How Can We Liberate Ourselves From Pointers?

Luc Bläser
Computer Systems Institute, ETH Zurich, Switzerland
blaeser@inf.ethz.ch

Abstract

Pointers or references can be identified as the root cause of many fundamental problems in current programming languages, typically resulting in unspecified object dependencies and missing hierarchical encapsulation. We therefore propose to abandon references from the language and to use expressive program relations instead. For this purpose, we have developed a programming language which is only based on hierarchical composition and interface connections.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—classes and objects

General Terms Languages

Keywords Components, Interfaces, Hierarchical composition

1. Motivation

Todays popular programming languages all suffer from a severe problem. They require the usage of explicit references or pointers, as soon as program structures become dynamic. However, the semantic expressiveness of references is much too low-level, as they allow arbitrary interlinking of object instances, without the resulting object graph having a clearly specified shape. On a daily basis, programmers encounter the various negative consequences of this deficiency:

- *Unspecified dependencies*. As references can be unrestrictedly copied between object instances, invalid references and unspecified object dependencies may be easily introduced in a program. It is usually not defined, from which location in other objects an object may be referenced.
- *Missing hierarchy*. An object is not capable of containing a dynamic structure of other objects as an encapsulated unit. The objects that ought to be encapsulated can still be intentionally or accidentally referenced from other instances, such that the program has only a flat object structure.

To eliminate the pointer problems in a more direct way, we have developed a programming language [2], in which references and pointers have been entirely replaced by more expressive structures. More specifically, the language incorporates a more general notion of objects (called components), which are based on two structural relations:

1. *Hierarchical composition*. A component is able to contain a dynamic number of components that are fully encapsulated by the surrounding instance.
2. *Interface connections*. With a dual concept of offered and required interfaces, arbitrary networks of components can be built by connecting the interfaces. The network shape is thereby always controlled by the hierarchically outer component.

Our language is different to architecture description languages [5], as it allows constructing dynamic component structures at runtime and is not limited to static component assemblies. Other component-oriented programming models [4, 1] still require ordinary references for modeling dynamic program structures.

2. Programming Language

In our component language [2], a program is defined as a component, which can be again composed of other components. A *component* constitutes a closed program unit at runtime which encapsulates state (data values and subcomponents) and behavior (interactions and functionality). Strict encapsulation is enforced, i.e. a component can only be accessed from outside via explicitly defined *interfaces*. A component may both *offer* and *require* interfaces. An offered interface thereby represents an external facet of the component itself, enabling interactions between the component and its outer environment. Conversely, a required interface specifies an interface that is to be offered by another external component. A component is statically defined in the program

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.
ACM 978-1-59593-786-5/07/0010.

```
COMPONENT House
  OFFERS Residence, ParkingSpace
  REQUIRES Water, Electricity;
  (* implementation *)
END House;
```

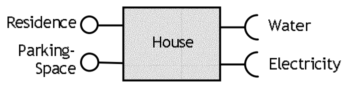

Figure 1. A component

code by a *template* (left hand side of Figure 1), which allows creating multiple instances of components at runtime (right hand side of Figure 1).

2.1 Hierarchical Composition

A component is able to contain an arbitrary number of components within its implementation scope. This is enabled by means of *variables*, which represent separate containers in which components can be stored. A variable can either store a single component (e.g. `garage` in Figure 2) or denotes a collection (e.g. `room[i: INTEGER]`), in which a dynamic number of components can be allocated. In the first case, the component is directly identified by the variable name, whereas in the latter case, a component in the collection is identified by the variable name and an index value.

At runtime, components can be created and installed within the variables. As a result, the components inside the variables are fully encapsulated and constitute sub-components of the surrounding instance. No explicit pointers or references are involved here, as a component is only accessible via its variable identifier (and index value).

Components within the same scope may also be connected forming networks. For this purpose, the required interface of a component can be connected to an interface that is offered by another component. In order to be connectable, both interfaces must have the same name (see Figure 2). A component may also connect an offered interface of its own to an offered interface of a sub-component (e.g. `ParkingSpace` in Figure 2). Analogously, the required interface of a sub-component may be connected to the required interface of the surrounding component (e.g. `Electricity` in Figure 2).

With these relations, the language permits general program structures. In fact, any hierarchy of component networks can be described, as each component may contain a set of sub-components with arbitrary interface connections.

With the hierarchical composition, components have an exactly defined deallocation time. The deletion of a component directly leads to the deletion of its sub-components. Moreover, a single component may be explicitly deleted by the programmer. In this case, the component's interfaces are automatically disconnected.

3. Conclusion

In order to eliminate the structural weaknesses of current programming languages, we have designed a component-based language which completely abandons references or pointers. Instead, the language offers more expressive program relations, which enable hierarchical compositions with guaranteed encapsulation and component networks with an accurate control of the shape and the dependencies. The language has been implemented and successfully applied to standard programming problems and demonstrated in a practical programming project, namely a traffic simulation package [3].

```
COMPONENT Room REQUIRES Entry, Electricity, Water;
END Room;

COMPONENT Garage OFFERS ParkingSpace REQUIRES Electricity;
END Garage;

COMPONENT Corridor
  OFFERS Entry REQUIRES Electricity;
END Corridor;

COMPONENT House;
  OFFERS Residence, ParkingSpace
  REQUIRES Water, Electricity;
  VARIABLE garage: Garage; corridor: Corridor; room[name: TEXT]: Room;
  BEGIN
    NEW(garage); NEW(corridor);
    WHILE new room x needed DO
       NEW(room[x]); CONNECT(Entry(room[x]), corridor);
       CONNECT(Electricity(room[x]), Electricity);
       CONNECT(Water(room[x]), Water)
    END;
    CONNECT(ParkingSpace, garage)
END House;
```

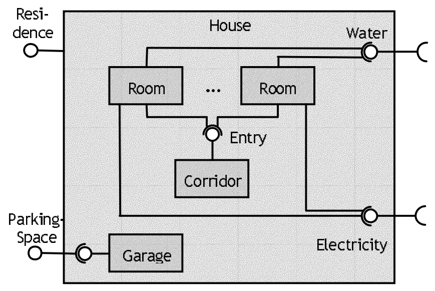

Figure 2. Hierarchical composition

Acknowledgments

I would like to thank Prof. Dr. Jürg Gutknecht for his support and advise in this project. I also gratefully appreciate the help of Dr. Felix Friedrich and Nadja Beeli for the proof-reading.

References

[1] J. Aldrich, C. Chambers, D. Notkin. *ArchJava. Connecting Software Architecture to Implementation*. Intl. Conference on Software Engineering (ICSE), May 2002.

[2] L. Bläser. *A Component Language for Structured Parallel Programming*. Joint Modular Language Conference (JMLC), September 2006, LNCS Vol. 4228, Springer Verlag, 2006.

[3] L. Bläser. *The Component Language and System*. http://www.jg.inf.ethz.ch/components.

[4] Microsoft COM. http://www.microsoft.com/com.

[5] N. Medvidovic and R. N. Taylor. *A Classification and Comparison Framework for Software Architecture Description Languages*. IEEE Transactions on Software Engineering, 26(1):70-93, 2000.