objects are in a traditional interaction diagram. Dashed lines show the connection between the component instances through their ports. A directed solid line from an instance to a port denotes a *drop* and a directed dashed line from a port to an instance denotes the invocation of a method associated with the port.

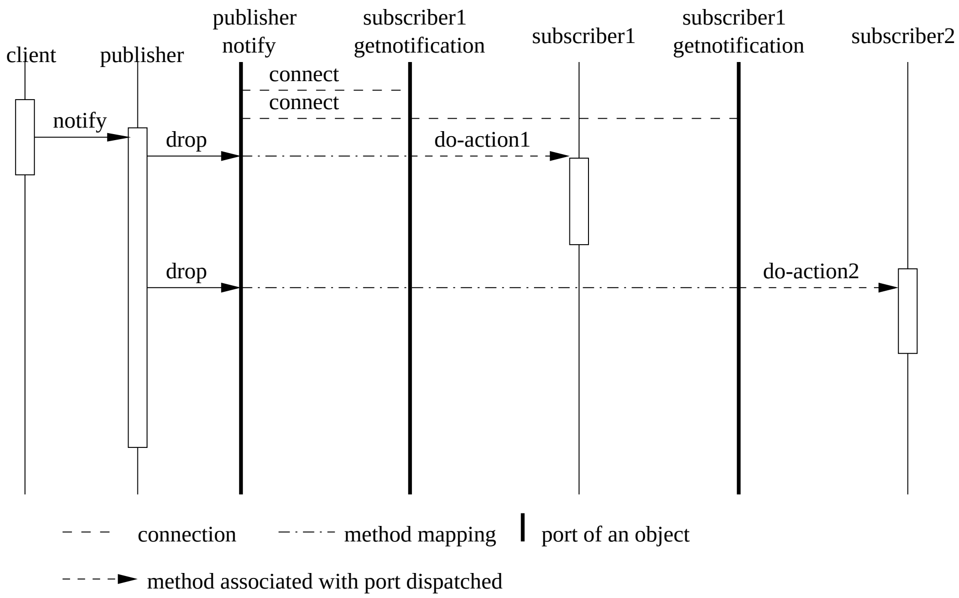

Fig. 8. Interaction diagram for publisher-subscriber pattern

5 Conclusion and Future Work

5.1 Conclusion

Our aim was the decoupling of method invocations from method definitions and removing explicit pointers to other interacting entities from a component. This decoupling helps in reconfigurability of a system. The proposed model achieves this using a level of indirection provided by the concept of *ports* in a component. Component interaction was achieved by simply connecting components through ports. Method invocation on external components takes place by *dropping* arguments on a port. *Reconfigurability* is supported by using the *connect* and *disconnect* statements and also the *dispatcher* concept. The paradigm was implemented using *Closette*, a metaobject protocol for CLOS. Implementation of two design pattern equivalents in the paradigm (publisher-subscriber and mediator) was demonstrated.

5.2 Future Work

The proposed model can be extended for distributed applications. The use of different types of connectors was not exploited although connectors are modeled as objects. The current connector can be classified as a procedure-call connector with multicastng semantics. The advantages of using an explicit connector are discussed in [14]. The paradigm is very generic and can therefore be extended to include different types of connectors and new features. A graphical interface for configuring the components should also be a useful tool.

References

1. Eric Gamma et al., *Design Patterns*, Addison-Wesley, 1995.
2. Gregor Kiczales and Jim des Riveres and Daniel Bobrow. *The Art of the Metaobject Protocol*, The MIT Press, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1995.
3. J. Magee and N. Dulay and J. Kramer. Structuring Parallel and Distributed Programs, IEEE Software Engineering Journal, 8(2), pages 73-82, March 1993.
4. James Noble and Antero Taivalsaari and Ivan Moore. *Prototype-Based Programming. Concepts, Languages and Applications*, Research and Applications, Springer, pages 247-272, 1999.
5. Richard Gabriel, *Patterns Of Software, Abstraction Descant*, Oxford University Press, New York, year 1996.
6. D. Ungar and R. B. Smith. Self: the power of simplicity, *Lisp and Symbolic Computation*, 4(3), pages 187-205, July 1991.
7. Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.
8. Graham M. Birtwistle and Ole-Johan Dahl and Bjorn Myhrhaug and Kristen Nygaard. *Simula Begin*. Studentlitteratur (Lund, Sweden), Bratt Institut fuer neues Lernen (Goch, FRG), Chartwell-Bratt Ltd (Kent, England), 1979.
9. Bertrand Meyer. *Object-Oriented Software Construction, 2nd Ed*, Prentice-Hall, Englewood Cliffs, NJ 07632, USA, second edition, 1997.
10. B. Stroustrup. *The C++ Programming Language*, Addison-Wesley, Reading, Mass., third edition, 1997.
11. P. Fritzson and V. Engelson. *Modelica — A Unified Object-Oriented Language for System Modeling and Simulation*, Lecture Notes in Computer Science, volume 1445, 1998.
12. Bruce Shriver and Peter Wegner. *Research Directions in Object-Oriented Programming*, The MIT Press, Massachusetts, Computer Systems Series, 1988.
13. Nenad Medvidovic and David S. Rosenblum and Richard N. Taylor. A Language and Environment for Architecture-Based Software Development and Evolution, *Proceedings of the 21st International Conference on Software Engineering (ICSE 99)*, pages 44-53, 1999.

14. Peyman Oreizy and David S. Rosenblum and Richard N. Taylor. *On the Role of Connectors in Modeling and Implementing Software Architectures*, Technical Report, Department of Information and Computer Science, University of California, Irvine, 1998.
15. Judy Bishop and Roberto Faria. *Connectors in Configuration Programming Languages: are they necessary?*, 1995.
16. James R. Armstrong and F. Gail Gray. *VHDL Design: Representation and Synthesis*, second edition, 2000.