A Component Language for Structured Parallel Programming

Luc Bläser

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

Abstract. Current programming languages are still underdeveloped for the construction of well-structured concurrent software systems. They typically impose many unnecessary and unacceptable compromises and/or workarounds due to a multiplicity of different suboptimal concepts. With regard to object-orientation, one can identify references, methods and inheritance as such inappropriate constructs.

To overcome this unfavourable situation, we have designed and implemented a substantially new programming language which integrates a general component notion. Three fundamental relations govern components in this language: (1) hierarchical composition, (2) symmetric connections with a dual concept of *offered* and *required* interfaces and, (3) communication-based interactions. With the use of various examples, the advantage of the new component language is demonstrated in this paper.

1 Motivation

The current trend within the field of software engineering is steadily evolving towards programming languages which possess an increasing number of different and unfortunately, counterproductive concepts. This growing conceptual incoherence often implicates such high complexity, that it decisively limits the flexible construction of structured parallel programs. With regard to the current most prevalent object-oriented programming paradigm, we are confronted with three fundamental problems:

- *References*
  References (or pointers) form semantically very weak constructs for describing relations between dynamically created object instances. Arbitrary interlinking of object instances is therewith promoted, leading to an object graph of non-hierarchical shape[1]. Clear program structures and general encapsulations remain unsupported: any abstraction that consists of a dynamic structure of sub-elements is not adequately representable as a hierarchically composed object. Instead, this has to be forcibly modelled as a reference-linked conglomerate of elementary object instances, constituting an undifferentiated part in the common overall and flat object graph. As a con-

[1] C.A.R. Hoare unequivocally criticizes the unstructured nature of references and calls their introduction in high-level programming languages a step backwards [17, page 20].

sequence, incautious reference copying may quickly lead to incorrect program dependencies (aliasing problems [16, 4, 11, 8, 22, 2]). Moreover, object exchangeability is strongly impacted by dependencies of outgoing object references which are unspecified in object interfaces[2].

- *Methods*
  Methods fail the realization of a true message passing paradigm, as they in fact only constitute procedures (with an implicit reference to the containing object). An object is not capable of maintaining an arbitrarily long state-full interaction with multiple clients individually, but can only hold a client-specific context during a method invocation[3]. The pattern of a method for a client-specific interaction is however oversimplified, having only one parameterised input followed by one possible output, with generally only one value. Methods additionally obstruct concurrency by blocking the invocator during their entire execution, instead of running at the expense of the actual containing object.
- *Inheritance*
  The main object-oriented mechanism for type polymorphism, known as inheritance, enforces a groundless hierarchisation and classification of object types at compile-time. Unlike a symmetric polymorphism, objects can not be represented by multiple, equally important facets, without artificially preferring some facets as sub-types of others. Inheritance also unsuitably combines the two antagonistic concerns of polymorphism and code reuse, often resulting in mutual imports of different classes. A special object class, which needs to be inherited from a general class for the purpose of type polymorphism, should not be obligated to also inherit the general implementation of the super-class, as the special class' code is naturally more specific than that of the general class[4].

This unfavourable situation demands a total revision of the conceptual basis of current programming languages. We are challenged to design new languages, which base on a new more powerful paradigm that uniformly enables structured, dynamic, and safe software development. Clearly, this requires the liberation of the language concepts from the often unreasonable close binding to a concrete machine model. Instead, there is a need for real high-level programming languages, which are still effectively implementable on different computer platforms.

In order to achieve this ideal, we have designed and implemented a substantially new programming language, which integrates a general high-level component notion. Three simple but fundamental relations govern components in this language: (1) hierarchical composition without use of explicit pointers, (2) symmetric connections with a dual concept of *offered* and *required* interfaces and, (3) communication-based interactions. The new *component language* takes a completely different path in com-

[2] Every element of public visibility in the object may be considered as part of the object's interface.

[3] The iteration over a collection stands for a client-individual state-full interaction that can not be accurately expressed with methods (cf. 3.2).

[4] Clearly, the example of a rectangle and square shows this contradiction: a square is a geometrical special-case (modelled as a *sub-class*) of a rectangle but on the other hand, should not inherit the general rectangle implementation (with the two variables *length* and *width*).

parison to existing component models, architecture description languages, and object structure specification models (see Section 5). As innovation, it provides a fully-fledged programming language, which only features high-level concepts for the implementation of components. The component language inherently abolishes the fundamental deficiencies of current programming models and offers the following attractive features:

- *Hierarchical encapsulation*
  A component is able to contain any (static or dynamic) structures of components and program logic of any complexity. The hierarchically contained components and the relations among them are thereby fully encapsulated and exclusively managed by the surrounding component.
- *Expressive structural relations*
  All structures of components are described by semantically rich relations, such that classical references (and pointers) can be entirely abandoned without loss of expressiveness: each component contains its own arbitrary network of sub-components. This prohibits uncontrolled program dependencies (such as aliasing problems).
- *Intrinsic concurrency*
  Concurrency inherently results from the language model, in which all components run fully autonomously and have their own intrinsic activities. Components only interact via bidirectional message communications (with non-blocking message sending).
- *Unrestricted polymorphism*
  Components can be represented by an arbitrary set of independent interfaces, activating unrestricted symmetric polymorphism in total separation from implementation reuse. A new type description ensures the correct handling of polymorphic components.
- *Interoperability*
  Although the component language is designed for general purposes (except machine-close programming) and common programs are entirely developable in components, the language also permits safe interoperability due to the guaranteed encapsulation. Terminal components, which do not contain sub-components, may be just as well implemented in any programming language, such as for the purpose of machine-specific implementations.

1.1 Contributions

The contributions of this paper can be summarised as follows:

- The presentation of a new programming language with an integrated general component notion for structured parallel programming.
- A comparison of the new language with classical object-orientation, showing the advantage by means of practical examples.
- The description of a complete implementation of the programming language, comprising compiler and runtime system.

The remainder of the paper is organised as follows; Section 2 presents the concepts of the new programming language and explains them by means of illustrative examples.

Section 3 shows practical examples of the new language and compares them with object-orientation. Section 4 describes the implementation of the programming language and also gives an experimental evaluation of the system. Section 5 discusses related work, which is finally followed by a conclusion.

2 Component Language

The new programming language follows the principle that any program forms a component which may be constructed again from an assembly of components and so on. With this paradigm of stepwise refinement, complex systems can be built with abstract program elements that hide detailed logic from a higher abstraction level.

A *component*[5] constitutes a closed program unit (*black box*) that encapsulates an arbitrary assembly of sub-components, together with runtime state and behaviour. Components are only allowed to have external program dependencies over explicitly defined interfaces. An *interface* represents an external facet of a component and thus establishes an explicit interaction point between the component and its exterior environment. Each component *offers* an arbitrary number of own interfaces and also *requires* an arbitrary number of foreign interfaces that belong to other external components[6].

By way of a first example, let us consider a standard house, which has the external facets of a residence and a parking space, requiring both electricity and water supplies from outside. The house may be described as a component called *StandardHouse*, which offers both a *Residence* and *ParkingSpace* interface (see Fig. 1). In addition, the house requires the foreign *Electricity* and *Water* interfaces from other external components. Clearly, all interfaces of the component have equal rights, i.e. there is no artificially preferred interface. With regard to the example, this means that the characterizations of a residence and parking space are equally important facets of the house.

```
INTERFACE Residence; (* … *)
INTERFACE ParkingSpace; (* … *)
INTERFACE Electricity; (* … *)
INTERFACE Water; (* … *)

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

Fig. 1. A component

[5] A component here always means a runtime instance of a component template.
[6] A variety of other component definitions can be found in [26, Chapter 11].

Arbitrarily many *component instances* (also simply called *components*) can be created from the same *component template* (also called *component type*[7]). In the example above, the program describes the component template, which can in turn be used to create as many house component instances as needed. One such possible instance of a house is depicted by the diagram in Fig. 1.

The component language is based upon three fundamental relations between components:

- *Hierarchical composition*
  Each component can be hierarchically composed, by containing an arbitrary assembly of component instances. The contained sub-components are fully encapsulated by the surrounding super-component.
- *Interface connections*
  An arbitrary network of components can be built by connecting the required interfaces of components to corresponding offered interfaces of other components. A component only constructs the network of its sub-components.
- *Communication-based interactions*
  Components can interact via interfaces by message communications. An individual communication channel is maintained between a component, which offers an interface, and each component, which uses the interface[8].

The component notion is designed to cover any conceivable encapsulated program unit and to enable higher generality than the classical component abstractions of objects and modules. For that reason, the general components establish the sole building units of the language.

2.1 Component Instances

Component instances must always be declared in the program scope of their containing super-component. The declaration of an instance requires a description of the corresponding component type (component template), in order to ensure the correct handling of instances. The concrete component type is one possibility for such a description. For example, *house1* and *house2* can be declared as two instances of the *StandardHouse* component type:

```
house1, house2: StandardHouse
```

In many cases, it is however necessary to declare component instances without statically fixing a specific type. Therefore, as another possibility, a component instance is also declarable in abstract terms, by simply postulating a set of offered and required interfaces. The example below shows such an abstract declaration of a *building* component instance, with the postulated offered interfaces *Residence* and *ParkingSpace*, and the required interfaces *Electricity* and *Water*.

```
building: ANY(Residence, ParkingSpace | Electricity, Water)
```

[7] A component instance has only one type, i.e. the concrete template from which it is created.
[8] Notably, the communication between components is fully symmetric and does not entail "inverse programming" by means of event-orientation.

Using this declaration, the component instance can be of *any* component type that fulfils the following requirements:

1. The component type *offers at least* the interfaces which are postulated as offered by the declaration (i.e. *Residence* and *ParkingSpace).* These interfaces are always guaranteed to be provided by the declared component instance.
2. The component type *requires at most* the interfaces which are postulated as required by the declaration (i.e. *Electricity* and *Water).* These interfaces have to be provided by the environment of the declared component instance, before the component's offered interfaces can be used.

Applying the rules above, the following *townHouse* component may well be of the *StandardHouse* type. Conversely, the *oldHouse* component can not represent a *StandardHouse* as no required *Electricity* interface is postulated.

```
townHouse: ANY(Residence | Electricity, Water, CentralHeating);
oldHouse: ANY(Residence | Water)
```

A static declaration of component instances is not always applicable as in some cases, the number of component instances may be determined only at runtime. Hence, it is also possible to declare a dynamic *collection* of component instances with the same type description. An *index*, qualified by a list of comparable data values, thereby allows the dynamic identification of a component within the collection. For example, the following declaration defines a collection of components of the *StandardHouse* type, requiring a street number and name to identify an instance.

```
house[number: INTEGER, street: TEXT]: StandardHouse
```

With this declaration, the following component instances (amongst others) may be accessed.

```
house[12, "Market Street"]    house[3, "First Avenue"]    house[100, "Grand Boulevard"]
```

2.2 Hierarchical Composition

A component can be hierarchically composed, by containing an arbitrary static or dynamic number of sub-components. The sub-components are fully encapsulated and exclusively managed by the surrounding super-component, such that the inner components are completely invisible and inaccessible outside the super-component.

The program below delineates a hierarchical composition with the example of a *StandardHouse* component, which contains a garage and two floors as sub-components (see Fig. 1). In this language, variables enable hierarchical compositions by representing *separate containers*, in which a component instance with a compatible type can be stored.

```
COMPONENT StandardHouse OFFERS Residence, ParkingSpace REQUIRES Electricity, Water;
  VARIABLE garage: StandardGarage; groundFloor, firstFloor: ANY(Rooms | Electricity, Water);
  BEGIN
   NEW(garage); NEW(groundFloor, Floor); NEW(firstFloor, Floor)
END StandardHouse;
```

As a variable is empty by default, a component instance has to be created within it by the NEW-statement. If an abstract type description is declared for the variable (ANY-construct), the component type has to be explicitly specified as second parameter (see the two last NEW-statements in the example above).

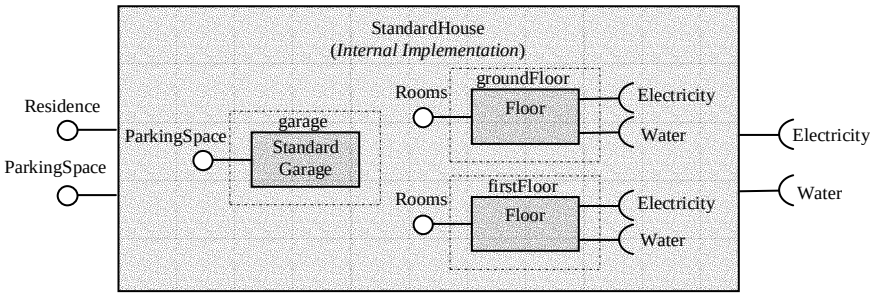

Fig. 2. A hierarchical composition of components

Naturally, a variable is also capable of storing a dynamic collection of component instances:

```
VARIABLE room[number: INTEGER]: HotelRoom;
FOR i := 1 TO N DO NEW(room[i]) END
```

Variables are only defined locally in a program scope, such that they directly imply a hierarchical lifetime dependency between the surrounding instance and the internal components.

2.3 Component Networks

Components systematically decompose programs into separated logical parts, with precisely defined dependencies in the form of offered and required interfaces. Networks of component instances can be built by explicitly connecting each required interface to one with an equal name which is offered by another component. The following example of a small city demonstrates the construction of such a network of component instances. By means of the CONNECT-statement, the required *Water* interface of *house1* is for instance connected to the offered *Water* interface of *river1*. (The offered interface is thereby implicitly defined by the first argument.) The resulting component network is visualised in Fig. 3.

```
COMPONENT HydroelectricPowerPlant OFFERS Electricity REQUIRES Water; (* ... *)
COMPONENT River OFFERS Water; (* ... *)

VARIABLE
  house1, house2: StandardHouse;
  powerPlant: HydroelectricPowerPlant;
  river1, river2: River;
BEGIN
  NEW(house1); NEW(house2); NEW(powerPlant); NEW(river1); NEW(river2);
  CONNECT(Water(house1), river1); CONNECT(Electricity(house1), powerPlant);
  CONNECT(Water(house2), river2); CONNECT(Electricity(house2), powerPlant);
  CONNECT(Water(powerPlant), river2)
```

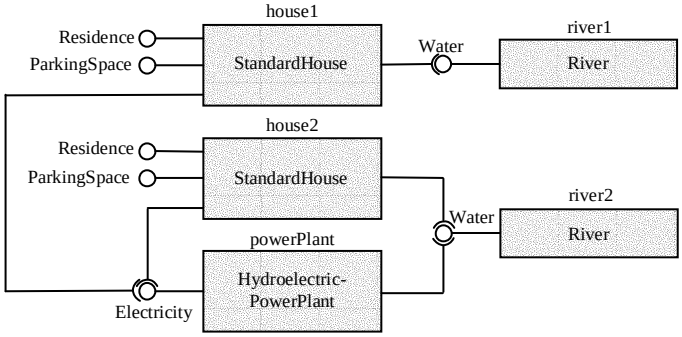

Fig. 3. A component network

Component networks can of course also be constructed with a dynamic number of component instances, as illustrated by the following program[9].

```
VARIABLE
  house[postalAddress: TEXT]: StandardHouse;
  powerPlant: HydroelectricPowerPlant;
  river[number: INTEGER]: River;
BEGIN
  FOR n := 1 TO N DO NEW(river[n]) END; (* N >= 1 *)
  NEW(powerPlant); CONNECT(Water(powerPlant), river[1]);
  REPEAT
    location := postal address of the new house;
    NEW(house[location]); CONNECT(Electricity(house[location]), powerPlant);
    n := number of nearest river;
    CONNECT(Water(house[location]), river[n])
  UNTIL no free building site available
```

Furthermore, a component may also *redirect* the implementation of its own offered external interfaces to its sub-components. For this purpose, an offered external interface (e.g. *ParkingSpace* of the *StandardHouse* below) can be connected to an offered interface with the same name that belongs to a sub-component (e.g. *garage*). Analogously, a required interface of a sub-component (e.g. the *Water* interface of the *groundFloor*) is also connectable to a corresponding interface, which is required by the super-component from outside.

```
COMPONENT StandardHouse OFFERS Residence, ParkingSpace REQUIRES Electricity, Water;
  VARIABLE garage: StandardGarage; groundFloor, firstFloor: ANY(Rooms | Electricity, Water);
  BEGIN
    NEW(garage); NEW(groundFloor, Floor); NEW(firstFloor, Floor);
    CONNECT(ParkingSpace, ParkingSpace(garage));
    CONNECT(Electricity(groundFloor), Electricity); CONNECT(Water(groundFloor), Water);
    CONNECT(Electricity(firstFloor), Electricity); CONNECT(Water(firstFloor), Water)
END StandardHouse;
```

Fig. 4 depicts the corresponding connections for the example above. As can be seen, hierarchical composition inherently enables implementation reuse. The *Standard-House* component can be flexibly built by integrating the existing *StandardGarage* implementation as a sub-component and by redirecting the *ParkingSpace* interface

[9] The elementary statements of the language are similar to the Oberon language [30, 31].

correspondingly. In contrast to object-oriented inheritance, the concerns of reuse and polymorphism are fully separated here.

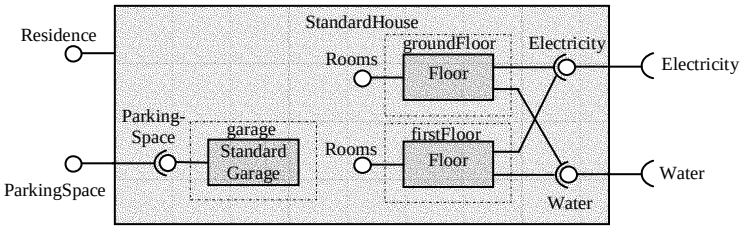

Fig. 4. Redirected interfaces

In the preceding examples, the pointer issue of ordinary programming languages is overcome: interface connections can arrange arbitrary component networks, which are always fully encapsulated by the surrounding component. This is due to the following two important distinctions:

1. A connection only constitutes a link which is exclusively set and controlled by the surrounding component, whereas a pointer (and a classical reference) forms a data value that can be freely copied from one to another object.
2. A connection establishes a symmetric link between a required and an offered interface, whereas a reference/pointer asymmetrically links a target from the reference holder and may not be visible outside the holder.

2.4 Communication-Based Interactions

Interfaces enable arbitrarily general communication-based interactions between components. Two components, which are connected by a required and offered interface, can communicate over the interface by bidirectional message exchange. The feasible sequences of message transmissions during the communication have to be explicitly defined by a protocol in the interface. As an example, the *HotelService* interface below describes the protocol for the communication between a component, which offers this interface, and an external component, which uses it (see the scenario in Fig. 5).

```
INTERFACE HotelService;
 {
   IN CheckIn
   (
     OUT AssignedRoom(number: INTEGER)
     { IN EnterRoom IN ExitRoom }
     IN CheckOut OUT Bill(price: INTEGER) [ IN DirectPayment(m: Money) ]
   | OUT FullyBooked
   )
 }
END HotelService;
```

A protocol is specified as a regular expression in the Extended Backus Naur Formalism (EBNF) [29][10]. The symbols in the protocol denote messages that are exchanged during the communication. Each message has a declared transmission direction (either IN or OUT), an identifier (e.g. *CheckIn*), and an optional list of parameters (e.g. *number*). The IN-direction defines that a message is sent to the component offering the interface, while the OUT-direction characterises the opposite direction of transmission. According to this, the communication protocol of the *HotelService* interface can be understood as the temporal series of messages outlined in Fig. 5.

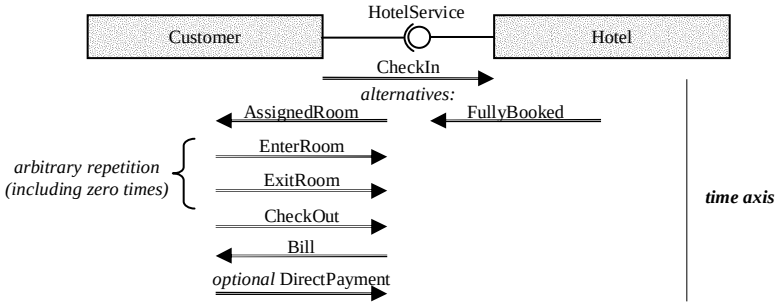

Fig. 5. Message communication via an interface

The parameters of a message represent component instances that are carried within a message. Transmitted instances are always sent as *copies* which have the same internal state and network of sub-components like the original (*deep copy*), and can in turn be safely plugged into the receiver. Naturally, really huge instances (e.g. files) should not to be transmitted as copies but should be rather represented by *unique identifiers* (e.g. file descriptors or invariant file path expressions). Such identifiers however do not form inbuilt language constructs (such as classical pointers) but have to be explicitly defined by the programmer itself, using normal data values or components. Consequently, a unique identifier can be utilised to interact (via connected interfaces) with the component that contains the actual huge instance (e.g. with the file system).

An offered interface of a component can be used in parallel by all the components which are connected to the corresponding interface, as well as by the containing super-component itself. The component which offers the interface plays the role of the *server* of the interface, whereas the other components which use the interface act as *clients* of this interface. For each client of an interface, the server automatically maintains a separate state-full[11] communication channel. Hence, some *Customer* components may simultaneously perform their individual hotel check-in, while other

[10] In EBNF, a concatenation of expressions represents a sequence, square brackets [ ] indicate an optional expression, curly brackets { } describe a repetition of zero or arbitrary times, and a vertical bar | denotes an alternative between two expressions. By default, concatenation has a stronger binding than an alternative. The default binding order can be explicitly changed with round brackets ( ).

[11] State-full means that the component saves the context for the interaction with each individual client.

clients are in another state of communication with the same *Hotel* instance (see Fig. 6).

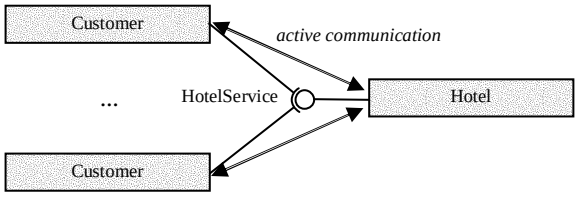

Fig. 6. Multiple parallel client communications

The following program code sketches the implementation of a communication between a *Customer* and a *Hotel* component. The *Hotel* component contains an implementation block for the offered *HotelService* interface. This implementation block is automatically incarnated *as a separate process for each client* and runs as an individual *service agent* for the client. Alternatively, the *Customer* component may directly communicate via its required interface.

```
COMPONENT Customer REQUIRES HotelService;
BEGIN
  HotelService!CheckIn; (* send message *)
  IF HotelService?AssignedRoom THEN(*receive test*)
    HotelService?AssignedRoom(n) (*accept message*)
    (* … *)
  ELSE (* fully booked *)
    HotelService?FullyBooked (* accept message *)
  END
END Customer;
```

```
COMPONENT Hotel OFFERS HotelService;
  IMPLEMENTATION HotelService;
  BEGIN
    WHILE ?CheckIn DO {EXCLUSIVE}
      ?CheckIn; (* accept message *)
      IF (*free room*) THEN !AssignedRoom(n)
      ELSE !FullyBooked END
    END
  END HotelService;
END Hotel;
```

The send statement (denoted with "!"), delivers a message to the other communication side, by filling the message with copies of the specified parameter arguments. A copy forms an identical clone of the original, such that the clone contains the same internal state, which includes the network of sub-components. These internal components are again recursively copied. Conversely, the receive statement (denoted with "?") awaits the arrival of a specific message from the other communication side and accepts the message on arrival. The contained component instances of the received message are eventually assigned to the corresponding variables, which are specified as parameter arguments. A receive statement blocks the execution as long as the message is not received. The receive-test function (an expression denoted with "?")[12], tests whether a specific message can be received from a specific interface by first awaiting any message input. The receive-test function hence blocks the execution until the arrival of any message from the interface but does not yet accept the message nor assign the message parameters[13].

Within the implementation block, the send- and receive-statements without specified interface directly refer to the corresponding client, which is served by the block. Conversely, for the communication in the role of a client, the interface has to be explicitly specified.

[12] Notably, a receive-test function is uniquely distinguishable from a receive-statement, as it forms a syntactical expression and not a statement.
[13] Additionally, there is also a non-blocking INPUT-function to check the arrival of a message.

It is dynamically checked that all required interfaces of a component are connected when a communication is initiated via one of its offered interface. During a communication between a client and server, all messages have to be sent and received according to the defined protocol. The fulfilment of the protocol is dynamically monitored for each communication, and in the case of a violation, a runtime error is generated. When a client is disconnected from a component, the implicit END message (without parameters), is automatically delivered to the server side and may be optionally accepted by the server.

In the course of the subsequent application of the component language, some of the aforementioned elements for component implementations will be explained in more detail when required. Those, who desire a complete specification of the component language, are referred to the language report [9].

3 Examples

This section illustrates practical examples of the component language, by contrasting them to corresponding object-oriented solutions.

3.1 Producer-Consumer

The first example demonstrates a *producer-consumer* scenario, where both producer and consumer autonomously interact in parallel with a common bounded buffer.

```
COMPONENT Producer REQUIRES DataAcceptor;
  VARIABLE i: INTEGER;
  BEGIN FOR i := 1 TO 100000 DO DataAcceptor!Element(i) END
END Producer;

INTERFACE DataAcceptor;
  { IN Element(x: INTEGER) }
END DataAcceptor;

COMPONENT Consumer REQUIRES DataSource;
  VARIABLE i: INTEGER;
  BEGIN WHILE DataSource?Element DO DataSource?Element(i) END
END Consumer;

INTERFACE DataSource;
  { OUT Element(x: INTEGER) }
END DataSource;

COMPONENT BoundedBuffer OFFERS DataAcceptor, DataSource;
  CONSTANT Capacity = 10;
  VARIABLE a[position: INTEGER]: INTEGER; first, last: INTEGER; finished: BOOLEAN;

  IMPLEMENTATION DataAcceptor;
  BEGIN
    WHILE ?Element DO {EXCLUSIVE}
      AWAIT(last-first < Capacity); ?Element(a[last MOD Capacity]); INC(last)
    END;
    BEGIN {EXCLUSIVE} finished := TRUE END
  END DataAcceptor;
```

```
IMPLEMENTATION DataSource;
BEGIN
  REPEAT {EXCLUSIVE}
    AWAIT((first < last) OR finished);
    IF first < last THEN !Element(a[first MOD Capacity]); INC(first) END
  UNTIL finished
END DataSource;

BEGIN first := 0; last := 0; finished := FALSE
END BoundedBuffer;
```

In the previous example, the component body of the *BoundedBuffer* initialises the buffer, before interactions over offered interfaces are accepted. The server-side processes (*service agents*) of the offered interfaces are internally synchronised by using an exclusive *monitor lock* on the component instance, in combination with AWAIT-statements. An AWAIT-statement blocks the execution until the fulfilment of a local condition, by temporarily releasing the monitor lock. This monitor-oriented synchronization is only applicable inside the component instance, and forms a supplement to inter-component interactions, which are merely communication-based. The consumer-producer program may consequently be set up as follows (see Fig. 7):

```
COMPONENT Simulation;
  VARIABLE buffer: BoundedBuffer; producer: Producer; consumer: Consumer;
BEGIN
  NEW(buffer); NEW(producer); NEW(consumer);
  CONNECT(DataAcceptor(producer), buffer); CONNECT(DataSource(consumer), buffer)
END Simulation;
```

Producer and consumer immediately start to interact with the buffer, when the *Simulation* is created and the components have been appropriately connected. Naturally, one can also connect multiple producers and multiple consumers to the same buffer.

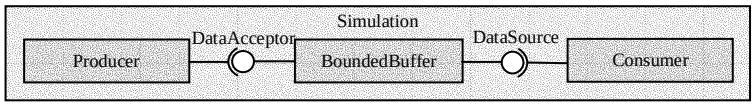

Fig. 7. Producer-consumer scenario

In object-orientated languages, such a scenario entails the explicit incarnation of *threads*, which run as concurrent procedural executions on the passive objects. Concurrency is therewith not only poorly supported as a secondary programming element (mostly provided by a separate library) but thread interactions are also only insufficiently describable. Threads may only interact implicitly by operations on shared resources, whereas the autonomously running components of our language interact in a clearly defined way by bilateral message exchange according to a formal protocol.

3.2 Digital Library

By way of a second example, we program a digital library which contains a dynamic collection of books. In the library, generic books with the offered *Book* interface can

be stored. The library is usable in parallel by an arbitrary number of connected customer components (see Fig. 8), which may request digital copies of books or may also list the book catalogue. Book references are directly modelled as what they really are: unique identities in the form of international standard book numbers (ISBNs). These real references do not involve any specific language concept but only form self-defined identifiers of component instances. Hence, real references imply neither a direct access link nor an existence guarantee. An identified book can be transmitted as a copy within a message from the library to the corresponding customer. The program code for the digital library is:

```
INTERFACE Library;
  { IN RequestBook(isbn: TEXT) (OUT Book(b: ANY(Book)) | OUT Unavailable)
  | IN ListCatalogue { OUT BookReference(isbn: TEXT) } OUT EndOfList }
END Library;

COMPONENT DigitalLibrary OFFERS Library;
 VARIABLE book[isbn: TEXT]: ANY(Book);

 IMPLEMENTATION Library;
 VARIABLE isbn: TEXT; b: ANY(Book);
 BEGIN
  WHILE ?RequestBook OR ?ListCatalogue DO
   IF ?RequestBook THEN {EXCLUSIVE}
    ?RequestBook(isbn);
    IF EXISTS(book[isbn]) THEN !Book(book[isbn]) ELSE !Unavailable END
   ELSE {SHARED}
    ?ListCatalogue; FOREACH isbn OF book DO !BookReference(isbn) END; !EndOfList
   END
  END
 END Library;
END DigitalLibrary;
```

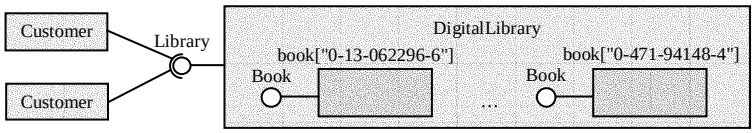

Fig. 8. Encapsulated library

Again, a few explanatory remarks may be helpful. The books in the library are stored within a dynamic component collection (cf. Section 2.1). To identify the contained instances in the collection, ISBNs are used as indexes. The inbuilt EXISTS-function tests whether a defined element is contained in the dynamic collection. If present, a copy of the appropriate book is sent. Note that the case of an inexistent book can be accurately communicated by an alternative message (named *Unavailable*), whereas in object-orientation, an artificial *null* reference often represents this case. The state-full process of listing the book catalogue, involves a shared lock of the library, permitting concurrent iterations by other users. During iterations, any modification is however prevented by exclusive locks. The FOREACH-statement allows the iteration over all instances in a collection, where each iteration step assigns a valid index to the specified iteration variable.

3.2.1 An Object-Oriented Library as Contrast

Unlike our language, an object-oriented program can not accurately describe the encapsulation of dynamic object structures inside other objects, as object-orientation does not feature a hierarchical composition relation. Therefore, an object-oriented language can not guarantee the encapsulation of books in the library but compels the programmer to allocate the internal books of the library as normal objects in the system-wide flat object graph. Very cautious programming is then required to prevent passing out references to internal books of the library in error. The following object-oriented program illustrates this situation:

```
class Book {
  string isbn; string content; Book[] references;
  void Annotate(string note) { content += note; }
}

class Library {
  Book[] books;
  Book RequestBook(string isbn) {
    for (int i = 0; i < books.Length; i++)
      { if ((books[i] != null) && (books[i].isbn == isbn)) { return books[i].Clone(); } }
    return null; /* null means unavailable */
  }
}
```

Analogous to the component-oriented program, the requested book objects are also transferred as copies between the library and the customer, as the client could otherwise modify the original book in the library. However, despite this precaution, the (directly or indirectly) referenced books in the library may then still be incorrectly accessed by an external customer (see following program fragment and also Fig. 9).

```
class Customer {
  Library library;
  void IncorrectUse {
    Book book = library.RequestBook("3-468-11124-2");
    Book x = book.reference[0];
    read(x.content); /* forbidden reading use of an internal book of the library */
    x.Annotate("personal note"); /* forbidden modifying use of an internal book of the library */
  }
}
```

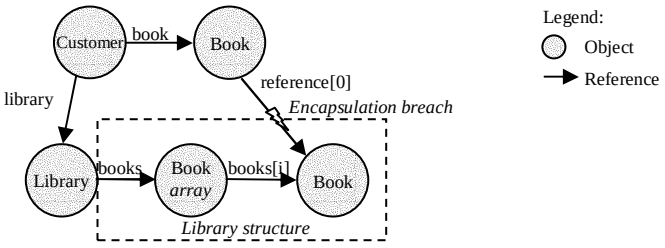

Fig. 9. Incorrect referencing

This demonstrates how vulnerable object-oriented programs are, by the fact that references can conceptually link arbitrary objects in the system and can be freely copied around. Hence, it may be argued that object-oriented references ought not to be used to represent book references in this example. Another approach of only

passing *read-only* references [22], does not give any sustainable solution either, since books may still be read without permission.

Catalogue listing is also only inadequately realizable in object-orientation, because the client-individual iteration process has to be forcibly outsourced to an artificial *iterator* object. As a consequence, the external iterator has to store then a reference (or other specific information) that directly breaks into the internal library structure (see Fig. 10). (This encapsulation breach is often considered as a counter-example for the proposed object-oriented encapsulation mechanisms [11, 22].)

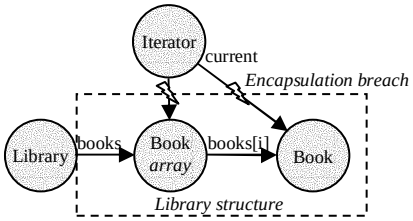

Fig. 10. Iterator object

4 Language Implementation

The presented component language has been completely implemented, comprising a compiler and runtime system, which are based on the Bluebottle operating system [10, 23]. The runtime system is designed as a stack-based virtual machine, supporting an intermediate language that consists of a sensibly selected combination of both primitive functionality (e.g. integer addition) and more complex functionality (e.g. message sending and receiving). These complex instructions directly correspond to fundamental high-level language abstractions. The compiler generates the intermediate code, which is in turn automatically transformed to the backend machine code by the virtual machine. Backend code generation is only initiated at the time when the intermediate code is loaded.

For hierarchical composition, component instances are dynamically organised in the linear heap memory with appropriate memory indirections. An internal data structure automatically manages an indexed collection of component instances. Here, an adaptive data structure may be reasonable, e.g. a simple linear list for small collection sizes and a B-tree for larger sizes. Due to the hierarchical lifetime dependencies of compositions, automatic garbage collection for memory-safe runtime management is no longer needed. Components can be directly de-allocated on the disposal of the super-component, without suffering extensive (and generally system-blocking) garbage collection.

High and efficient parallelism is most critical for the adequate runtime support of component instances and their internal processes. For this purpose, the Active Object technology [23] of the Bluebottle operating system is advantageous, as it provides particularly light-weighted parallelism with low-cost context switches. Of course, there is still much potential and need for further improvement of concurrency.

The communication between two components is implemented by an internal bidirectional message channel. These channels have bounded buffer sizes, to avoid

dynamic memory allocations on message sending. The communication protocol is dynamically monitored by using a finite state machine, that is automatically generated by the backend compiler from the protocol specification.

Table 1 gives an impression of the system's performance and scalability by means of experimental measurements with three test applications (available at [9]): (1) a producer-consumer scenario with 100,000 exchanged elements, (2) a small city simulation (as in Section 2) with 100 houses, each consuming 1,000 units of water and electricity, and (3) a large city simulation with 1,000 houses. Whereas the small city simulation only involves about 500 components and 300 processes, the large city requires more than 5,000 components and 3,000 processes. The results are first compared to analogous programs written in Active C# [13] and to a Windows implementation of AOS (called WinAOS [12]). On a Intel P4, 2.6GHz with 2 logical processors, our component system shows a substantially higher performance than the Windows-based systems and also scales higher with regard to the number of parallel processes. The performance advantage is mainly due to the fast context switches of processes in the underlying Bluebottle system; direct context switches are for example performed on message sending and receiving, if the other communication partner is already waiting for a message transfer. Compared to the traditional thread-based systems, the higher scalability results from the lower stack overhead of the active object technology. To estimate the costs of the virtual machine of the component system, the performance is also measured with analogous Active Oberon programs, which directly run on the native Bluebottle system (whereon our virtual machine runs as well). As the difference between both systems shows, the overhead of component language is relatively small, i.e. not higher than about 10 percent.

Table 1. Comparison of execution times (in seconds)

Test application	Component System	Active C#	WinAOS	Native Bluebottle
Producer-consumer	1.6	4.4	10	1.6
Small city simulation	2.9	360	24	2.7
Large city simulation	30	- (*out of memory*)	- (*out of memory*)	28

5 Related Work

The presented language is to our knowledge, the first general-purpose programming language which directly integrates a general component notion with only high-level programming concepts, and which is free of the classical problematic constructs of references, methods and inheritance (see Section 1). Some fundamental concepts of this language are however similar to previous works.

Interface connections. The Microsoft COM [27, 28] wiring mechanism (see [26], Section 10.3) with incoming and outgoing interfaces has similarities to the offered and required interfaces in our language, but is only designed to support asynchronous events using classical method calls. Hence, conventional pointers (or references) still establish the typical component relations in COM. The model of provided and required interfaces is also often used in architecture description languages [3, 20, 21]. However, these languages do not form real programming languages but just allow the

formal description and specification of software architectures. Dynamic structures of components are generally not describable, as the number of components is either static or fixed by a parameter. Moreover, interactions have to be either inadequately represented by method-based interfaces [21], or by low-level message channels (called ports), which are often even unidirectional (like electronic wires) [20]. Other architecture description languages [3] do not have dual provided and required interfaces, but instead necessitate artificial constraints (called glue) to bind a set of ports. With these low-level ports, each client requires a separate interface port for individual communication but a component is typically unable to support an arbitrary (dynamic) number of ports.

Symmetric polymorphism. The symmetric support of offered interfaces is comparable to COM and Zonnon, but in our language, interfaces are merely communication-oriented. Interfaces are also often provided together with a special concept of reusable implementation parts, such as mixins [5] or traits [24]. However, in our language, composition and interface redirection inherently permit flexible implementation reuse without needing such an artificial code mixing mechanism.

Communication-based interactions. The paradigm of message communication has been introduced with CSP [18] and realised in Occam [19]. However, a decisive distinction to our language model is that a component (called process) in CSP/Occam can not interact with multiple interface clients individually, but has to explicitly handle all possible overlapping of client interactions via a time-multiplexed communication channel. The formal Actor model [15, 1], which also proposes communicating parallel components, requires the explicit identification of communication partners by means of references (called mail addresses). This does not only impede clearly described client-individual communications, but also implicates the elementary problems of references like in object-orientation. Our communication model with individual clients is rather influenced from the activity concept of Active C# [13] and Zonnon [14]. Though, in Active C# and Zonnon, clients have to explicitly invoke an activity and interact with the returned dialog, whereas this component language permits direct client-individual communications via interfaces. A further distinction can be made as the component language supports explicit messages with a set of data values and instances that are carried in parameters. Conversely, data values and explicit tokens/tags have to be transmitted as single items in Zonnon, Active C#, CSP, and Occam.

Component systems. A variety of other component models have been invented to enhance structuring, deployment, extendibility and reusability of software [26]. Java Beans, Enterprise Java Beans, CORBA, Microsoft COM, and the Microsoft .NET framework are only some representatives of popular component systems. All these models however have the same fundamental deficiencies with regard to references and methods (see Section 1). With the exception of COM, object-oriented component models also integrate the inheritance relation and its discussed disadvantages.

Other related work. In addition, many efforts have been made to tackle the problems of references with visibility restrictions [8], ownership models [16, 4, 11, 22, 6, 2], region models [7], encapsulation policies [24] and many more. The common problem of all these approaches is that they are still based on the classical low-level model of references and thus require complicated rule systems (mostly integrated in type systems), to ensure structural conditions. Moreover, these models can generally

not describe state-full and client-individual interactions (c.f. iterators in Section 3.2), such that the encapsulation has to be forcibly broken up, by using read-only references [22], dynamic parameter aliasing [16, 4], or simply normal unrestricted references. As conventional references are still supported as standard constructs in these models, the majority of objects may nevertheless be exposed as part of the system-wide flat object graph.

6 Conclusion

The presented component language is a radically new approach for more powerful and structured programming. It integrates a general component notion with appropriate high-level programming concepts, to enable structural clarity, high dynamicity, together with inherent parallelism. As a result, immanent solutions to the various shortcomings of the currently prevalent object-oriented programming paradigm can be gained. The complete implementation and the detailed report of the component language can be found at [9].

Acknowledgments

I am particularly grateful to Prof. Dr. Jürg Gutknecht for his support and helpful advice during this work and for this paper. Many thanks are also due to Dr. Thomas Frey, Dr. Felix Friedrich and other colleagues for their constructive reviews and suggestions for improvement.

References[14]

1. G. Agha. Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, 1986.
2. J. Aldrich and C. Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In European Conference on Object-Oriented Programming (ECOOP), June 2004.
3. R. Allen and D. Garlan. A Formal Basis for Architectural Connection. ACM Transactions on Software Engineering and Methodology, 6(3): 213-249, July 1997.
4. P.S. Almeida. Balloon Types: Controlling Sharing of State in Data Types. In European Conference on Object-Oriented Programming (ECOOP), June 1997.
5. G. Bracha and W. Cook. Mixin-based Inheritance. In Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), October 1990.
6. C. Boyapati, R. Lee, and M. Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), November 2002.
7. C. Boyapati, A. Salcianu, W. Beebee, M. Rinard. Ownership Types for Safe Region-Based Memory Management in Real-Time Java. In Programming Language Design and Implementation (PLDI), June 2003.

[14] With regard to the discussion in Section 3.2, this section lists real references.

8. B. Bokowski and J. Vitek. Confined Types. In Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), November 1999.
9. L. Bläser. The Component Language. ETH Zurich, Switzerland, 2006. Available from http://www.jg.inf.ethz.ch/components.
10. J. Gutknecht, P. J. Muller, T. M. Frey, et al. The Bluebottle Operating System. ETH Zurich, Switzerland. Available from http://www.bluebottle.ethz.ch.
11. D.G. Clarke, J.M. Potter, and J.Noble. Ownership Types for Flexible Alias Protection. In Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), October 1998.
12. F. Friedrich. The WinAOS Oberon System. ETH Zurich, Switzerland. Available from http://www.bluebottle.ethz.ch/winaos.
13. R. Güntensperger and J. Gutknecht. Active C#. .NET Technologies, May 2004.
14. J. Gutknecht and E. Zueff, Zonnon Language Report, ETH Zurich, Switzerland, October 2004. Available from http://www.zonnon.ethz.ch.
15. C. Hewitt, P. Bishop and R. Steiger. A Universal Modular Actor Formalism for Artificial Intelligence, International Joint Conference on Artificial Intelligence (IJCAI), 1973.
16. J. Hogg. Islands: Aliasing Protection in Object-Oriented Languages. In Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), October 1991.
17. C.A.R. Hoare. Hints on Programming Language Design. Stanford Artificial Intelligence Laboratory Memo AIM-224 or STAN-CS-73-403, Stanford University, Stanford, California, December 1973.
18. C.A.R. Hoare. Communicating Sequential Processes. Communications of the ACM, 21(8):666-677, 1978.
19. Inmos Ltd. Occam 2 Reference Manual. Prentice-Hall, 1988.
20. J. Magee and J. Kramer. Dynamic Structure in Software Architectures. In Fourth Symposium on the Foundations of Software Engineering (FSE), October 1996.
21. N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A Language and Environment for Architecture-Based Software Development and Evolution. In International Conference on Software Engineering (ICSE), May 1999.
22. P. Müller and A. Poetzsch-Heffter. A Type System for Alias and Dependency Control. Technical Report 279, Fernuniversität Hagen, 2001.
23. P. J. Muller. The Active Object System. Design and Multiprocessor Implementation. PhD thesis 14755, Department of Computer Science, ETH Zurich, 2002.
24. N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable Units of Behaviour. In European Conference on Object-Oriented Programming (ECOOP), July 2003.
25. N. Schärli, S. Ducasse, O. Nierstrasz, and R. Wuyts. Composable encapsulation policies. In European Conference on Object-Oriented Programming (ECOOP), June 2004.
26. C. Szyperski. Component Software, Beyond Object-Oriented Programming. Addison-Wesley, 1998.
27. A. Williams. Dealing with the Unknown – or – Type Safety in a Dynamically Extensible Class Library. Draft, Microsoft Application Division, 1988. Available from research.microsoft.com/comapps/docs/ Unknown.doc.
28. A. Williams. On Inheritance: What It Means and How to Use It. Draft, Applications Architecture Group, Microsoft Research, 1990. Available from research.microsoft.com/ comapps/docs/Inherit.doc.
29. N. Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? Communications of the ACM, 20(11): 822, 823, November 1977.
30. N. Wirth and J. Gutknecht. The Oberon System. Software – Practice and Experience, 19(9): 857-893, September 1989.
31. N. Wirth. The Programming Language Oberon. Software - Practice and Experience, 18(7): 671-690, July 1988.