Requirements for a Composition Language [*]

Oscar Nierstrasz
Theo Dirk Meijler

University of Berne[†]

Abstract

Abstract The key requirement for open systems is that they be flexible, or *recomposable*. This suggests that they must first of all be composable. Object-oriented techniques help by allowing applications to be viewed as compositions of collaborating objects, but are limited in supporting other kinds of abstractions that may have finer or coarser granularity than objects. A composition language supports the technical requirements of a component-oriented development approach by shifting emphasis from programming and inheritance of classes to specification and composition of components. Objects are viewed as processes, and components are abstractions over the object space. An application is viewed as an explicit composition of software components. By making software architectures explicit and manipulable, we expect to better support application evolution and flexibility. In this position paper we will elaborate our requirements and outline a strategy for the design and implementation of a composition language for the development of open systems.

* In Proceedings of the ECOOP 94 workshop on Models and Languages for Coordination of Parallelism and Distribution, Springer Verlag, LNCS 924, 1995, pp. 147-161.

† *Authors' address:* Institut für Informatik (IAM), Universität Bern, Neubrückstrasse 10, CH-3012 Berne, Switzerland. *Tel:* +41 (31) 631.4618. *Fax:* +41 (31) 631.3965. *E-mail:* {oscar, meijler}@iam.unibe.ch *WWW:* http://iamwww.unibe.ch/~scg

1 Introduction

Software systems can be viewed in two distinct ways. A running system can be seen as a collection of interacting run-time entities. At the level of system specification, however, we can view the same system as a *composition* of various software components [32]. The granularity and nature of the components need not necessarily correspond to that of the run-time entities, since the domain of discourse is different. In the first case, we are in the domain of application itself, whereas in the second case we are in the domain of systems construction. The nature of the abstractions that are useful for each domain will, in general, be different.

When we speak of open systems, this distinction is crucial, because the essential feature of open systems is that they be *recomposable* — a system is open if it is open to changing requirements. That means that the way in which the system is composed must be open to change over time. We claim that object-oriented techniques go a long way to supporting this view of open systems, but they fall somewhat short of the mark by (i) failing to clearly distinguish the computational and compositional views of applications, and (ii) over-emphasizing the object view, thus failing to provide general *component* specification and composition mechanisms. A composition language would function at a higher level of abstraction by providing an integrating framework for objects *and* components. Components function at the compositional level whereas objects function at the computational level. The interfaces are different because different concerns are addressed. Object interfaces address computational needs, whereas component interfaces must be designed to support flexible system construction.

To support open systems development, a composition language must support the specification of (i) systems as compositions of components, (ii) *component frameworks* that define standard interfaces, protocols and behaviour, and (iii) interface mappings to existing components possibly written in other programming languages. Component frameworks are analogous to object-oriented frameworks (i.e., abstract class hierarchies for particular domains [13]), but allow for the specification of general component abstractions, not just object classes. Component frameworks essentially define *architectural styles* [1] in that applications built using the same framework will exhibit similar architectural structure and make use of the same kinds of collaborations between components. Furthermore, individual application architectures will be made explicit and manipulable by specifying applications as compositions of components. The essential difference with approaches taken in projects such as *Darwin* [18] is that, instead of fixing the architectural styles of application through a particular set of composition mechanisms, the component frameworks supported by a composition language will be open-ended.The language can be used not only to specify compositions themselves, but also the kinds of composition mechanisms appropriate to a given architectural style.

The composition language we propose is an experimental prototype currently under development. To address open, distributed systems, it is based on an object model in which objects and components map to a formal abstract machine integrating processes and functions [28]. The mapping is also the basis for the experimental prototype, as the abstract machine is itself executable. Our development strategy is highly experimental, and we expect the language design to evolve rapidly as the result of our application experiments. One of our goals is to use the composition language as the back-end for a framework-driven visual composition tool. The formal model of objects and components must then translate to a graphical representation that will be more convenient for interactive application composition based on component frameworks.

In this position paper we present a motivating scenario in §2 and we summarize our requirements in §3. In §4 we discuss the formal object model and the open problems to be resolved. Our implementation strategy and ongoing work is outlined in §5, and we conclude with some remarks about future directions.

2 Compositional Development of Open Systems

Large companies and institutions often have a great need for open component frameworks since they must frequently develop many similar applications, and they must cope with long-lived applications whose requirements change over time. Let us consider an example which has been worked on by one of the authors.

The ministry of Education and Science in the Netherlands used to send around many different kinds of paper forms to many different kinds of educational institutions. These institutions would fill in and return these forms; on the basis of the information gathered, the size of subsidies and allowances would be determined. Now the ministry wants to replace the various paper forms by electronic form systems in order to gather and process the information more efficiently.

Since:

- the design of a given form may change considerably from year to year,
- the various different form applications should look and feel alike to users, and
- the ministry would like to reduce the costs of developing many similar forms applications,

a need was perceived for a generic “framework” for creating electronic form applications. The ministry, however, was also concerned about limiting the cost of developing the framework.

The framework that was developed was *not* realized as an abstract class hierarchy. For the kind of applications under consideration, an abstract class hierarchy was perceived as providing too much freedom, and being too difficult to use to create a single application. Furthermore, setting up such a framework is not deemed to be an easy task [13]. Instead, the most natural scenario for developers turned out to be to specify electronic forms applications by directly listing the questions that should be presented in the application and the kind of information that should be entered per question*. The framework was designed so that the specification *was* the application. A form of subclassing was still needed when, for example, integrity and correctness tests would have to be added to a form, or when particular forms or questions required non-standard presentation. At the moment, these forms are used within a partly human-driven work-flow for financing the various institutions. These applications may therefore well become part of a larger fully-automated work-flow system.

* Note that our description of the example has been simplified to the extreme for the purpose of this paper.

The example raises the following questions:

- How can we create families of components and applications in such a way that it is both easy to set up a framework for such a family, and easy to “instantiate” a specific application (or component)?
- How can we set up various frameworks such that we can still create a greater whole in which the different applications created using these various frameworks may be viewed as components that cooperate? Kinds of cooperation may for example be, that components are linked from input to output, or that one component may delegate a task to another component.
- How can we still exploit object-oriented reuse features, particularly inheritance, while adopting a more component-oriented development approach?

In general, the combination of these three aspects, and especially of the first two, has not been supported very well by software development environments and methods (whether object-oriented or not). Because frameworks based on class-hierarchies are still hard to create and to use these have been created for large scale use and/or by large institutions only[6][16][17]. Much work is being done on component interoperability [4][37][38][40][42] but these solutions are not frameworks and they are not framework-based, that is, they don’t support development of distributed applications as a whole, they provide only specific forms of connectivity.

We assert that the following approach should be taken towards the development and use of open and flexible frameworks:

A single framework should be based on an identification of the terms or components and composition mechanisms in which a user of the framework (the developer) wants to describe an application. In our example these components are forms, questions and their contents. The composition mechanism used here is object inclusion: a form consists of questions.

Creating such a framework will amount to defining these domain specific “high-level” components and composition mechanisms and mapping these to more general “low-level” components and composition mechanisms. In the example, this amounted for example to mapping the high-level question component, to somewhat lower level components of a general spreadsheet widget (interaction object) and text-field widgets.

It must be possible to use a single application instantiated from a framework as a component of a larger system. Creating frameworks for open systems requires extra effort. It must be possible to regard the *run-time abstractions* that are created as general computational entities so that they can cooperate. The run-time abstractions may either be objects (e.g., an editable form may be viewed as an object, possibly as an active object so that it may actively participate in the work-flow) or information filters. However, since the user of a framework thinks in terms of design-time entities or components, a framework must support the linking of these components. From the point of view of a component as part of a larger whole, the form application of the example must be viewed as a data-source component (since it is the source of a filled-in form), which can be linked to other components to process the information. In general links between components that may be supported are, for example, data-flow links [21] and links to describe how components in which requests for operations are generated (user interface components) can be served by components that execute these operations [20].

The creation of frameworks may be supported by a framework of frameworks or a "meta-framework." For that meta-framework, the components are the abstractions that are used by the framework-designer to *define* the components and composition mechanisms of the framework. This meta-framework may also support the mapping between abstractions by providing specific design-time links between definitions of high-level and definitions of low-level components (e.g. between the definition of "table-question" and "spread-sheet widget" for our example).

We can distinguish two kinds of technological support that can help support framework design and use:

1. Support for component framework specification. Since we seek high-level tools for instantiating applications from frameworks, the specification of a framework should assign a semantics to components and compositions by a mapping to the run-time entities [20]. It must be possible to specify a range of different kinds of composition and plug-compatibility rules between components.
2. Support for application composition. It must be possible to specify an application as a composition of components conforming to particular framework. Interactive tools for visualising frameworks, components and compositions would help as a bridge between analysis (requirements), design (frameworks) and implementation (compositions). Ideally, visual composition would be *framework-driven.* Support can, and must be than provided for reasoning about compositions, and for mapping compositions to the run-time system.

3 Requirements for a Composition Language

A composition language should support the flexible construction and evolution of applications by promoting systematic component-oriented development of open systems. This requires not only that one be able to specify systems as compositions of software components, but that these compositions conform to component *frameworks* that encode flexible architectures for a range of applications. It is not enough to have a library of "reusable" software artifacts — what makes a software component interesting is the fact that it has been *designed* to work together with other components within the context of a generic architecture. This is precisely what distinguishes an object-oriented framework from a traditional software library. A component framework generalizes this notion to frameworks of components that are not necessarily object classes, and are not necessarily related by inheritance: components may be of finer or coarser granularity, and other forms of abstraction and composition may relate components.

If we consider the key characteristic of open applications, we can derive a series of requirements for a composition language:

- *Open topology:* open applications are inherently concurrent and distributed;
- *Heterogeneity*: applications may run on a variety of hardware and software platforms;
- *Evolving requirements:* application requirements are not fixed in advance, so a flexible architecture is needed to meet changing requirements.

These characteristics motivate the following technical requirements:

1. *Encapsulation:* to cope with system complexity and to provide for flexible architectures, both *objects* and *components* are needed;
2. *Objects as Processes:* objects may be active or passive, local or remote, simple or composite, but all objects can be viewed as processes;
3. *Components as Abstractions:* components are (potentially higher-order) software abstractions that are composed in various ways to yield applications;
4. *Plug compatibility:* a type model encompassing both objects and components is needed to reason about valid bindings and compositions;
5. *Formal object model:* a standard object model is needed to bridge the gap between the composition and language and implementation of components on one hand, and higher-level composition tools on the other hand;
6. *Scalability:* the use of the language should scale from small to large systems, and from highly dynamic usage with incomplete type information to compiled and highly optimized usage.

Let us consider each of these requirements in turn. We have already briefly discussed the need for both objects and components. Objects are run-time entities that are configured to achieve a certain end. The configuration — or *composition* — itself need not correspond precisely to the object-level view, since both more finely-grained and more coarsely grained components may come into play. Mix-ins, interfaces and protocols are all prime candidates for specification as components, but cannot be instantiated directly as objects, and are not necessarily conveniently modeled as abstract classes either. Modules, packages, frameworks and even generic configurations or architectures are also good candidates for more coarsely-grained components [2].

Objects, being run-time entities, in general have state and may execute concurrently. Whether or not objects have their own internal threads, or may be otherwise considered "active," each object can be viewed as a kind of server, or *process*. The process view of objects formalizes the notion that an object is an autonomous entity, and not just a data structure to be accessed by a designated set of procedures. In the context of concurrent clients, it is even more important to formalize the process view, as it will allow us to express the synchronisation constraints directly within the behaviour of an object rather than as an orthogonal concept. This corresponds to the "homogeneous" object model for object-based concurrency [34] in which all objects are potentially active, and assume responsibility for their own concurrency control.

Components, on the other hand, are software abstractions. They may also be run-time entities to which applications may connect, but, more generally, components must be composed and instantiated before they are part of a running application. Mix-ins and templates are prime examples of components that are *not* run-time entities. Components will typically be higher-order, in the sense that the composition of existing components may yield new components that must be further instantiated before run-time entities are obtained. This is precisely what happens with inheritance: an abstract class is a higher-order component that is partially instantiated to another component — a subclass — by inheritance [3]. By supporting the notion that software components may be *any* useful abstraction, not just object classes, we allow for the definition of much more flexible, and, we argue, more conceptually natural component frameworks.

To support flexible reconfiguration of applications, it is important that not only compositions be explicit, but that the generic architectures from which they are derived be explicit as components. In an object-oriented framework, the architecture of applications based on the framework is typically *implicit* in the interfaces of the class hierarchy. In a component framework, architectures (generic compositions) would themselves be components.

Components, however, can still benefit from an object flavour: One of the strengths of inheritance over functional composition is that the sequence in which formal parameters (i.e., virtual methods) are bound is not fixed in the class interface. A similar degree of flexibility can be obtained by using name-based binding for parameters for components as well [5]. Default values for parameters can also be specified and optionally overridden in the same way as is possible in most inheritance schemes.

Plug compatibility formally expresses how components may be composed. Various different forms of "composition" may co-exist, however, and plug-compatibility must cope with these different forms. At the very least, functional composition and communications interconnection, for components and objects respectively, must both be considered. Although most attempts to formally specify type systems for object-oriented languages adopt a functional view of objects [10], such a view ignores the fact that objects typically require clients to conform to a simple protocol in order for their composed behaviour to be valid [30]. Components, on the other hand, being static rather than dynamic, are truly functional entities. A unified type model that accommodates both objects and components would allow one to reason about plug compatibility for both kinds of entities, and would at least partially address the problem of checking compatibility of object protocols.

A composition language should also serve as a bridge between traditional implementation languages and higher-level composition tools. A formal object model is necessary to act as the "glue" between these layers. Integrating objects and components, concurrent activities, communication, and a rigorous notion of plug compatibility, into a common framework presents various semantic difficulties [31]. We argue, therefore, that not only is a common object model necessary, but the model must have a formal foundation that addresses the semantics of both functional composition and concurrency. This foundation can then not only be used to formalize plug compatibility, but it can be used as a reference for establishing the correspondence between the composition language and implementation languages, and it can be used to provide either directly or indirectly (i.e., via the composition language) a formal semantics for graphical composition tools [21].

A final key requirement for a composition language is that it be scalable. Both small and large systems, and centralized and distributed systems should be configurable. A scalable language can be used both in a "rapid prototyping" mode and in production mode. In the first case, changes can be dynamically made to a running system, and in the second case, static analysis can be performed to generate a more optimized runtime system. Explicit type declarations may be left out when composing a system, but will be typically required when a component framework is released and published. When a composition language is used as back-end for a visual composition tool, it must be possible to immediately propagate changes in configuration to running applications, rather than requiring the tool to generate and compile code from the graphical specifications. On the other hand, it must be possible to eliminate unnecessary dynamic lookups and checks in stable configurations by analyzing the specification and generating optimized code. It should not be necessary to sacrifice flexibility to achieve acceptable performance, or vice versa.

4 Formal Models of Objects and Components

Difficulties in integrating concurrency into object-oriented languages are well-documented [14][19][31][34]. Integrating a component-oriented approach with concurrent objects poses yet more semantic difficulties, particularly in terms of developing a usable model of plug-compatibility for both objects and components. A formal model of objects and components should be chosen in order to help answer the following questions:

- *How can objects be viewed as processes?* Objects are message-passing entities with hidden state, and so are processes, but processes may exhibit arbitrary behaviour, whereas objects are interesting because (i) they provide *services* according to a public interface, and (ii) they have a very regular internal structure that allows them to be both instantiated and specialized. A formal object model must dictate what *kinds* of processes are objects.
- *How can components be viewed as process abstractions?* A composition specifies how an application, as a collection of interacting objects, is composed from a set of software components. If objects are processes, then components are abstractions (functions) over the process space.

What does this mean formally? May components be arbitrary abstractions, or, as with objects, must some discipline be imposed in order to obtain only meaningful abstractions?

- *How can inheritance co-exist with other forms of composition?* Inheritance can be viewed as a combination of higher-order compositions [3]. Integrating inheritance cleanly with concurrency features is non-trivial [14][19], so extending a concurrent object-oriented model to accommodate component-oriented composition cannot be straightforward. To achieve such an integration, we believe that it will be necessary to "unbundle" inheritance and understand it in terms of more primitive composition operations. Once we can talk about both inheritance and composition within the same formal model, the integration issues should become transparent.

- *What is plug-compatibility for objects and components?* Objects and components will have different kinds of "plugs" or type interfaces, but plug compatibility is an issue in both cases, as we would like to be able to replace objects and components within a composition by plug-compatible ones. Types and subtyping rules must be formally specified, and their semantics justified in terms of the formal object model. An open problem is how to extend traditional type systems to express limited dynamic properties such as protocols: a valid client-server relationship typical requires the client to obey a simple protocol. Rather than simply raise exceptions when protocols are violated, it would be desirable to (i) express protocols formally as part of an object's type interface, (ii) statically validate clients' conformance to protocols, and (iii) determine automatically when one protocol can be formally viewed as a "subtype" of another [30].

- *How can we infer properties of the behaviour of composed applications from the specifications of their components?* Reasoning about behaviour in a compositional way is a notoriously difficult problem. So far it is not possible to prove systems correct from the knowledge that individual components may be correct, for various technical reasons (such as aliasing [11]). To have any hope of proving even partial properties of systems from the specifications of components, we need to assign formal semantics to compositions.

- *What does it mean to compose components from different hardware and software environments?* For a composition language to act as "glue" between components written in different languages, or running in different environments, a common reference model is needed to define what composition and interaction mean across the boundaries of different language models.

Technically, there are a number of features that a formal model should capture:

1. *Communication and binding:* objects' behaviour consists in the exchange of messages; components' functionality is instantiated by binding formal parameters to values; objects and components may themselves be values;
2. *Concurrency:* an application is a concurrent composition of objects (whether or not there may be multiple concurrent threads active at any time);
3. *Choice:* an object typically provides an interface consisting of a choice of services; a component may be composed in a variety of alternative ways;
4. *Abstraction:* objects and components are abstract entities whose behaviour and functionality is only accessible through their interface;
5. *Instantiation:* objects can be dynamically instantiated, so it must be possible to generate new names for objects and their communication channels, and to communicate these names to existing objects.

Considering these requirements, it appears that we need a formal model that combines features of a process calculus (to model objects as processes) and a λ calculus (to model components as abstractions). In fact, there has been some success in modeling objects and inheritance using CCS as a foundation [33]. CCS [22] is a process calculus modeled loosely after the λ calculus, in which functional abstraction is replaced by input guards over process expressions, where guards are associated with named channels, application is replaced by output guards also addressed to named channels, and composition is by choice (over guarded expressions) and concurrent composition. (Non-deterministic) reduction occurs when input and output guards match, and communication takes place.

The shortcomings of CCS are well-known, the most significant being that, although new processes can be dynamically instantiated, channel names cannot be communicated and no new channel names can be dynamically introduced. Functional abstraction cannot be modeled, and it is also not possible to pass process expressions as values. These technical shortcomings where attacked by various researchers [7] [23] [25] [36] [39], and have culminated in the π calculus, a calculus of "mobile processes" in which channel names can be communicated and newly introduced using rules analogous to those for the λ calculus to avoid capture of names. Although the π calculus *only* allows for the communication of names as values, it has been shown that both a polyadic π calculus [26] (allowing the communication of tuples), and a higher-order π calculus [36] (allowing the communication of process abstractions as values) can be faithfully modeled by a mapping to the monadic calculus.

The higher-order π calculus is a close fit to our requirements and appears to be an excellent basis for developing a formal model of objects and components. Already, there have been some attempts to develop an "object calculus" based on variants of the π calculus [12] [28], and some researchers have experimented with modeling object-oriented language features in the π calculus [35]. So far, however, there is no standard model of objects as processes, and the relative advantages and disadvantages of the possible mappings have not be systematically catalogues or evaluated.

Another consideration for a calculus of objects and components would be the use of names for component abstractions. As demonstrated by Dami [5], a λ calculus with names not only goes a long way towards modeling object-oriented features (such as inheritance) more conveniently than a conventional λ calculus (though without addressing concurrency and communication), it greatly increases flexibility in the specification and use of abstractions since parameters can be bound by name in any order. This suggests that a higher-order π calculus based entirely on names would be a good foundation for a formal model of objects and components.

5 Incrementally Developing a Composition Language

We propose an experimental and evolutionary approach to developing and implementing a practical composition language. There are too many open questions to consider a conventional approach in which the language is fully specified before a compiler is implemented and the language is first used:

- What is an appropriate semantic foundation?
- How can objects and components best be modeled within that foundation?
- What language features are most useful for defining compositions and component frameworks?

Instead, we propose an approach in which the language can be used already *while* it is being formally specified, and so insight concerning the usefulness of concepts and features can feed back more quickly into the language design. In fact, the language itself is not the goal, but the discovery and identification of the concepts and mechanisms that will best support component-oriented development.

A layered approach to language design would proceed as follows: an abstract machine would be built that implements a higher-order process calculus, as described above. Language features to implement objects and components are then designed by specifying a semantic mapping from the syntactic constructs of the language to the process calculus [22][33]. The mapping will be an executable specification, as will the target calculus. As a consequence, a running prototype will be available at all times.

Since the goal is to support component-oriented development, is critical that the language be tested on “real” rather than “toy” examples. If the language is designed “on paper” before it is used, it will be impossible to experiment with more than toy examples until an implementation is complete and available. If the language specification is itself a running prototype, it will be possible to experiment with interesting examples from the beginning and to have these experiments directly influence the evolution of the language itself. Experiments will include existing (object-oriented) frameworks as well as composition tools. Since compositions may act as glue between existing components, it will be important to experiment with existing components as early as possible.

The experiments should serve to answer the questions listed above, namely, what abstractions are most useful for defining compositions and component frameworks, what basic model of objects and components best supports these abstractions, and what formal semantics should serve as a foundation. Clearly, the most crucial part to fix is the formal foundation. The first experiments should determine what features of a higher-order process calculus are needed for an adequate abstract machine, and whether there are any technical shortcomings to be resolved. Next, the different possible ways of modeling objects and components must be systematically evaluated. This suggests that a *class* of possible language designs must in effect be evaluated.

Finally, specific language features must be introduced and evaluated. This should be the last aspect of the language to be frozen. In fact, since the language should serve as a “meta-framework” for defining various component frameworks, it should be as flexible as possible. Ideally, one should be able to define new language concepts with their associated syntax as new abstractions, analogous to the way that new language features can be introduced into CLOS by defining a suitable meta-object protocol [15].

Once the semantic foundation and the object model are stabilized, it may be possible (and desirable) to eliminate an implementation layer by directly implementing the object model. The generality of the process calculus will not be needed to implement language features, since they will all map to the level of the object model, and it will certainly be possible to exploit properties of the object model to achieve a more efficient prototype. The underlying process calculus semantics will serve, however, as a reference implementation, and can provide formal justification for any optimizations introduced.

Although a textual language has the advantage of being easier to formally specify and implement, a graphical interface to software composition has more appeal as a natural way to view and think of components and compositions. We plan to develop in parallel a framework-driven visual composition tool that can act as a front end to a composition language. As with the work of de Mey [21], such a tool would differ from existing commercial visual composition tools in that it would be application domain-independent. The tool would be parameterized by “composition models” for various component frameworks that determine what constraints apply to their composition. In our approach, additionally, the semantics of composition would be directly inherited from that of the composition language, rather than from a separately provided model. Moreover, the way that components and compositions are visualized will also be configurable for different domains, as part of the composition model. In the long term, we expect that this configuration itself will just be another application of visual composition.

6 Concluding Remarks

A composition language would function at a higher level than a programming language by allowing one to specify explicitly components, compositions and component frameworks. Such a language would lie somewhere between Smalltalk [8], ML [24] and Perl [41], providing a computational model in which one may talk about (concurrent) objects, higher-order abstractions, and interaction with external components. As with Smalltalk, one should be able to define frameworks of abstract components; as with ML, one should be able to specify strongtyped, higher-order abstractions; and as with Perl, one should be able to compose applications quickly and flexibly from both newly-defined and existing components.

We have argued in favour of a formal approach to specifying and developing a composition language, but we have only presented our requirements. The precise natures of the semantic foundation, of the formal object model, and of the language itself have not been defined here. Instead, we have proposed an evolutionary approach based on executable language specification, that will allow us to arrive at a stable language design by experimentation and incremental refinement. Some earlier experiments were encouraging [27][28], and we plan now to embark on a more ambitious experiment.

A number of difficult technical issues remain to be explored and resolved. Aside from the problem of elaborating the formal object model, there is an open question of developing a suitable type model for both objects and components. “Plug-compatibility” should take not only interfaces into account, but the fact that objects, to be used correctly, often require clients to respect a particular protocol. A framework for specifying protocols as finite state processes has been proposed by one of the authors in a separate paper [30], but it remains unclear how such a framework could be practically incorporated into a composition language.

Efficient implementation of a composition language is a long-term goal, but it is too early to say how flexibility and acceptable performance can both be provided while respecting the desired formal semantics. We expect that it will be possible to partially optimize the implementation of statically analysed components by techniques that can be justified by the formal semantics.

Finally, we acknowledge that the most difficult problems are not so much technological as methodological [29]. Component-oriented development is based on a different software process and software lifecycle from traditional (or event object-oriented) development. In particular, the incremental development of component frameworks must be explicitly supported by the methods and by the project management infrastructure [9]. A composition language only attempts to provide limited technological support for such methods.

Acknowledgements

We gratefully acknowledge the financial support of the Swiss National Science Foundation under project no. 21-40610.94.

References

[1] Gregory Abowd, Robert Allen and David Garlan, “Using Style to Understand Descriptions of Software Architecture,” *Proceedings SIGSOFT 93, ACM Software Engineering Notes*, vol. 18, no. 5, Dec 1993, pp. 9-20.

[2] Gilad Bracha, “The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance,” Ph.D. thesis, Dept. of Computer Science, University of Utah, March 1992.

[3] William Cook and Jens Palsberg, “A Denotational Semantics of Inheritance and its Correctness,” *Proceedings OOPSLA ’89, ACM SIGPLAN Notices*, vol. 24, no. 10, Oct. 1989, pp. 433-443.

[4] William Cook, “Application Integration, not Application Distribution,” *ACM OOPS Messenger, Addendum to the Proceedings of OOPSLA 1993*, vol. 5, no. 2, April 1994, pp. 70-71.

[5] Laurent Dami, “Software Composition: Towards an Integration of Functional and Object-Oriented Approaches,” Ph.D. thesis No. 396, University of Geneva, 1994.

[6] Thomas Eggenschwiler and Erich Gamma, “ET++SwapsManager: Using Object Technology in the Financial Engineering Domain”, *Proceedings OOPSLA ’92, ACM SIGPLAN Notices*, vol. 27, no. 10, Oct. 1992, pp. 166-177.

[7] Uffe Engberg and M. Nielsen, “A Calculus of Communicating Systems with Label Passing,” DAIMI PB-208, University of Aarhus, 1986.

[8] Adele Goldberg and David Robson, *Smalltalk 80: the Language and its Implementation*, Addison-Wesley, May 1983.

[9] Adele Goldberg and Kenneth S. Rubin, *Succeeding With Objects: Decision Frameworks for Project Management*, Addison Wesley, 1995, forthcoming.

[10] Carl A. Gunter and John C. Mitchell, *Theoretical Aspects of Object-Oriented Programming*, The MIT Press, 1994.

[11] John Hogg, “Islands: Aliasing Protection in Object-Oriented Languages,” *Proceedings OOPSLA ’91, ACM SIGPLAN Notices*, vol. 26, no. 11, Nov 1991, pp. 271-285.

[12] Kohei Honda and Mario Tokoro, “An Object Calculus for Asynchronous Communication,” *Proceedings ECOOP ’91*, P. America (Ed.), LNCS 512, Springer-Verlag, Geneva, Switzerland, July 15-19, 1991, pp. 133-147.

[13] Ralph E. Johnson and Brian Foote, “Designing Reusable Classes,” *Journal of Object-Oriented Programming*, vol. 1, no. 2, 1988, pp. 22-35.

[14] Dennis G. Kafura and Keung Hae Lee, “Inheritance in Actor Based Concurrent Object-Oriented Languages,” *Proceedings ECOOP ’89*, S. Cook (Ed.), Cambridge University Press, Nottingham, July 10-14, 1989, pp. 131-145.

[15] Gregor Kiczales, Jim des Rivières and Daniel G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press (Ed.), 1991.

[16] Mark a. Linton, John M. Vlissides and Paul r. Calder, “Composing user interfaces with InterViews”, *Computer*, Vol. 22, no. 2, 1989, pp. 8-22.

[17] Peter W. Madany, “An Object-Oriented Framework for Filesystems, *Ph.D. Thesis University of Illinois at Urbana-Champaign*, 1992

[18] Jeff Magee, Naranker Dulay and Jeffrey Kramer, ‘‘Structuring Parallel and Distributed Programs,” *Proceedings of the International Workshop on Configurable Distributed Systems*, London, March 1992.

[19] Satoshi Matsuoka and Akinori Yonezawa, “Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages,” *Research Directions in Concurrent Object-Oriented Programming*, G. Agha, P. Wegner and A. Yonezawa (Ed.), MIT Press, 1993, pp. 107-150.

[20] Theo Dirk Meijler, “User-level Integration of Data and Operation Resources by means of a Self-descriptive Data Model”, Ph.D. Thesis, Erasmus University Rotterdam, Sept. 1993

[21] Vicki de Mey, “Visual Composition of Software Applications,” Ph.D. thesis (no. 2660), Dept. of Computer Science, University of Geneva, 1994.

[22] Robin Milner, *Communication and Concurrency*, Prentice-Hall, 1989.

[23] Robin Milner, Joachim Parrow and David Walker, “A Calculus of Mobile Processes, Parts I and II,” Reports ECS-LFCS-89-85 and -86, Computer Science Dept., University of Edinburgh, March 1989.

[24] Robin Milner, M. Tofte and R. Harper, *The definition of standard ML.*, MIT Press, Cambridge, 1990.

[25] Robin Milner, “Functions as Processes,” *Proceedings ICALP ’90*, M.S. Paterson (Ed.), LNCS 443, Springer-Verlag, Warwick U., July 1990, pp. 167-180.

[26] Robin Milner, “The Polyadic pi Calculus: a tutorial,” ECS-LFCS-91-180, Computer Science Dept., University of Edinburgh, Oct. 1991.

[27] Oscar Nierstrasz, “A Guide to Specifying Concurrent Behaviour with Abacus,” Object Management, D. Tsichritzis (Ed.), Centre Universitaire d’Informatique, University of Geneva, July 1990, pp. 267-293.

[28] Oscar Nierstrasz, “Towards an Object Calculus,” *Proceedings of the ECOOP ’91 Workshop on Object-Based Concurrent Computing*, M. Tokoro, O. Nierstrasz, P. Wegner (Ed.), LNCS 612, Springer-Verlag, 1992, pp. 1-20.

[29] Oscar Nierstrasz, Simon Gibbs and Dennis Tsichritzis, “Component-Oriented Software Development,” *Communications of the ACM*, vol. 35, no. 9, Sept 1992, pp. 160-165.

[30] Oscar Nierstrasz, “Regular Types for Active Objects,” *Proceedings OOPSLA ’93, ACM SIGPLAN Notices*, vol. 28, no. 10, Oct. 1993, pp. 1-15.

[31] Oscar Nierstrasz, “Composing Active Objects,” *Research Directions in Concurrent Object-Oriented Programming*, G. Agha, P. Wegner and A. Yonezawa (Ed.), MIT Press, 1993, pp. 151-171.

[32] Oscar Nierstrasz and Laurent Dami, ‘‘Component-Oriented Software Technology,’’ *Object-Oriented Software Composition*, O. Nierstrasz, D. Tsichritzis (Ed.), Prentice-Hall, 1995, to appear.

[33] Michael Papathomas, “A Unifying Framework for Process Calculus Semantics of Concurrent Object-Oriented Languages,” *Proceedings of the ECOOP ’91 Workshop on Object-Based Concurrent Computing*, M. Tokoro, O. Nierstrasz, P. Wegner (Ed.), LNCS 612, Springer-Verlag, 1992, pp. 53-79.

[34] Michael Papathomas, “Language Design Rationale and Semantic Framework for Concurrent Object-Oriented Programming,” Ph.D. thesis No. 2522, Dept. of Computer Science, University of Geneva, 1992.

[35] Benjamin C. Pierce, “Programming in the Pi-Calculus — An Experiment in Concurrent Language Design,” PICT Version 3.4c tutorial, ftp://ftp.dcs.ed.ac.uk/pub/bcp/pict.tar.Z, University of Edinburgh, March, 1994.

[36] Davide Sangiorgi, “Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms,” Ph.D. thesis, CST-99-93 (also: ECS-LFCS-93-266), Computer Science Dept., University of Edinburgh, May 1993.

[37] Richard Soley (Ed.), *Object Management Architecture Guide*, Object Management Group, Frameington, MA, Nov. 1990.

[38] Alan Snyder, “Open Systems for Software: An Object-Oriented Solution,” *ACM OOPS Messenger, Addendum to the Proceedings of OOPSLA 1993*, vol. 5, no. 2, April 1994, pp. 67-68.

[39] Bent Thomsen, “Calculi for Higher Order Communicating Systems,” Ph.D. thesis, Imperial College, London, 1990.

[40] Jon Udell, “Componentware,” in *Byte*, Vol. 19, No 5, May 1994, pp 46-56.

[41] Larry Wall and Randal L. Schwartz, *Programming Perl*, O’Reilly & Associates, Inc., 1990.

[42] Antony S. Williams, “The OLE 2.0 Object Model,” *ACM OOPS Messenger, Addendum to the Proceedings of OOPSLA 1993*, vol. 5, no. 2, April 1994, pp. 68-70.