A Component-Oriented Language for Pointer-Free Parallel Programming

Luc Bläser

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

Abstract. Today's programming languages typically have substantial deficiencies with regard to structuring and concurrency. To overcome these shortcomings, we have developed a new programming language that is based on a general notion of components with three expressive relations: (1) hierarchical composition without explicit pointers, (2) connections with a dual concept of offered and required interfaces and, (3) communication-based interactions. The programming language is implemented by a runtime system that surpasses existing systems in the scalability and efficiency of concurrency.

1 Introduction

The need for developing a new programming language is usually something that arises in circumstances of an unsatisfactory state of the art. This is also true in our case, where we find that today's predominant programming languages lack the adequate support for constructing modern software systems. More specifically, two most decisive problems can be identified in the currently prevalent programming languages:

- *Structuring.* The use of *pointers* or *references* principally leads to unstructured programs, as object instances can be arbitrarily interlinked (with only some restrictions at the type level). The resulting object graph has no clearly specified shape and no defined hierarchical structure. This means that an object is not capable of encapsulating a sub-structure of objects, since such a structure has to be modelled as normal pointer-linked objects and may still be referenced from other logically external instances.
- *Concurrency.* The increasing need for concurrency cannot be adequately satisfied by a second-class programming concept such as *threading*, which has only been added with hindsight to the procedural programming model. Not only is the use of concurrency with this approach cumbersome and inefficient but it is also unsafe, as threads can operate on arbitrary objects (via method calls) without any clear specification of the potentially accessed instances. As a result, thread dependencies remain largely unspecified in the program, such that concurrency is inherently error-prone (e.g. susceptible to race conditions).

We aim to overcome these weaknesses in our language by way of a new conceptual basis. Instead of featuring classical objects with pointers and methods, a general notion of components is provided. Components can only be organised with three expressive relations:

- *Hierarchical composition.* A component can contain an arbitrary number of other component instances, which are hierarchically encapsulated.
- *Interface connections.* A network of components can be built by connecting the required interfaces of components to corresponding offered interfaces of other components.
- *Communication-based interactions.* All components run concurrently and only interact by means of bidirectional exchange of messages.

The language has already been described in [2] and we here limit ourselves to a summary of the main concepts. In contrast to existing architecture description languages [4] and other component-oriented programming models (such as Microsoft COM, or [1]), our language enables dynamic structures of components and is entirely free of ordinary references, even if compositions are constructed at runtime.

The programming language is implemented by an efficient runtime system that runs as a stand-alone operating system. The highlights of the new system are:

- *High scalability.* The system supports *millions* of parallel processes (threads) on conventional computer machines.
- *High performance.* The execution speed of concurrent programs is on average faster by a factor of 2.7 than in conventional systems.

The remainder of this paper is organised in three sections. The new programming language is described in Section 2. The runtime system is presented in Section 3, where the results of experimental measurements are also shown. In Section 4, we finally draw conclusions for this work.

2 Programming Language

In this language, programs are solely built by components. A *component* constitutes a closed program unit at runtime which encapsulates state (data values and sub-components) and behaviour (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 code by a *template*, which allows creating multiple instances of components at runtime (see Figure 1).

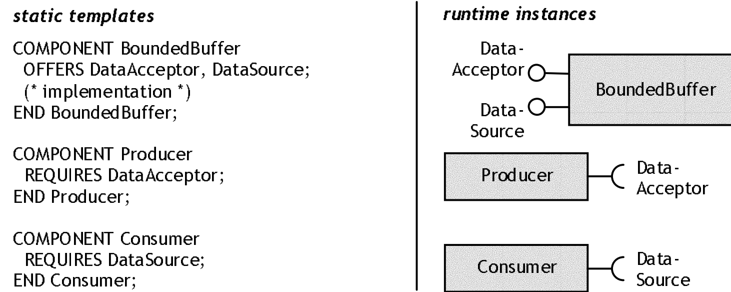

Fig. 1. Component templates and instances

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. `buffer` in Figure 2) or denotes a collection (e.g. `producer[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 (`NEW`). 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). Of course, the components need to be appropriately organised in the memory by using memory indirections. This however happens automatically in the runtime system and is not visible at the level of the programming language.

Components within the same scope may be also connected to networks. For this purpose, each required interface of a component can be connected to an interface that is offered by another component. In order to be connectable, the required and offered interfaces must have the same name (see Figure 2). With these relations, component networks of arbitrary shape can be constructed under the control of the surrounding instance. Besides connecting sub-components, the language also supports connecting the interfaces of the surrounding component with interfaces of sub-components (see [2]).

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

```
COMPONENT Simulation;
VARIABLE
 buffer: BoundedBuffer;
 producer[i: INTEGER]: Producer;
 consumer[i: INTEGER]: Consumer;
 i, n, m: INTEGER;
BEGIN (* read n and m from user *)
 NEW(buffer);
 FOR i := 1 TO n DO
  NEW(producer[i]);
  CONNECT(DataAcceptor(producer[i]), buffer)
 END;
 FOR i := 1 TO m DO
  NEW(consumer[i]);
  CONNECT(DataSource(consumer[i], buffer)
 END
END Simulation;
```

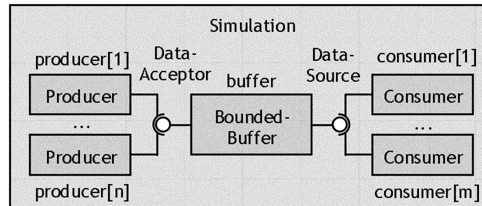

Fig. 2. Hierarchical composition

2.2 Concurrency and Communication

Every component runs its own inner processes and only interacts by message communication. Two components, which are connected, can directly exchange messages via their connected interfaces. The component which offers the interface may be regarded as the server, while the other component acts as the client with regard to this interface. A message transmission involves one side sending the message and the other one explicitly receiving it. A particular feature is that the server component maintains a separate communication for each client individually. This means that each client-server pair a client and server has a separate communication channel via the interface, such that the server may communicate with multiple clients in parallel (see Figure 3).

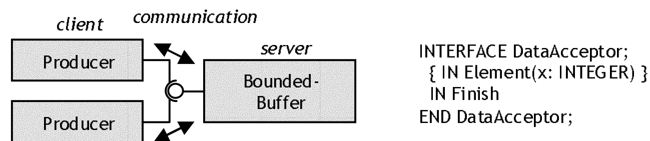

Fig. 3. Message communication

However, two components cannot send and receive messages in an arbitrary way but must follow a communication protocol. This protocol is defined in EBNF [6] (right-hand side of Figure reffigure:Communication) within the interface and specifies all message transmissions that are allowed between a client and a server of that interface. A message transmission is therein prefixed by `IN`, if it is sent from the client to the server, and `OUT`, if sent in the opposite direction. A sequence of messages is denoted by subsequent expressions, while curly brackets represent arbitrary repetition (including zero times) of a sub-protocol.

As for the implementation of the components, the server component automatically starts a separate service process for each client, running inside the server component and performing the server-side communication with the corresponding client. For a more detailed explanation, the reader is referred to [2]. Since the interactions between different components are inherently synchronised by the communication, only the processes *within* the same component instance need to be explicitly synchronised by monitor protection. The compiler excludes uncontrolled concurrent accesses (race conditions) inside a component.

3 Runtime System

In order to enable efficient execution of the component-oriented programs, the language is implemented by a runtime system that directly runs with its own kernel on normal PCs (IA32) and has the following features:

- *Fine-granular stacks.* Processes are extremely light-weighted, with stacks that can have arbitrarily small size. In general, stack sizes do not need to grow and shrink at runtime, because the programming model uses communication instead of method calls.
- *Software-controlled preemption.* Preemptive execution of processes is realised by instrumented code that is automatically inserted by the compiler at required points. No unnecessary register backups have to be taken for preemption.
- *No garbage collection.* The system ensures memory safety without the need for automatic garbage collection. This is because the time for the memory deallocations is exactly defined by the hierarchical compositions.

We have measured the maximum support number of processes (threads) using a computer machine with 4GB main memory, see Figure 4. Our system (Component OS) indeed permits millions of processes, whereas the other systems limit the number of processes to a small amount.

We also assembled a set of concurrent programs, to determine the runtime performance. The programs are written in our component language and analogously in classical object-oriented languages, using threads in place of the intrinsic component processes. To stick to the right paradigm, we implemented the object-oriented programs by using methods for object interactions. As can be seen in Figure 4, the new system outperforms the C#, Java and AOS [5] with a median speedup of factor 2.7, if we compare the result of each test case with the corresponding best other system. Only for some simple programs is our implementation slower (the dynamic collections are more expensive than explicit arrays). All test programs are available at [3].

4 Conclusion

The presented component language provides a new programming model which enables dynamic structuring with expressive and hierarchical relations instead

Maximum number of processes (threads)

Component OS	Windows C#	Windows Java	AOS
5,010,000	1,890	9,999	15,700

Runtime in seconds

test program (in seconds)	Component OS	C#	Java	AOS	speedup[2]
City	**0.26**	0.66	440	4.1	2.5
ProducerConsumer	**18**	19	130	60	1.1
Eratosthenes	**1.6**	6.8	4.6	5.8	2.9
News	**0.82**	3.5	3.9	3.7	4.6
Library	0.78	0.74	1.5	**0.59**	0.76
TokenRing	**2.1**	22	22	18	8.6
Mandelbrot	0.89	0.43	**0.39**	0.6	0.44
TrafficSimulation	**0.05**	33	-[1]	-[1]	660

Intel Xeon, 6 CPU with 700MHz, 4GB memory, C# and Java ran on Windows Server 2003 Enterprise Edition; [1] not implemented; [2] speedup of Component OS compared to the fastest other system

Fig. 4. Experimental measurements

of pointers. At the same time, the language institutionalises first-class support of well-controlled concurrency by self-active components that solely interact by message communication. The language is also efficiently supported by our customised runtime system, clearly surpassing existing systems in the number of processes and the performance of concurrency.

Acknowledgments

I gratefully appreciate the helpful support and constructive advice from my supervisor Prof. Dr. Jürg Gutknecht. I also express my thanks to Dr. Felix Friedrich, who proofread this paper and also gave valuable feedback.

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), Sept. 2006, LNCS Vol. 4228, Springer Verlag, 2006.
3. *The Component Language and System*, http://www.jg.inf.ethz.ch/components.
4. N. Medvidovic and R. N. Taylor. *A Classification and Comparison Framework for Software Architecture Description Languages*, Software Engineering, 26(1):70-93, 2000.
5. P. J. Muller. *The Active Object System Design and Multiprocessor Implementation.* PhD Thesis, Diss. ETH No. 14755, ETH Zurich, 2002.
6. N. Wirth. *What Can We Do About the Unnecessary Diversity of Notation for Syntactic Definitions?*, Communications of the ACM, 20(11):822-823, Nov. 1977.