Composing Distributed Applications through Generative Communication

Tom Holvoet and Yolande Berbers
Dept. of Computer Science
K.U.Leuven, Belgium
{*Tom.Holvoet, Yolande.Berbers*}*@cs.kuleuven.ac.be*

Abstract

Component-oriented approaches to software development aim to support cost-effectiveness in building applications. Components are defined as reusable entities. In this paper, we present $\mathsf{CO^3PS}$, a software development method for building agent-based software. Agents coordinate through generative communication. We employ generative communication to define reusable $\mathsf{CO^3PS}$ components. Components are entities that export object spaces, i.e. places from which they can retrieve data objects and in which they can put data objects. Components can be reused by assembling them with other components through defining a mapping of their respective object spaces. The high degree of reusability that is achieved is due to the anonymous and uncoupled nature of generative communication.

1 Introduction

Various terms referring to the words component or composition have been introduced recently. The marketing value of such terms is undoubtedly one important reason, as these words have a connotation of "support for distribution" and "reuse" and hence for cost-effective software development. Two categories of what can generally be called component-related approaches can be distinguished.

A first category is concerned with **supporting communication** between software entities that may be distributed over several hosts in a network environment. As object technology has evolved rapidly over the past decade, recent proposals in this category focus on object-to-object communication. Components in this category are hence *objects*, while the term composition refers to the fact that distributed objects can communicate easily, i.e. with a syntax and semantics that resemble object communication in a non-distributed environment. This is the basic starting point for systems like CORBA [Obj93], the ISO reference model of open distributed processing (RM-ODP) [ISO95] and Java's RMI (Remote Method Invocation) [AG96]. These approaches heavily rely on remote references. Such systems are vulnerable for dangling references and require non-trivial naming schemes.

A second category of component-related approaches concerns **techniques for developing reusable software**, mostly referred to as "component-oriented software development". The underlying philosophy is the following. System requirements typically change frequently. Class hierarchies are the soul reuse mechanism supported by object-oriented languages, but it is obvious that this does not suffice for all reuse purposes. Components are defined as generic "entities that are designed to be composed and reused" [ND95]. Component-oriented software development methods need to define components as well as composition techniques. Composition allows to combine existing components in order to constitute more complex components or applications. Only few approaches are thoroughly discussed in the current literature. E.g. components in Darwin [MK96] (as well as in related approaches that are called "Architectural Description Languages") are entities for which input and output ports are identified. A composition of components consequently is the result from bindings between input and output ports of the composite entities of the component.

In this paper, we propose a definition of components and composition mechanisms as part of a component-oriented software development method, called $\mathsf{CO^3PS}$. $\mathsf{CO^3PS}$ is based on **generative communication**. Generative communication allows to achieve high reusability of components and system openness (systems can allow fragments of software to cooperate easily and successfully with new, a priori unknown entities).

This paper is organised as follows. In Section 2, we describe generative communication. We indicate the main advantages of this coordination model, and illustrate it. Section 3 presents principles of $\mathsf{CO^3PS}$, including definitions of agents, passive objects, agent coordination and configurations. Then we define components and composition within $\mathsf{CO^3PS}$ in Section 4. We distinguish between *static composition*, which shares its goals with so-called architectural description languages (ADLs), and *dynamic composition*, which denotes a run-time mechanism for assembling software entities. We illustrate components through an example of a master/worker setting. Finally, Section 5 presents concluding remarks.

2 Generative Communication

Generative communication is a communication paradigm that is extensively being researched in the area of coordination [Cia90, MC94, GC92]. In generative communication, communicating entities do not exchange messages directly, but through a shared data space. Concurrent entities can drop (or "generate") items in the data space, and others can retrieve items from the data space using associative matching. This way of communication inherently *uncouples* communicating processes. A sender of a data item does not directly contact another party, and a reader only contacts the shared space when it actually requires the data. The reader does not even have to exist at the time of generation. This leads to the other major advantage of generative communication: the active entities are able to communicate although they are *anonymous* to each other.

This uncoupled and anonymous communication style directly contributes to the design of parallel and distributed applications: uncoupled communication allows to abstract from details (such as identification and interface) of the entities that are cooperating. Instead, focus is on the coordination data that each entity requires and produces. Due to these advantages, coordination models based on generative communication are superior to message passing or trader-based schemes because these both rely on knowledge about a receiver's or server's identification.

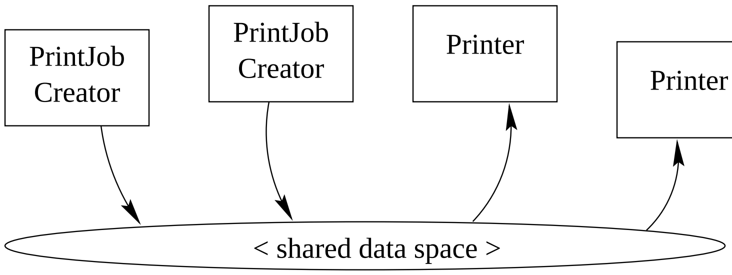


Figure 1: A producer/consumer setting using generative communication.

Figure 1 presents a situation for a producer/consumer setting using generative communication. A number of producer entities (called *PrintJobCreator*s) repeatedly generate items (print jobs) that need to be consumed (printed). Consumer entities (*Printer*s) repeatedly retrieve items from the data space and consume these. The behaviour of either entity is independent of the name or the type of the other. If, for example, another *Printer* is presented, it can seamlessly be added to the system without requiring existing entities to be changed. Also, if an existing *Printer* is eliminated, this will not affect the behaviour of the *PrintJobCreator*s, as these do not have a direct or indirect reference to it.

Linda [GC92] is a generative coordination language. It supports a minimal set of language primitives for storing and retrieving data items from a data space.

3 Computation and Coordination in CO^{3}PS

CO^{3}PS [Hol97] is an object-oriented method and execution environment for developing distributed applications, based on generative communication. Distributed applications are modelled as sets of cooperating active objects, called *agents*. Agents perform *computations* by creating and manipulating passive object. Passive objects are not shared between agents and do not have an autonomous behaviour. The CO^{3}PS coordination model can shortly be explained as follows. Each agent is attached to one or more *object spaces*. Object spaces are particular objects which may be shared amongst agents. Agents communicate by exchanging *passive objects* through object spaces. They use interface operations that are defined on object spaces to achieve this.

CO^{3}PS enhances Linda mainly in three ways:

- *object matching*: contrary to Linda, where matching of items in the shared data space is based on the contents of record fields, object matching in CO^{3}PS is based on object *types* and the *predicates* defined by the corresponding type interfaces; the items that entities (agents) can put and retrieve from object spaces are (passive) objects;
- *non-blocking multi-set operations on object spaces*: the operations defined on the object spaces (mainly insertion and retrieval of objects) are annotated with a time-out value; this is necessary to cope with system failures, which are not unlikely in open distributed systems; moreover, the operations are extended such that multi-sets of matching objects can be inserted in or retrieved from object spaces atomically.
- *multiple object spaces*: having only one object space is not feasible mainly for two reasons: scalability (for a huge set of agents, one global shared blackboard is untenable from a practical point of view) and communication structure (using one blackboard as the communication medium for multiple groups of closely related agents imposes undesirable complexity); since object spaces are the forum for communication between objects, multiple object spaces may be used for separate goals, i.e. closely related objects should use the same object space (see Figure 2); a setting consisting of several agents and object spaces is called a CO^{3}PS **configuration**.

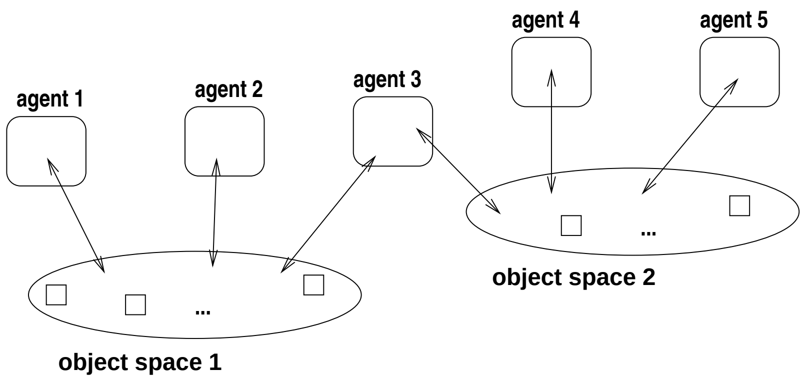


Figure 2: An example of a CO^{3}PS configuration. Object spaces are the communication forum for closely related agents. Agents can be related to more than one object space.

The language constructs that support the above object-oriented way of generative communication basically consists of the notion of PassiveObject and ObjectSpace. The CO^{3}PS coordination operations on object spaces are out, in and rd (cf. Figure 3). The operations in and rd specify multisets of matching objects to be retrieved by two parameters, namely min and max; min gives the minimal number of objects to be found in order to successfully complete the operation whereas max denotes an upper bound allowing to retrieve (small) portions of all objects of a kind. An infinite value for max allows to retrieve all currently available objects of a kind.

For a more elaborate description and examples of CO^{3}PS, the reader is referred to [Hol97].

class ObjectSpace {

Boolean out (m : Multiset ; timeout : Real)
Tries to move the objects contained in m into the object space. Returns true if the operation could be completed successfully (i.e. if all objects contained in the multiset are put in the object space); returns false if the operation could not be performed within *timeout* seconds (and hence none of the objects contained in the multiset are put in the object space).

Multiset in (o : PassiveObject ; min, max : Integer ; timeout : Real)
Tries to remove multiple objects $o'_1 \ldots o'_n$ matching the template object o from the object space and returns a multiset containing them if at least min matching objects could be found within *timeout* seconds. In this case, the multiset contains at most max objects, even if the object space contained more. If min matching objects could not be found within *timeout* seconds, the operation returns an null object.

Multiset rd (o : PassiveObject ; min, max : Integer ; timeout : Real)
Tries to return clones of multiple objects $o'_1 \ldots o'_n$ matching the template object o and returns a multiset containing them if at least min matching objects could be found within *timeout* seconds. In this case, the multiset contains at most max objects, even if the object space contained more. If min matching objects could not be found within *timeout* seconds, the operation returns an null object.

}

Figure 3: Interface specification for the ObjectSpace class: coordination operations.

4 Components and Composition based on generative communication

[Hol97] and [NM95] investigate and deduce a set of requirements for component approaches. These requirements reflect the reusability demands for component-oriented software development.

First, similar to objects, components should **encapsulate** their internal representation, and only be accessible in restrictive manners: for each component, *composition plugs* (similar to interfaces of objects) need to be identified through which the component can be reused and composed with other components. Second, compositions of components should be components themselves too, such that these can be composed later on; **higher-order components** are required for managing scaling complexity. And third, **objects** are the basic building block in object-oriented systems. Objects are therefore the first candidates for being components; a too large distortion between these two concepts is therefore unacceptable.

Components and Generative Communication

Coordination is normally understood as the modelling of the interactions and dependencies between multiple concurrent entities. One definition of coordination that is often used is *"the process of building programs by gluing together active pieces"* [GC92]. This indicates a direct link between the notions of coordination and composition. In $\mathsf{CO^3PS}$, we use generative communication as the basis for coordination as well as for composition.

Generative communication allows entities to communicate although they are anonymous to each other. Entities can be developed without requiring a direct link to other entities. In $\mathsf{CO^3PS}$, the entities (the agents) hence only depend on a set of object spaces. The reuse advantage of this characteristic is obvious: agents can be used in several different configurations without change. This principle is the basis for the definition of components and composition in $\mathsf{CO^3PS}$.

Definition of CO^3PS components

We present the following recursive definition of CO^3PS components, which we explain below. A CO^3PS components is either a CO^3PS **agent** or a **composite component**, i.e. a configuration of CO^3PS components, for which a set of object spaces is identified as the **composition plugs**.

A CO^3PS agent is an active entity which depends on a set of object spaces. When considering *agent3* in Figure 2 in isolation, it is a component depending on two object spaces, while the other agents each are a component depending on one object space. The entire configuration presented in Figure 2 is a composite component. It consists of a number agent components which are connected through object spaces. Such a configuration is defined as a mapping (in particular, an equivalence relation) of the composition plugs of the agents. In the example, this mapping relates the object space of *agent1* with the object space of *agent2* and one of the object spaces of *agent3*, and similarly it relates the second object space of *agent3* with the object space of *agent4* and *agent5*.

The CO^3PS component language

The above definition is supported in CO^3PS through a *component language*. This language is extremely simple because CO^3PS components are defined through very few principles. It suffices to support (1) the description of which object spaces an agent uses, (2) the specification of the components that a composite component consists of, and (3) the definition of an equivalence relation between object spaces of composite components.

First, we adapt the CO^3PS agent class specification such that the set of object spaces used by an agent is exported. This is achieved by a new tag, namely ObjectSpaces.

The CO^3PS component language introduces the language concept component. It allows to describe composite components. A component specification looks as shown in Figure 4. It contains the following:

- a *component name*;
- an enumeration of object spaces that this component exports as its own composition plugs, using the *ObjectSpaces* tag;
- an enumeration of components of which this component consists, using the Composites tag;
- and a definition of an equivalence relation on the object spaces of the components that this component consists of, using the *ObjectSpaceMapping* tag; the equivalence relation on object spaces is defined through a set of *bind* statements.

```
component ComponentName {

    ObjectSpaces:
        // enumeration of object spaces
    Composites:
        //enumeration of the components
    ObjectSpaceMapping:
        // defines an equivalence relation on object spaces
}
```

Figure 4: Composite component specification in the CO^3PS component language.

Examples

A few examples for single agent components can be found in Figure 5. In the upper half of this figure, partial class specifications for Agent1 and Agent3 (cf. supra) are shown. Agent1 uses one object space, Agent3 uses two. Underneath, partial class specifications are given for Worker and Master agents. A Worker agent is an agent which autonomously and continuously retrieves a task object from an object space (called agendaspace) and performs appropriate computations for this task. A Master agent continuously produces task objects into an object space (called deliverspace).

```
class Agent1 {
   ObjectSpaces: objspace;
   // internal specifications
}
```

```
class Agent3 {
   ObjectSpaces: objspace1,
                 objspace2;
   // internal specifications
}
```

```
class Worker {
   ObjectSpaces: agendaspace;
   // internal specifications
}
```

```
class Master {
   ObjectSpaces: deliverspace;
   // internal specifications
}
```

Figure 5: Examples of single agent components.

Figure 6 and 7 illustrate the component language for composite components. Figure 6 defines a composite component which results in the configuration described earlier (Figure 2). It consists of five single agent components and a suitable mapping of their object spaces.

```
Component Agent-i-App {
   ObjectSpaces: ;
   Composites:
     Agent1 agent1;        Agent2 agent2;        Agent3 agent3;
     Agent4 agent4;        Agent5 agent5;

   ObjectSpaceMapping:
     bind (agent1.objspace,   agent2,objspace,  agent3.objspace1);
     bind (agent3.objspace2, agent4,objspace,  agent5.objspace);
}
```

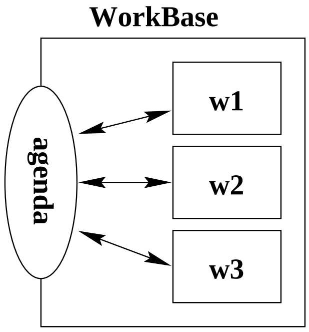
Figure 6: Composite components for the configuration in Figure 2.

Figure 7 first defines a composite component called WorkBase. A WorkBase component is a component that consists of three Worker agents which each retrieve their tasks from the same object space. This shared object space (called agenda) is exported as the composition plug of the WorkBase component. The first bind statement maps the object spaces of the three Worker agents onto each other, the second bind statement defines that the agenda object space coincides with the object space shared by the Worker agents. An graphical representation illustrates this.

The lower part of Figure 7 defines a composite component MasterWorkerApp which consists of one Master agent and one instance of the WorkBase component. The bind statement maps the deliverspace of the Master agent onto the agenda object space of the WorkBase component.

Each component can, in theory, be considered to be the specification of an application. The MasterWorkerApp component e.g. is a component which does not export any object spaces, and can therefore not be reused through composition with other components. Its only purpose is to describe the initial configuration of a Master/Worker Application. On the other hand, a component

```
Component WorkBase {
   ObjectSpaces: agenda;
   Composites:
     Worker w1, w2, w3;
   ObjectSpaceMapping:
     bind (w1.agendaspace, w2.agendaspace,
                           w3.agendaspace);
     bind (agenda, w1.agendaspace);
}
```




```
Component MasterWorkerApp {
   ObjectSpaces: ;
   Composites:
     WorkBase wb;
     Master master;
   ObjectSpaceMapping:
     bind (wb.agenda, master.deliverspace);
}
```

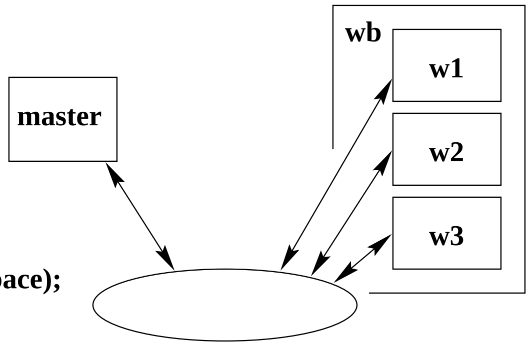


Figure 7: Composite components for the master/worker example.

which exports object spaces can be reused through composition, but might also be considered to describe an initial configuration of an application. If e.g. the MasterWorkerApp component would export the object space, it could serve as an application as well as a reusable component.

The proposed requirements for components and composition are clearly met. $\mathsf{CO^3PS}$ components are *encapsulated* entities whose behaviour can only be observed and influenced through *composition plugs*, namely a set of object spaces. Composite components can export object spaces, and as such are components too. The $\mathsf{CO^3PS}$ component language thus supports *higher-order components*. Finally, there is no distortion between the notion of $\mathsf{CO^3PS}$ agents and $\mathsf{CO^3PS}$ components. Agents are trivial components, consisting of just one entity.

5 Conclusion

This paper presented an approach for component-oriented software development based on generative communication. The approach extends the $\mathsf{CO^3PS}$ approach for agent-oriented modelling. In $\mathsf{CO^3PS}$, applications are modelled as set of active objects, agents, which *coordinate* using generative communication, i.e. through shared object containers (object spaces). The anonymous and uncoupled nature of this coordination model proofs beneficial for defining reusable components. We described a component and composition language. This language is used to specify components which can then be composed with other components. Care is taken to define components as encapsulated entities, which moreover specify a set of composition plugs, i.e. “hooks” through which components can be composed with other components. We illustrated this with a simple example of a master/worker architecture.

The presented component language is however merely a subset of the full component language. We described only those features that are necessary to explain the principles. A specification of the entire language is being prepared at this time.

The $\mathsf{CO^3PS}$ concepts of agents, passive objects, object spaces are provided in a software framework for easy extendibility. The implementation uses the Java language. For the $\mathsf{CO^3PS}$ component language, we developed a prototype preprocessor. The preprocessor translates component configurations into object space and agent construction.

The limited space prohibited us to present a complementary way for software composition, namely *dynamic composition*. The definition of components and composition we presented in this paper concerns the activity of staticly assembling portions of software in order to obtain a new component or an application. The result is a static description of a configuration, i.e. a set of agents that are connected through a number of object spaces. Dynamic composition, however, is a run-time activity for changing the configuration of an application. $\mathsf{CO^3PS}$ agents can dynamically change the object spaces it is connected to. Therefore, $\mathsf{CO^3PS}$ allows references to object spaces to be dropped into object spaces. Agents can retrieve object space references and attach themselves to the referenced object space. This allows configurations to change dynamically. $\mathsf{CO^3PS}$ supports dynamic composition by a set of language constructs similar to the language constructs for coordination.

References

[AG96] K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

[Cia90] Paolo Ciancarini. Coordination Languages for Open System Design. In *Proc. of IEEE Intern. Conference on Computer Languages*, pages 120–129, New Orleans, 1990.

[GC92] David Gelernter and Nicholas Carriero. Coordination Languages and their Significance. *Communications of the ACM*, 35(2):96–107, 1992.

[Hol97] T. Holvoet. *An Approach for Open Concurrent Software Development.* PhD thesis, Department of Computer Science, KULeuven, Belgium, December 1997.

[ISO95] ISO/IEC JTC1/SC21/WG7. Reference Model of Open Distributed Processing. Draft International Standard ISO/IEC 10746–1 to 10746–4, Draft ITU–T Recommendation X.901 to X.904, May 1995.

[MC94] Thomas W. Malone and Kevin Crowston. The Interdisciplinary Study of Coordination. *ACM Computer Surveys*, 26(1):87–119, 1994.

[MK96] J. Magee and J. Kramer. Dynamic Structure in Software Architectures. In D. Garlan, editor, *Proc. 4th ACM SIGSOFT Conf. on Foundations of Software Engineering*, volume 21:6 of *ACM SIGSOFT Software Engineering Notes*, pages 3–14, 1996.

[ND95] Oscar Nierstrasz and Laurent Dami. Component–Oriented Software Technology. In O. Nierstrasz and D. Tsichritzis, editors, *Object–Oriented Software Composition*, chapter 1, pages 3–28. Prentice-Hall, 1995.

[NM95] O. Nierstrasz and T. Meijler. Requirements for a Composition Language. In P. Ciancarini, O. Nierstrasz, and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, volume 924 of *Lecture Notes in Computer Science*, pages 147–161. Springer-Verlag, Berlin, 1995.

[Obj93] Object Management Group. The Common Object Request Broker: Architecture and Specification. OMG Document Number 93.12.43, 1993.