Institute for Language, Logic and Information

OBJECT ORIENTED APPLICATION FLOW GRAPHS AND THEIR SEMANTICS

Erik de Haas
Peter van Emde Boas

ITLI Prepublication Series
for Computation and Complexity Theory CT-92-01

University of Amsterdam

The ITLI Prepublication Series

1986 86-01 The Institute of Language, Logic and Information
86-02 Peter van Emde Boas A Semantical Model for Integration and Modularization of Rules
86-03 Johan van Benthem Categorial Grammar and Lambda Calculus
86-04 Reinhard Muskens A Relational Formulation of the Theory of Types
86-05 Kenneth A. Bowen, Dick de Jongh Some Complete Logics for Branched Time, Part I Well-founded Time, Forward looking Operators
86-06 Johan van Benthem Logical Syntax
1987 87-01 Jeroen Groenendijk, Martin Stokhof Type shifting Rules and the Semantics of Interrogatives
87-02 Renate Bartsch Frame Representations and Discourse Representations
87-03 Jan Willem Klop, Roel de Vrijer Unique Normal Forms for Lambda Calculus with Surjective Pairing
87-04 Johan van Benthem Polyadic quantifiers
87-05 Víctor Sánchez Valencia Traditional Logicians and de Morgan's Example
87-06 Eleonore Oversteegen Temporal Adverbials in the Two Track Theory of Time
87-07 Johan van Benthem Categorial Grammar and Type Theory
87-08 Renate Bartsch The Construction of Properties under Perspectives
87-09 Herman Hendriks Type Change in Semantics: The Scope of Quantification and Coordination
1988 LP-88-01 Michiel van Lambalgen *Logic, Semantics and Philosophy of Language:* Algorithmic Information Theory
LP-88-02 Yde Venema Expressiveness and Completeness of an Interval Tense Logic
LP-88-03 Year Report 1987
LP-88-04 Reinhard Muskens Going partial in Montague Grammar
LP-88-05 Johan van Benthem Logical Constants across Varying Types
LP-88-06 Johan van Benthem Semantic Parallels in Natural Language and Computation
LP-88-07 Renate Bartsch Tenses, Aspects, and their Scopes in Discourse
LP-88-08 Jeroen Groenendijk, Martin Stokhof Context and Information in Dynamic Semantics
LP-88-09 Theo M.V. Janssen A mathematical model for the CAT framework of Eurotra
LP-88-10 Anneke Kleppe A Blissymbolics Translation Program
ML-88-01 Jaap van Oosten *Mathematical Logic and Foundations:* Lifschitz' Realizabiility
ML-88-02 M.D.G. Swaen The Arithmetical Fragment of Martin Löf's Type Theories with weak Σ-elimination
ML-88-03 Dick de Jongh, Frank Veltman Provability Logics for Relative Interpretability
ML-88-04 A.S. Troelstra On the Early History of Intuitionistic Logic
ML-88-05 A.S. Troelstra Remarks on Intuitionism and the Philosophy of Mathematics
CT-88-01 Ming Li, Paul M.B.Vitanyi *Computation and Complexity Theory:* Two Decades of Applied Kolmogorov Complexity
CT-88-02 Michiel H.M. Smid General Lower Bounds for the Partitioning of Range Trees
CT-88-03 Michiel H.M. Smid, Mark H. Overmars, Leen Torenvliet, Peter van Emde Boas
Maintaining Multiple Representations of Dynamic Data Structures
CT-88-04 Dick de Jongh, Lex Hendriks, Gerard R. Renardel de Lavalette Computations in Fragments of Intuitionistic Propositional Logic
CT-88-05 Peter van Emde Boas Machine Models and Simulations (revised version)
CT-88-06 Michiel H.M. Smid A Data Structure for the Union-find Problem having good Single-Operation Complexity
CT-88-07 Johan van Benthem Time, Logic and Computation
CT-88-08 Michiel H.M. Smid, Mark H. Overmars, Leen Torenvliet, Peter van Emde Boas Multiple Representations of Dynamic Data Structures
CT-88-09 Theo M.V. Janssen Towards a Universal Parsing Algorithm for Functional Grammar
CT-88-10 Edith Spaan, Leen Torenvliet, Peter van Emde Boas Nondeterminism, Fairness and a Fundamental Analogy
CT-88-11 Sieger van Denneheuvel, Peter van Emde Boas Towards implementing RL
X-88-01 Marc Jumelet *Other prepublications:* On Solovay's Completeness Theorem
1989 LP-89-01 Johan van Benthem *Logic, Semantics and Philosophy of Language:* The Fine-Structure of Categorial Semantics
LP-89-02 Jeroen Groenendijk, Martin Stokhof Dynamic Predicate Logic, towards a compositional, non-representational semantics of discourse
LP-89-03 Yde Venema Two-dimensional Modal Logics for Relation Algebras and Temporal Logic of Intervals
LP-89-04 Johan van Benthem Language in Action
LP-89-05 Johan van Benthem Modal Logic as a Theory of Information
LP-89-06 Andreja Prijatelj Intensional Lambek Calculi: Theory and Application
LP-89-07 Heinrich Wansing The Adequacy Problem for Sequential Propositional Logic
LP-89-08 Víctor Sánchez Valencia Peirce's Propositional Logic: From Algebra to Graphs
LP-89-09 Zhisheng Huang Dependency of Belief in Distributed Systems
ML-89-01 Dick de Jongh, Albert Visser *Mathematical Logic and Foundations:* Explicit Fixed Points for Interpretability Logic
ML-89-02 Roel de Vrijer Extending the Lambda Calculus with Surjective Pairing is conservative
ML-89-03 Dick de Jongh, Franco Montagna Rosser Orderings and Free Variables
ML-89-04 Dick de Jongh, Marc Jumelet, Franco Montagna On the Proof of Solovay's Theorem
ML-89-05 Rineke Verbrugge Σ-completeness and Bounded Arithmetic
ML-89-06 Michiel van Lambalgen The Axiomatization of Randomness
ML-89-07 Dirk Roorda Elementary Inductive Definitions in HA: from Strictly Positive towards Monotone
ML-89-08 Dirk Roorda Investigations into Classical Linear Logic
ML-89-09 Alessandra Carbone Provable Fixed points in $I\Delta_0+\Omega_1$
CT-89-01 Michiel H.M. Smid *Computation and Complexity Theory:* Dynamic Deferred Data Structures
CT-89-02 Peter van Emde Boas Machine Models and Simulations
CT-89-03 Ming Li, Herman Neuféglise, Leen Torenvliet, Peter van Emde Boas On Space Efficient Simulations
CT-89-04 Harry Buhrman, Leen Torenvliet A Comparison of Reductions on Nondeterministic Space
CT-89-05 Pieter H. Hartel, Michiel H.M. Smid, Leen Torenvliet, Willem G. Vree A Parallel Functional Implementation of Range Queries
CT-89-06 H.W. Lenstra, Jr. Finding Isomorphisms between Finite Fields
CT-89-07 Ming Li, Paul M.B. Vitanyi A Theory of Learning Simple Concepts under Simple Distributions and Average Case Complexity for the Universal Distribution (Prel. Version)
CT-89-08 Harry Buhrman, Steven Homer, Leen Torenvliet Honest Reductions, Completeness and Nondeterminstic Complexity Classes
CT-89-09 Harry Buhrman, Edith Spaan, Leen Torenvliet On Adaptive Resource Bounded Computations
CT-89-10 Sieger van Denneheuvel The Rule Language RL/1
CT-89-11 Zhisheng Huang, Sieger van Denneheuvel, Peter van Emde Boas Towards Functional Classification of Recursive Query Processing
X-89-01 Marianne Kalsbeek *Other Prepublications:* An Orey Sentence for Predicative Arithmetic
X-89-02 G. Wagemakers New Foundations: a Survey of Quine's Set Theory
X-89-03 A.S. Troelstra Index of the Heyting Nachlass
X-89-04 Jeroen Groenendijk, Martin Stokhof Dynamic Montague Grammar, a first sketch
X-89-05 Maarten de Rijke The Modal Theory of Inequality
X-89-06 Peter van Emde Boas Een Relationele Semantiek voor Conceptueel Modelleren: Het RL-project
1990 *Logic, Semantics and Philosophy of Language*
LP-90-01 Jaap van der Does A Generalized Quantifier Logic for Naked Infinitives
LP-90-02 Jeroen Groenendijk, Martin Stokhof Dynamic Montague Grammar
LP-90-03 Renate Bartsch Concept Formation and Concept Composition
LP-90-04 Aarne Ranta Intuitionistic Categorial Grammar
LP-90-05 Patrick Blackburn Nominal Tense Logic
LP-90-06 Gennaro Chierchia The Variablity of Impersonal Subjects
LP-90-07 Gennaro Chierchia Anaphora and Dynamic Logic
LP-90-08 Herman Hendriks Flexible Montague Grammar
LP-90-09 Paul Dekker The Scope of Negation in Discourse, towards a flexible dynamic Montague grammar
LP-90-10 Theo M.V. Janssen Models for Discourse Markers
LP-90-11 Johan van Benthem General Dynamics
LP-90-12 Serge Lapierre A Functional Partial Semantics for Intensional Logic
LP-90-13 Zhisheng Huang Logics for Belief Dependence
LP-90-14 Jeroen Groenendijk, Martin Stokhof Two Theories of Dynamic Semantics
LP-90-15 Maarten de Rijke The Modal Logic of Inequality
LP-90-16 Zhisheng Huang, Karen Kwast Awareness, Negation and Logical Omniscience
LP-90-17 Paul Dekker Existential Disclosure, Implicit Arguments in Dynamic Semantics
ML-90-01 Harold Schellinx *Mathematical Logic and Foundations* Isomorphisms and Non-Isomorphisms of Graph Models

Instituut voor Taal, Logica en Informatie
Institute for Language, Logic and Information

Faculteit der Wiskunde en Informatica
(Department of Mathematics and Computer Science)
Plantage Muidergracht 24
1018TV Amsterdam

Faculteit der Wijsbegeerte
(Department of Philosophy)
Nieuwe Doelenstraat 15
1012CP Amsterdam

OBJECT ORIENTED APPLICATION FLOW GRAPHS AND THEIR SEMANTICS

Erik de Haas
Peter van Emde Boas
Department of Mathematics and Computer Science
University of Amsterdam

ITLI Prepublication Series
for Computation and Complexity Theory
ISSN 0924-8374

Received February 1992

Object Oriented Application Flow Graphs and their Semantics

E. de Haas* P. van Emde Boas†

February 25, 1992

Abstract

In this paper we combine two principles that are present in software development: Application Flow Graphs and Object Oriented programming/design. Both principles originated from a practical approach to software development, and not from some mathematical concept. We constructed a language called OOAFG (Object Oriented Application Flow Graphs), that bears the main features of both principles. We provided a formal semantics for that language, because we think the only way one can properly define a language and make well founded claims about that language is when the meaning of that language is unambiguously given by a formal system.

1 Introduction

We start with a short description of the two programming styles which OOAFG attempts to unify.

1.1 Application Flow Graphs

The Application Flow Graph (AFG) paradigm originated from the data flow paradigm for programming languages (see for a nice and formal approach to Data Flow structures for example [Kah74] or [Kok88]). AFG strives for seamless integration of modelling, design and implementation. It accomplishes this by providing a method to decompose a model of reality into a design of reusable building blocks.

AFG relates executable blocks, called *components*, and handles the interaction between these blocks. It is said that a *Application Component Manager (ACM)* takes care of the actions that are needed to handle the interaction between the components, like communication and synchronization. In order to take care of the interaction the ACM should have access to *static* information of the components. For example how these components are related to each other in the graph, how an executable component can be executed and where it must be executed (distribution of processes) and which part of the output of which component should be send to which other component. For that purpose the ACM should have access to a repository where all the static information of an application is stored. Furthermore the ACM must keep

*ILLC ; Dept. of Math. and Comp. Science; University of Amsterdam; Plantage Muidergracht 24 1018TV Amsterdam NL

†ILLC ; Dept. of Math. and Comp. Science; University of Amsterdam; Plantage Muidergracht 24 1018TV Amsterdam NL

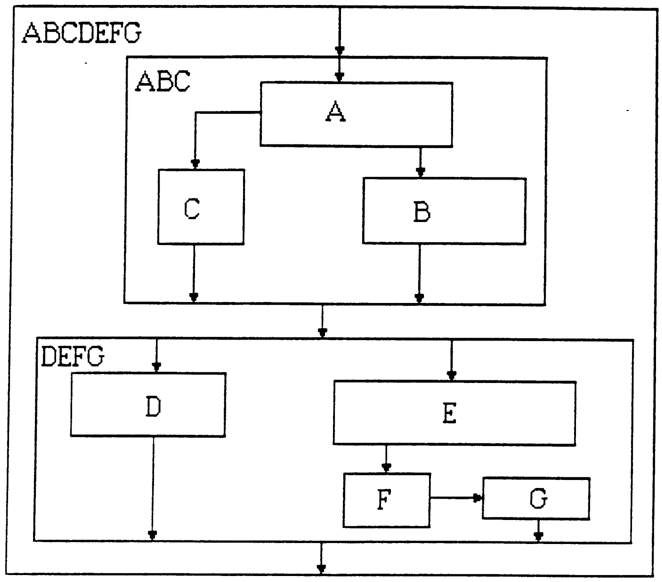

Figure 1: An Application Flow Graph

track of which component is executing at which time and which component is not (dynamic information).

The components of an Application flow graph are *completely independent* of each other, and can be of any complexity, e.g. from a simple update routine for some data structure, to a sophisticated compiler. This independence could make these components a good candidate for **reuse**.

Globally seen an AFG specification consists of a collection of components, that are related to each other as nodes in a directed graph. A component itself can be a graph of (sub)components. This way one can construct a layered AFG specification. At the lowest level in the layering the components should be executable. The directed edges in the graph represent the control and data flow of the specified application. Along the edges *data packets* are send from one component and the other. The graph structure of the AFG method supports **parallelism** and and **layering** and capitalizes on **modularity**. Figure 1 illustrates the idea.

1.2 The Object Oriented Zoo

There exist many points of view on what is meant by the Object Oriented style of simulating the real world as in programming and modelling. One concept most of these points of view have in common is that if one knows what an 'object' is, we can define Object Oriented programming or designing as a style of programming in which a whole system is described as consisting of a collection of objects. It is difficult, if not impossible however to give a rigorous definition of the most fundamental concept *object*, but in a first approximation it can be described as an integrated unit of data and procedures acting on these data.

A methodology could safely be called Object Oriented if it is possible to describe its entities in terms of objects, classes and methods and preferably support a feature like inheritance for its objects and classes. Furthermore the objects must interact with each other via messages. This definition largely coincides with the definitions in [Mey89],[Ame86a] and [Weg89].

The practical importance of object oriented programming and design styles is rapidly growing, and object oriented programming and design languages are used more and more often.

Nevertheless the theoretical foundations of these object oriented methods is relatively unexplored. One reason for this might be the fact that object oriented programming did not evolve from a simple mathematical model of programming as one could say for functional or logic programming, but rather from a practical approach to programming supported by a vague and intuitive insight. This fact causes some problems when giving a formal description for a object oriented language, something that is very important if one wants to make well founded claims on correctness of programs written in such a language. For only a very few languages attempts have been made to give a formal description and designing a proof theory for the language (The language POOL is one (if not the only one) [ABKR86], [ABKR89], [Rut88], [Ame86b]). One observation that was pointed out in formalizing an object oriented language was that it was not inherently difficult to describe the language formally with 'standard' techniques (see for 'standard' techniques for example [Bak82]), but that problems turn up when one tries to reflect the special characteristics of object oriented languages in the description. It is not easy to make good use of the extra information that is supplied by the features of the object oriented languages like the protection of objects against each other.

Another point of discussion in the framework of the object oriented paradigm is whether object oriented programming offers anything new and whether it is a sound and useful paradigm at all. Judging the fact that a lot of effort is put in the description of an object oriented language called POOL and acknowledging that far from trivial problems had to be solved to do so, one could suspect that the concepts of object oriented programming are not *that* common. But whether it offers anything substantially new and useful to the arsenal of programming techniques will, by grace of the discourse in the community that is engaged in computer science, be a point of discussion as long as the concept of object oriented programming survives the whimsical fit of time.

2 Definition of OOAFG

In this section we will present a a language that is an object oriented extension to languages that emanate from the Application Flow Graphs paradigm. This language we call **O**bject **O**riented **A**pplication **F**low **G**raph language (abbreviated by OOAFG).

2.1 The features of OOAFG

A specification written in the language of OOAFG consists of *graphs* and *definitions of data objects*. In this section we will give an informal description of OOAFG.

2.1.1 Definition of data objects

A definition of a data object (in short *data definition*) defines a class of *data objects*. A data object consists of a collection of *fields*. Each field has a *type*: the elementary type of the data in that field. For example a field f can be of the type *number* or *string* or complex data type etc. etc..

In other words we can say that a data definition is a set of fields with types associated to these fields. This data definition describes from what types a class of data objects is built.

Data objects travel through a structure graph or in other words a graph gives the operation sequence of a data object.

2.1.2 The structure graph

The structure graph consists of a set of acyclic directed graphs. These graphs have as their nodes entities called *components* and between the components directed edges are drawn. The directed edges between the components determine a *partial order* on the components of the graph in the following manner: A component C_1 precedes a component C_2 in the partial ordering determined by a graph if there exists a directed path from C_1 to C_2 in the graph.

Components and layering. A component is a procedural entity. In the framework of the object oriented paradigm a component is a method (see section 4). A component is either an atomic entity, or a compound entity. If a component is an atomic one, it represents a basic procedural operation and we will call it an *executable* component. If a component is a compound one, it consists of a graph; we will call such a component a *structural component.*

With each component we associate a data definition. This data definition gives the properties a data object must have to be processed by that component. For an executable component such a data definition is to be *given.* For a structural component this data definition is composed from the data definitions of its nodes, the components in the graph of that structural component. We will say that the nodes in the graph of a structural component *cover* the data object of that structural component.

By allowing a component to consist of an entire graph, we introduce a form of *layering* as a feature of the language.

Sequencing and parallellism. If there exists in a graph a directed edge from component C_1 to C_2 then component C_2 succeeds sequentially component C_1. If in a graph there does *not* exist any path from a component C_1 to a component C_2 then C_1 and C_2 are *in parallel* or *collateral,* in the sense that it does not matter if C_1 is executed before, after or at the same time as C_2. Note that if two components are collateral, there does not exists any relation between them in the partial order that is given by the graph.

Choice and nondeterminism. An other construct that should be present is *choice* or *nondeterminism.* In general terms there must exist some construct that describes that not all of the existing paths in the graph will be traversed all the time. The most straightforward way to achieve this kind of nondeterminism is to allow more outgoing edges for one node (component), and interpret these outgoing edges as *possible* extensions of the control flow. In other words, every outgoing edge of a node can be traveled along (by a data object that has visited the node), but it does not have to be that case. So when a node has n outgoing edges, and a data object travels along this node, there can be 0 or 1 or 2 or ... or n different edges that at the same time (parallel) are extensions of the control flow. Which edges and how many edges will be traversed after executing a component, is totally dependent of what is happening *inside* a component, and therefore transparent to the OOAFG model. For OOAFG this choice is nondeterministic, i.e. every choice is possible, but only one choice of the flow will actually be made.

Iteration. We want to allow *iteration,* that is in terms of graphs, we want to allow *loops* in the structure graph of OOAFG. We will not actually draw the loop edge, but mark a component *'repeatable'* if we want it to be possibly iterated. When a node is marked repeatable then it will be executed one or more times, before the flow of control for a data object continues. The number of iterations is not determined (nondeterminism)

We certainly can not allow cycles in general, because cycles in the structure graph would destroy the ordering relation we defined directly from the graph. The notion of *before* and *after* (before $\preceq$ after) will have to be totally different because it will be destroyed when we allow cycles. We can illustrate this by an example:

Example 2.1 *Consider the following sentence:*

After I have eaten my lunch I bring my plate to the dishwashers.

In this sentence the notion of after *nicely follows the ordering of time. But if we reconsider and realize that we will also eat lunch tomorrow, the notion of* after, *behaving as the plain ordering in time, fails, and we may end up never bringing our plate to the dishwashers.*

We also do not need arbitrary cycles in our graphs. We can omit arbitrary cycles and only allow loops without loss of generality. It is shown by Böhm and Jacopini [BJ66] that formation rules for composition and iteration are sufficient to express every flow of control one can possibly think of in terms of our components. This means that the given composition rules in OOAFG are sufficient.

Synchronization. The partial order that is defined by the structure of the graph determines the flow of control of the components: **A component can be executed iff there do not exists any predecessors of that component that are not finished yet.**

It can very well be the case that some component (node) can only be started when *several* other components are finished. In other words a node can have more than one incoming edge. These incoming edges are interpreted in such a way that an execution flow can be continued, if all components that are on a path to this node are already executed for some data object. Because we introduced nondeterminism (there exist 'possible' extensions) it can be the case that not all the paths to a component (node) will be traversed.

For each data object we will allow the control flow to continue at (node) component C if there exist no more components (nodes) on the paths to C that *can* be executed by this data object before executing C.

We call this feature *synchronization.*

2.2 The language of OOAFG

An OOAFG specification consists of specification of a component, A component consist of an **OOAFG-graph** and a **data definition**. It is sufficient to see a specification of an application as the specification of a component, because an application itself is seen as a component consisting of subcomponents.

Data definitions: A data definition D defines a class of data objects consisting of a collection of *fields* with their *type*, i.e.

$$D \subseteq \textbf{FIELDS} \times \textbf{TYPES}$$

where **FIELDS** is the set of all fields and **TYPES** is the set of all types.

OOAFG-graphs OOAFG-graphs consist of the following building blocks:

- **nodes** denoting the **components** of the application. The nodes itself consist of an OOAFG-graph. A node can be marked repeatable.

- **directed edges** denoting the control and data flow of in the OOAFG-graph.
- a *sub cover* denoting which component (node) processes which part of a data object (fields). We represent a sub cover by a function

  $$f : V \to P(\textbf{FIELDS})$$

  where V is the set of nodes of an OOAFG-graph and $P(\textbf{FIELDS})$ the set of all sets of fields of data objects. The function f maps a node to the set of fields this node (component) processes.

An OOAFG-graph for component C is built as follows:

- it consists of a directed connected acyclic graph (its **structure graph**, denoted by $G_C = (V_C, E_C)$) together with the OOAFG-specifications for every node of the structure graph of C.
- it consists of a function f_C representing a sub cover over the data definition of component C.
- and it optionally consists of the marker *repeatable*
- An OOAFG-graph is a *set*

Note that *all* components are specifies by a OOAFG graph. We will say that a component C is *executable* if its structure graph G_C is empty (i.e. $G_C = < \emptyset, \emptyset >$). When a component is executable it denotes a directly executable procedure for example on a computer.

We will call the nodes of G_C (elements of V_C) the *sub components* of C. If C' is a sub component of C then we will call C the *outer component* of C'.

We will often write $f(C')$ to denote the collection of fields associated to (sub) component C' instead of $f_{C'}(C)$ (note that C is the outer component of C').

We mark a component repeatable, if the component itself is repeatable. That way we defined this characteristic of a component *internally*. The reason that we defined the repeatable characteristic internally, and not externally, by giving an outer component the information which of its sub component is repeatable, is a technical one. If we want to express the outmost component being repeatable, we do not have to define some artificial layer around this outer component to express it to be repeatable.

The requirement that an OOAFG graph is a set is necessary and sufficient to guarantee wellfoundedness of the definition of the OOAFG graphs.

We will denote the set of Object Oriented Application Structuring specifications (i.e. the language) by **OOAFG**

2.3 Illustration of OOAFG

A simple order handling application example

The component HANDLE_ORDERS (figure 2) gives a simple orders application. The component ORDERS precedes all the other components in the graph. So first the component ORDERS will be executed, when a data object calls HANDLE ORDERS. We want all the subcomponents of component ORDERS to be executed, in the right ordering. After a report of the order is made one of three things can happen: The delivery for the order can be arranged (ARRANGE DELIVERY) if the goods are in the inventory, and the client is credible.

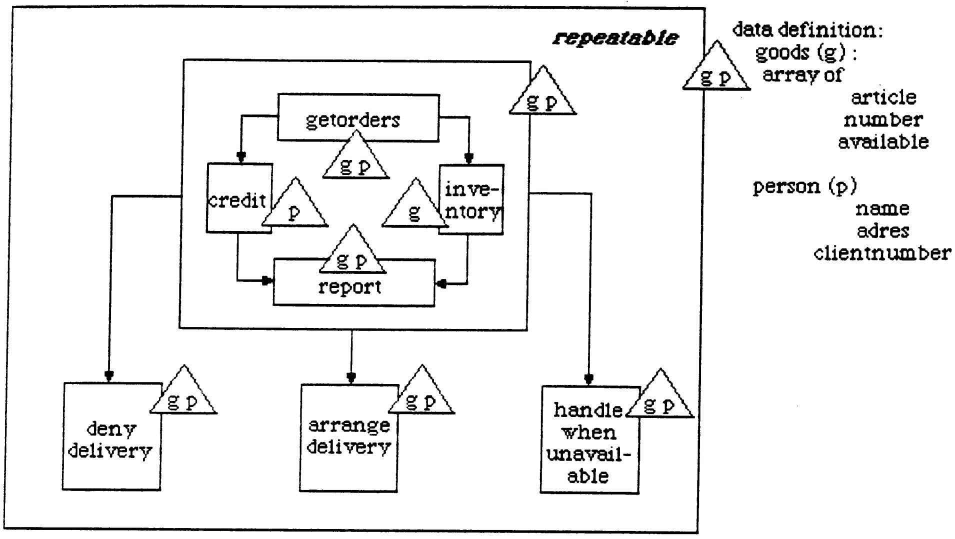

Figure 2: Example of an OOAFG

If the client is not credible, the delivery can be denied (DENY DELIVERY). If the goods are not available arrangements can be made, for example make sure that the goods will be in the inventory soon (HANDLE WHEN UNAVAILABLE), and then repeat the order.

Note that the decision along which the flow of control will continue is made *inside* a component, by triggering some of its outgoing edges (or the outgoing edges of its outer component). A so called Application Component Manager takes care of sending the data from one component to an other starting the components and checking whether the safety conditions for the data hold.

3 Semantics of OOAFG

Below we will present the semantics of OOAFG. The semantics describe what *happens* executing a OOAFG specification in terms of transitions.

3.1 Preliminaries

3.1.1 Operational Semantics

A semantics is a mapping from a syntactic domain to a semantic domain. The semantics we will use for our language is called an *operational semantics* and is based on a *transition system.* Transition systems were first used by Hennessy and Plotkin [HP79], [Plo81], [Plo83]. A transition system is a deductive system based on axioms and rules that specify a *transition relation.* In order to introduce this kind of system we shall first explain what a *transition* step is. Therefore let us consider the set **Configurations** consisting of tuples:

$$< s, I >$$

which consist of a statement s (s is a word in the language that we give a semantics for) and some amount of information I that has been collected until now. (The actual configurations we will use for OOAFG are more complicated; these configuration tuples only serve as an illustration). A transition describes what a statement s in our language can do as it next step. The intuitive meaning of the transition:

$$< s_1, I_1 > \rightarrow < s_2, I_2 >$$

is: executing s_1 *one* step with information I_1 can lead to a new amount of information I_2 with s_2 being the *remainder* of s_1 still to be executed. Note that in general there are different transitions possible, given some tuple.

To define our operational semantics we use a transition system, which is a syntax driven deductive system for proving transitions. This system consists of *axioms* and *rules.* The axioms tell us what we consider basic transitions and are of the form $C_1 \rightarrow C_2$, with C_1 and C_2 in **Configurations**. The rules tell us how we can deduce new transitions from old one and have the format

$$\frac{C_1 \rightarrow C_2}{C_3 \rightarrow C_4}$$

with $C_i (1 \leq i \leq 4)$ in bf Configurations. The meaning of this rule is: if the upper transition holds, then the lower transition also holds.

Rules and Axioms together determine a *transition relation*

$\rightarrow \in$ **Configurations** $\times$ **Configurations**

being the set of all transitions that are derivable in the system. The derivable transitions of the system are transitions that either are axioms or are deducible from the axioms using the rules.
It has to be remarked that some parts of the axioms and rules will be unspecified (i.e. given by a variable). Then such a rule or axiom stands for a whole collection of axioms or rules.

Informal example:

$$< \text{print}(3), I > \rightarrow < \varepsilon, I \cup \{\text{the number 3 is printed}\}$$

where I is unspecified, stands for all the axioms where I is *any* collection of information I_i in the domain for information and ε stands for the empty statement.

Given a certain transition system we consider transition sequences

$$< s_1, I_1 > \rightarrow < s_2, I_2 > \rightarrow \ldots$$

such that for all $n > 0$ the following is derivable:

$$< s_n, I_n > \rightarrow < s_{n+1}, In + 1 >$$

Now we can give a meaning to a program P by defining its semantics being (for example) the set of all possible transition sequences

$$< P, I_0 > \rightarrow \ldots$$

where I_0 is a basic amount of information, which has to be defined beforehand and P is the program text.(I_0 for example consists of the type declarations of the program)

3.1.2 Variant notation

In the course of this text I will make use of the *variant notation* to indicate a change in some function.

Let ρ be a (possibly partial) function then $\rho\{y/x\}$ is defined by:

$$\rho y/x(z) = \begin{cases} \rho(z) & \text{if} z \neq x \text{ (possibly undefined)} \\ y & \text{if} z = x \end{cases}$$

If $A = \{(x_1, y_1), ..., (x_n, y_n)\}$ and $(x_i \neq x_j for 1 \leq i, j \leq n$ and $i \neq j)$ is some finite collection of pairs, we mean by $\rho\{A\}$ the following variant of ρ

$$\rho\{y_1/x_1\}...\{y_n/xn\}$$

Observe that this definition is independent of the ordering of the tuples (x_i, y_j)

3.1.3 Power set

Let A be a set, then $P(A)$ denotes the set of all subsets of A (the powerset of A) and $P_{\text{fin}}(A)$ denotes the set of all *finite* subsets of A.

3.2 Auxiliary Definitions

3.2.1 Labeled statements

The labels identify a data object. α, β, ... denote *labels*. We denote the set of all labels by **Labels**. A *name of a component* is a statement. We define labeled statements as being a tuple $< \alpha, s >$ where α some label We denote by **LSTAT** the set of labeled statements.

3.2.2 Data and values

To keep track of what values the fields of the data objects have, we will use a function of the following signature:

$$\textbf{Labels} \mapsto (\textbf{FIELDS} \mapsto \textbf{VALUES})$$

This function assigns to each data object a function that maps the fields of a data object to a value. We will call a function of the above signature a *data state*. We denote the set of all data states by Δ with typical element δ. We will call $\delta(\alpha)$: **FIELDS** $\mapsto$ **VALUES** the data state for object α.

To denote a change in the data state we use the variant notation (see preliminaries). A change in the data state is given by the (partial) function $C(\alpha, \delta)$: **FIELDS** $\mapsto$ **VALUES**. Given a component name C an object α and a data state δ, $C(\alpha, \delta)$ determines a set of field/value pairs[1] that give the changes in the values of some fields of object α. The domain of $C(\alpha, \delta)$ is a subset of $f(C)$, where f is the *cover* of the OOAFG specification of the outer component of C.

Because $C(\alpha, \delta)$ is a function it holds that $(x, y) \in C(\alpha, \delta)$ and $(x, z) \in C(\alpha, \delta)$ iff $y = z$, so we may assume for $C(\alpha, \delta) = \{(x_1, y_1), (x_2, y_2), \ldots\}$ that all x_i are different. Thus we can denote with the variant notation a change in the data state for an object α by $\delta(\alpha)\{C(\alpha, \delta)\}$. A change in a data state δ is then denoted by $\delta\{\delta(\alpha)\{C(\alpha, \delta)\}/\alpha\}$, meaning that the data state is

[1] A function f can be seen as a set of pairs where $(x, y) \in f$ iff $f(x) = y$

changed in such a way that the function that δ assigns to an object α is updated with $C(\alpha, \delta)$. We will abbreviate this by

$$\delta\{C(\alpha, \delta)\}$$

We will also use a function Initial(α), that assigns to fields of α an initial value. We will handle this function in the same manner as we did with $C(\alpha, \delta)$.

We also define a predicate called *changes*. We say *changes*(C, field$_a$) only if C covers field$_a$ (i.e. field$_a \in f(C)$ for f the cover of the outer component of C. If field$_a \in f(C)$ then field$_a$ can be in the domain of $C(\alpha, \delta)$ (i.e. $C(\alpha, \delta)$ changes field$_a$).

3.2.3 Configurations

We will formulate the semantics of OOAFG in using operational semantics. An operational semantics consists of a transition system, that defines a relation between **configurations.**

A configuration in our transition system will consists of three parts:

- A set of **labeled statements.**
- A **data state** that assigns values to the fields of data objects
- An **OOAFG specification** that consists of a OOAFG-graph and a data definition for the data objects.

in other words

$$\textbf{CONFIG} = P_{fin}(\textbf{LSTAT}) \times \Delta \times \textbf{OOAFG}$$

Remark: We will often write a configuration down like this: $< X \cup \{< \alpha, C >\}, \delta, \text{ooafg} >$. We assume then that $< \alpha, C > \notin X$. We can call $< \alpha, C > \in X$ a *component instance of component C for the object α.*

3.2.4 A partial-ordering on the nodes of an OOAFG-graph

We construct a partial ordering on the components of an OOAFG-graph. That is on the components of the structure graph and on the components of the structure graphs of the components if this structure graph etc. etc..

Because we allow iteration it is possible that more than once the same component can be executed. In order to make things go well, we have to define some ordering between the subcomponents of a component and the component itself, because for one object it is possible to execute a structural component more than once and therefore it is possible that for that object the component itself together with some of its sub components can occur in the same set of labeled statements.

Let AV_{CP} denote the set of all the sub components of CP and of all the sub components of these sub components etc. etc..

Let AE_{CP} denote the set of all the edges of G_{CP} and of all the edges of the the graphs of the sub components of CP etc. etc. .

We construct a partial ordering $(AV_{CP}, \preceq)$ directly from its OOAFG-graph.

We define the relation $\preceq$ on AV_{CP} as follows:

for all $C, C' \in AV_{CP}$

1. $C \preceq C'$ if $(C, C') \in AE_{CP}$
2. $C \preceq C$
3. $C \preceq C'$ if there exists a $C'' \in AV_{CP}$ such that $C \preceq C''$ and $C'' \preceq C'$
4. if $C \preceq C'$ and $C \neq C'$ then
   (a) for all sub components $C_i \in V_C$ holds $C_i \preceq C'$
   (b) for all sub components $C'_i \in V_{C'}$ holds $C \preceq C'_i$
   (c) for all $C_i \in V_C$ and all $C'_i \in V_{C'}$ holds $C_i \preceq C'_i$ (this rule is superfluous because it already follows from the above two plus transitivity (c))
5. .for all $C' \in V_C$ holds $C' \preceq C$

We could have restricted rule 5 for components that are repeatable (i.e. 5. for all $C' \in V_C$ holds $C' \preceq C$ *if* C *is repeatable*) since the rule is used only for this special case; this restriction however turns out to be unnecessary.

It is easy to see that $(AV_{CP}, \preceq)$ is a partial-order. Because of (2) and (3) reflexivity and transitivity are automatically satisfied by the $\preceq$ relation. Because the graphs are all acyclic, and because the relation between a component (node) and its contained sub components (nodes) is one-way (by rule 5) The $\preceq$ relation is also antisymmetric in the sense that if $C_1 \preceq C_2$ and $C_1 \neq C_2$ then $\neg(C_2 \preceq C_1)$

We will use the partial order to determine which component is to be executed before or after or in parallel to which other component. We say $C \preceq C'$ if C is to be executed before C', $C' \preceq C$ if C is to be executed after C' and $C/botC'$ if C is to be executed in parallel with C' (i.e. $C \bot C'$ iff $C' \bot C$ iff $\neq (C \preceq C' \vee C' \preceq C \vee C = C')$).

The following assertion holds for a component to be allowed to be executed: A data object α can execute a component C considering a set of labeled statements X **iff** there does not exist any component C' to be executed in the set of labeled statements X by the same data object α that precedes α in X. In other words $< \alpha, C >$ can be executed considering the configuration $< X \cup \{< \alpha, C >\}, \delta, \text{ooafg} >$ if

$$\neg\exists < \alpha, C' > \in X(C' \preceq C \text{ and } C' \neq C)$$

3.2.5 the 'safe' predicate

We only allow changes in the values of a data object when it is *safe* to make them. It is safe to change some part of a data object only if it is not possible that the same part of that data object can be executed at the same time, for then we have inconsistent data. Considering a configuration $< X \cup \{< \alpha, C >\}, \delta, \text{ooafg} >$, it is safe to execute $< \alpha, C >$ (i.e. it is safe for a component C to change some part of the data object α) if in X there does not exist any collateral component C' $(C \bot C')$, that is labeled with the same label (i.e. same data object α) and covers a field of the data object that is also covered by C. So

$$\text{safe}(C(\alpha, \delta), X)$$

iff

$$\forall < \alpha, C' > \in X(C \bot C' \rightarrow \neg\exists \text{field}_a \in f(C)(\text{changes}(C', \text{field}_a))$$

3.2.6 The nondeterminism

As stated before in OOAFG we allow nondeterminism. In other words if a node has several outgoing edges, only a subset of these edges has to be traveled along by a data object that has visited that node. This amounts to the phenomenon that considering a node C and an object α, only a subset of the nodes that are incident with the outgoing edges of C will be executed by (on) α after executing C. We will describe this phenomenon by considering only a connected sub graph of an OOAFG-graph representing a structural component. This connected sub graph we call a *sub-OOAFG-graph.*

Let **G** be the OOAFG graph for some component C consisting of a structure graph G_C and a cover f_C. We distinguish a subset of the nodes in G_C, the set of *begin nodes of* C or *begin components of* C, being those nodes that have no ingoing edges in structure graph G_C. We define a sub-OOAFG-graph as follows:

Definition 3.1 H is a *sub-OOAFG-graph of* G if it satisfies the following condition:

- Let $G_C = (V_C, E_C)$ and let I_C denote the set of *begin nodes* of G_C. Let $H_C = (V_H, E_H)$ be a structure graph for H and I_H the set of begin nodes for H_C. Then
  1. $I_H \subseteq I_C$
  2. H_C is a connected sub graph of G_C such that all the nodes of H_C are on a directed path starting from a begin node.
     (i.e.$\forall v \in V \exists b \in I_H \exists v_1, ..., v_n \in V[(b, v_1), (v_1, v_2), ..., (v_{n-1}, v_n), (v_n, v) \in V]$)
- f_H is a cover for H such that for all $C' \in V_H(f_H(C') = f_C(C'))$ (i.e. f_H is f_C restricted to the nodes of H).

It is easy to see that taking a sub-OOAFG-graph of a structural component describes the phenomenon of nondeterminism as we need it. Traveling through the connected sub-OOAFG-graph is equivalent with not traveling along all edges in the whole graph.

3.2.7 The transition system

Let ooafg be the OOAFG-specification of component CP, and AE_{CP} the collection of all the edges in the OOAFG-graph of CP.

1. *execution of a structural component*

$$< X \cup \{< \alpha, C >\}, \delta, \text{ooafg} > \rightarrow < X \cup Y, \delta, \text{ooafg} >$$

where

- C is structural (i.e. if G_C is the structure graph of C then G_C is not the empty graph)
- $\neg \exists C'(< \alpha, C' > \in X \wedge C' \preceq C \wedge C' \neq C)$ (i.e. there is no predesessor of C in X with the same label)
- $Y = \bigcup_{z \in Z} < \alpha, z >$ where Z is the set of nodes of a sub-OOAFG-graph of component C

Executing a structural component amounts to replace the instance of the structural component by the instances of the components of a sub-OOAFG-graph of the structural component. A component C can be executed if there do not exist any other component instances of the same data object that precedes C.

2. *execution of an executable component*

$$< X \cup \{< \alpha, C >\}, \delta, \text{ooafg} > \rightarrow < X, \delta', \text{ooafg} >$$

where

- C is executable (i.e. the structure graph if C is the empty graph)
- $\neg\exists C'(< \alpha, C' > \in X \wedge C' \preceq C \wedge C' \neq C)$
- $\delta' = \delta\{C(\alpha), \delta)\}$ (i.e. δ is updated to δ' by the execution of component C)
- safe($C(\alpha, \delta), X$) (changes are safe)

Executing an executable component amounts to process the data fields of the executing data object. The executable component C can be executed if it is safe to do so and if there does not exist any component instance for the same data object that should be executed before C.

3. *iteration of a structural component*

$$< X \cup \{< \alpha, C >\}, \delta, \text{ooafg} > \rightarrow < X \cup Y \cup \{< \alpha, C >\}, \delta, \text{ooafg} >$$

where

- C is structural and repeatable
- $\neg\exists < \alpha, C' > \in X(C' \preceq C \wedge C' \neq C)$
- $Y = \bigcup_{z \in Z}\{< \alpha, z >\}$ where Z is the set of nodes of a sub-OOAFG-graph of component C

Iterating a component amounts to executing a component and create an other instance of this component. Note that a structural component succeeds its sub components in the ordering.

4. *iteration of an executable component*

$$< X \cup \{< \alpha, C >\}, \delta, \text{ooafg} > \rightarrow < X \cup \{< \alpha, C >\}, \delta', \text{ooafg} >$$

where

- C is executable and repeatable
- $\neg\exists < \alpha, C' > \in X(C' \preceq C \wedge C' \neq C)$
- safe($C(\alpha, \delta), X$) (i.e. changes are safe)
- $\delta' = \delta\{C(\alpha, \delta)\}$

3.2.8 Semantic mapping

Before we can give the semantics of an OOAFG specification we need some definitions.

- We call $< \emptyset, \delta, \mathrm{ooafg} >$ a *final configuration.* This is justified by the observation that assuming some fixed domain of data objects , in a configuration of the form $< \emptyset, \delta, \mathrm{ooafg} >$ all the data objects that satisfy the OOAFG specification *ooafg* are either already totally processed or not processed at all.
- With $config_1 \xrightarrow{*} config_2$ we mean that configuration $config_1$ can be transferred to $config_2$ in *zero or more* transition steps ($\rightarrow$). Naturally all the transition steps are derived from (given by) the transition system.
- With $\mathbf{CONFIG}_\perp$ we denote the set $\mathbf{CONFIG} \cup \{\perp.\}$.

Now we can give the semantics of an OOAFG specification in terms of a nondeterministic configuration transformation function:

$$\mathcal{O} : \mathbf{OOAFG} \mapsto (\mathbf{CONFIG} \mapsto \mathcal{P}(\mathbf{CONFIG}_\perp))$$

where:

$$\mathcal{O}[\mathrm{ooafg}](< X, \delta, \mathrm{ooafg} >) =$$

$$\{< \emptyset, \delta', \mathrm{ooafg} > \in \mathbf{CONFIG} |\ < X, \delta, \mathrm{ooafg} > \xrightarrow{*} < \emptyset, \delta', \mathrm{ooafg} >\} \cup$$

$$\{\perp |\ \text{there exists an infinite sequence } < X, \delta, \mathrm{ooafg} > \rightarrow \ldots \rightarrow < X_n, \delta_n, \mathrm{ooafg} > \rightarrow \ldots\}.$$

Because we did not demand the number of data objects to be finite and because we allow unguarded loops in the OOAFG specification, a configuration *config* will in general be mapped to an infinite subset of $\mathbf{CONFIG}_\perp$ by the function that describes an OOAFG specification. In other words for an OOAFG specification *ooafg* the function $\mathcal{O}[ooafg]$ will map *config* to an infinite subset of $\mathbf{CONFIG}_\perp$. If we fix the number of data objects, it is easily shown that the collection of configurations that can be reached in **one** transition step starting at some specific configuration is finite, i.e. for a configuration $config_0$ the set $\{config | config_0 \rightarrow config\}$ is finite. This can be shown by proving that the number of axioms or rules that can be applied to a configuration is finite. Königs lemma then shows that $\mathcal{O}[\mathrm{ooafg}](config)$ contains $\perp$ or is finite.

The configuration transformation function associates a set of end configurations with every configuration. This way a meaning is given to a configuration. If we want to obtain a proper meaning for an OOAFG specification, we will have to consider a proper configuration that denotes an initial state of an environment that runs an OOAFG specification.

If we want to obtain a (uniform) semantics for an OOAFG specification that is independent of an environment (initial data state), we have to fix the initial δ. A proper way to do so is to demand the initial δ to have no field assigned to any value (i.e. for all fields a and for all data objects α that $\delta(\alpha)(a)$ is undefined). A consequence of that is that the specific values of the fields have to be given by the components when executing, and furthermore that there exist no predetermined initial values for the fields of the data objects.

3.3 Extensions to the semantics of OOAFG

There are several extensions on the sematics os OOAFG thinkable in order to enlarge the understanding of OOAFG and give directions towards a broader theoretical fundaments for the concepts of OOAFG.

In the transition system for describing OOAFG we treated parallellism as interleaving of actions (an action is in this context a change of the data state). We also constructed a semantics for OOAFG in which we can express parallellis as *true* parallellism (i.e. more that one action can take place at one moment in time) or even maximal parallelism (all the actions that could possibly take place in parallel do take place at exactly the same time).

An other variant for the sematics of OOAFG we constructed, is to describe the semantics is term of *histories* in the spirit of [BKMOZ85]. This approach gives the possibility to model side effects of the execution of a component (i.e. a component does not only change the data state but also produces some side effect like printing or bleeping).

We also studied the sematics of a subset of the flow graphs, the series parallel graphs. Thise graphs are interesting because these graphs have a linear syntax where the description of such a graph can be mapped to a mathematical description of a graph in a compositional manner.

We will not present these three extended semantics in this paper, due to lack of space.

4 How OO is OOAFG?

4.1 Towards Object Oriented Application Flow Graphs

A method or language that could safely be called object oriented (OO) must consist of objects, classes, methods and messages and furthermore support inheritance. To be a little more specific about the objects we define an *object* as consisting of some (state)*data* and having a set of *methods* that can operate on this data.

The basic idea of making AFG into an object oriented AFG is to consider **data packets** that travel through the AFG-graph as the *(state)data*, the **components** that are the nodes in the AFG-graph as the *methods* and the **signals**, carrying the data packets along the edges from one component to the other, as the *messages.*

We make the following observations. Consider a message in some traditional object oriented language (for example Smalltalk [GR83]) and a signal in AFG. Conceptually there is a difference between them, because in AFG a signal is associated with a data packet and in OO the message is associated with a method. If one thinks about the practical effect the sending of an OO message or an AFG signal has on the data, there is not much of a difference, if one considers that both the method in OO and the component in AFG operate in a way on the data. The method and the component could do exactly the same thing with the data.

Let us take a look at what an object in a object oriented language does. When a method is executed it sends sequentially a number of messages to some objects. These messages invoke methods to be executed by the receiving objects. We can denote that in terms of a graph, by drawing the methods that are executed as nodes, and drawing the directed edges between them to give the sequence of the methods.

For example suppose a method sends two messages, first one to object A and then one to object B. The messages urge the receiving objects A and B to execute respectively method 1 and method 2. We can map this event directly in an AFG graph by drawing the methods that are being invoked, and associating the state data of the receiving with the signal that arrive at

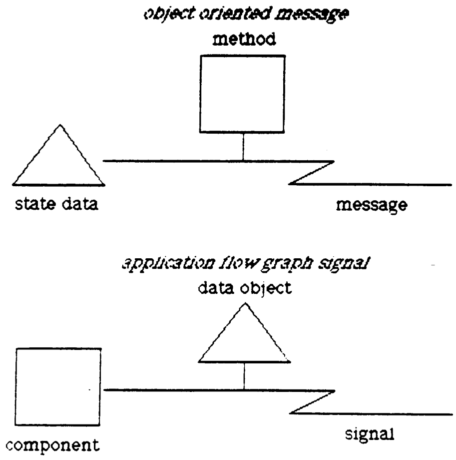

Figure 3: OO message versus AFG message

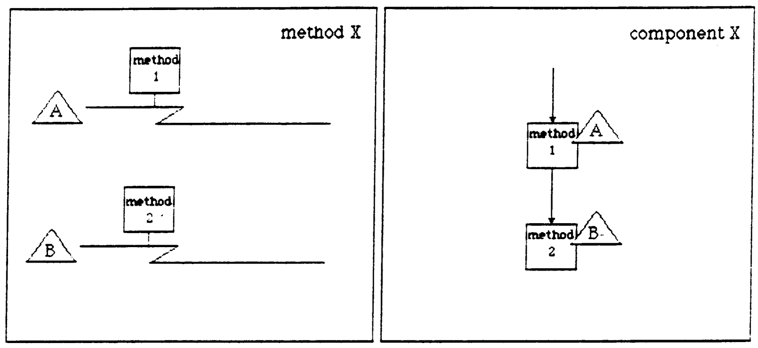

Figure 4: a sequence of messages

these methods. In a traditional OO language like Smalltalk the execution of methods is always sequential, because whenever a message is send by a method it must wait for the answer of that message before going further. In other words a after a method has send a message, it has to wait until the method that is invoked by that message on the receiving object is finished before it can send the next message. In OOAFG we can express parallelism for we can denote the flow of control in term of a graph where methods are executed in parallel. Describing a method in terms of a graph, we can express parallelism. The OO language called POOL [Ame87] also bears parallelism. The main difference between the parallelism in POOL and in OOAFG will be that that in POOL we have to state explicitly when we expect an answer back from a message i.e. we have to take care of the asynchronous message passing explicitly in the program. In OOAFG we will not have to do so

4.2 Objects, Classes and Inheritance

Each component C covers a collection of fields say $f(C) = \{a, b, c, \ldots\}$. We say that C is a method of a data object α that satisfies the data definition $\mathbf{D} = \{p, q, r, \ldots\}$ if and only if

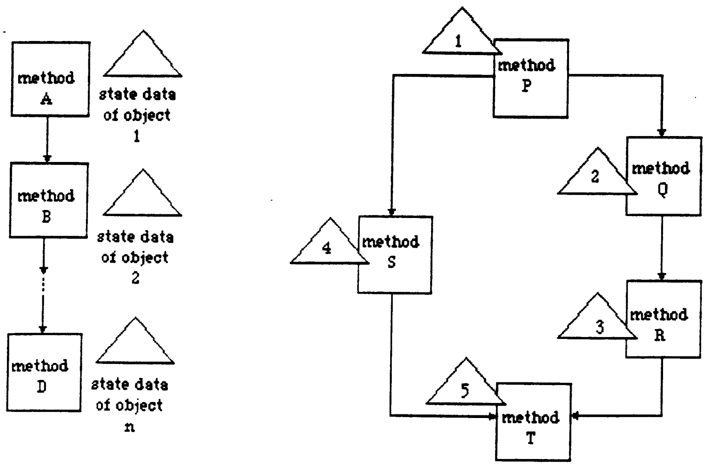

Figure 5: parallelism

$f(C) \subseteq \mathbf{D}$. This way we assosiate methods with state data, together forming the notion of object in the OO paradigm. It is easy to see that it is natural to associate a component C with a data object α if the collection of fields covered by C is contained in α, because then the object α contains the nessesary ingredients to be processed by C.

Now we have defined the objects the classes are easily defined being the data definitions for the data objects. The data definition define the structure of the (state) data of an object and also determine which methods (components) are associated with that object.

We also have inheritance. A classes $\mathbf{D}$ (i.e. a data definition) inherits the properties (i.e. methods and data structure) from all classes that consist if a subset of fields of $\mathbf{D}$. So for example the class $\mathbf{D} = \{a, b, c\}$ inherits the properties of class $\mathbf{D'} = \{a, b\}$ and from $\mathbf{D''} = \{b, c\}$ and from $\mathbf{D'''} = \{a\}$ etc. .

5 Conclusion

We succeeded in constructing a language that bears features and characteristics of both Application Flow Graphs and Object Oriented programming/design. A nice observation is that the principle that underlies this combination of OO and AFG and is expressed by the language OOAFG can be described in only four axioms. One could say that the principle that originated from combining AFG and OO is very elegantly formalizable, and therefore not difficult to comprehend. We also studied extensions of this semantics (not presented in detail in this paper) in which OOAFG proved to also have a elagant sematics in term of true parallelism, or when inspecting side effects.

Another observation we made is that the features of OO and AFG do not inherently clash with each other, although usually the conceptual points of view both principles take differ widely. We have seen that difference when comparing OOAFG with the traditional OO languages.

6 Acknowledgements

This paper refects the theoretical aspects of work I did, performing a student internship at ESAT IBM Uithoorn to obtain my master degree at the University of Amsterdam. I would like to thank the members of ESAT IBM Uithoorn and especially Ghica van Emde Boas and Gilles Schreuder, who largely contributed to the ideas presented in this paper.

7 References

[Ame86a] P. America, *Object Oriented programming: a theoreticians introduction*, Bulletin of the European Association for Theoretical Computer Science, 29, 1986, pp.69-84.

[Ame86b] P. America, *A proof theory for a sequential version of POOL*, ESPRIT Project 415 Document 188, Philips Research Laboratories, Eindhoven, The Netherlands, 1986

[Ame87] P. America, *POOL-T: A parallel object oriented Language*, in A. Yonezawa, M. Tokoro (Eds.), Object Oriented Concurrent Programming, MIT Press, 1987, pp.199-220.

[ABKR86] P. America, J.W. de Bakker, J.N. Kok, J.Rutten, *Operational semantics for a parallel object oriented language*, in Conference Record of the 13th Symposium on Principles of Programming Languages (POPL), St. Petersburg Florida, 1986, pp.194-208.

[ABKR89] P.America, J.W. de Bakker, J.Rutten, J.N. Kok, *Denotational semantics of a parallel object oriented language*, Information and Computation vol. 83, pp.152-205, 1989

[Bak82] J.W. de Bakker, *Mathematical theory of program correctness*, Information and Control, vol.54, 1982, pp.70-120.

[BKMOZ85] J.W. de Bakker, J.N. Kok, J.-J.Ch. Meyer, E.-R. Olderog, J.I. Zucker, *Contrasting themes in the semantics of imperative concurrency*, Current Trends in Concurrency (J.W. de Bakker e.a. eds.), lecture notes in computer science 244, Springer 1985.

[BJ66] C.Böhm, G. Jacopini, *Flow-diagrams, Turing Machines, and Languages with Only Two Formation Rules*, Comm. ACM 9 5 ,May 1966, pp. 366-371.

[GR83] A. Goldberg, D, Robson, *Smalltalk-80: The language and its implementation*, Addison-Wesley, Reading, MA, 1983.

[HP79] M.C.B. Hennessy, G.D. Plotkin, *Full abstraction for a simple parallel programming language*, Proceedings of the 8th MFCS (J. Becvar ed.), LNCS 74 Springer 1979, pp.108-120.

[Kah74] G. Kahn, *The semantics of a simple language for parallel programming*, Proceedings Information Processing (Rosenfeld ed.), pp.471-475, North Holland, 1977.

[Kok88] J.N. Kok, *Dataflow semantics*, Technical report CS-R8835, Centre for Mathematics and Computer Science, Amsterdam, 1988.

[Mey88] Bertrand Meyer, *Object Oriented Software Construction*, Prentice Hall 1988, ISBN 0-13-629049-3.

[Plo81] G.D. Plotkin, *A structural approach to operational semantics*, Technical Report DAIMI FN-19, Aarhuis University, Computer Science department, 1981.

[Plo83] G.D. Plotkin, *An operational semantics for csp*, in D. Bjørner, editor, Formal Description of Programming Concepts II, pp. 199-233, North Holland, 1983.

[Rut88,] J. Rutten, *Semantic correctness for a parallel object oriented language*, Report CS-R8843, Centre for Mathematics and Computer Science, Amsterdam, November 1987.

[Weg89] Peter Wegner, *Learning the Language*, BYTE, march 1989.

The ITLI Prepublication Series