LiTH-IDA-R-88-21 ISSN-0281-4250

# INSTITUTIONEN FÖR DATAVETENSKAP
## UNIVERSITETET OCH TEKNISKA HÖGSKOLAN
## LINKÖPING

# A Paradigm for Distributed System Design and Test

by

Johan Fagerström

Lars Strömberg



RESEARCH REPORT
PELAB, August 1988

*Postadress:*
Institutionen för datavetenskap
Universitetet i Linköping och
Tekniska Högskolan
581 83 Linköping

*Mailing address:*
Department of Computer and
Information Science
Linköping University
S-581 83 Linköping, Sweden

# The Department of Computer and Information Science
# Linköping University

## PhD theses:

* Available at: University of Microfilms Intl., 300 N. Zeeb Road, Ann Arbor, MI 48106, USA.

( *Linköping Studies in Science and Technology. Dissertations.* )

| | |
|---|---|
| *No 77 | **Östen Oskarsson:** Mechanisms of Modifiability in large Software Systems, 1982. |
| No 94 | **Hans Lunell:** Code Generator Writing Systems, 1983. |
| No 97 | **Andrzej Lingas:** Advances in Minimum Weight Triangulation, 1983. |
| *No 109 | **Peter Fritzson:** Towards a Distributed Programming Environment based on Incremental Compilation, 1984. |
| *No 111 | **Erik Tengvald:** The Design of Expert Planning Systems. An Experimental Operations Planning System for Turning, 1984. |
| *No 155 | **Christos Levcopoulos:** Heuristics for Minimum Decompositions of Polygons. 1987. |
| No 165 | **James W. Goodwin:** A Theory and System for Non-Monotonic Reasoning. 1987. |
| No 170 | **Zebo Peng:** A Formal Methodology for Automated Synthesis of VLSI Systems. 1987. |

## Licentiate of engineering Theses:

(*Linköping Studies in Science and Technology. Theses.*)

| | |
|---|---|
| No 48 | **Henrik Nordin:** On the Use of Typical Cases for Knowledge-Based Consultation and Teaching. 1985. |
| No 52 | **Zebo Peng:** Steps Towards the Formalization of Designing VLSI Systems. 1985. |
| No 60 | **Johan Fagerström:** Simulation and Evaluation of an Architecture based on Asynchronous Processes. 1985. |
| No 71 | **Jalal Maleki:** ICONStraint, A Dependency Directed Constraint Maintenance System. 1987. |
| No 72 | **Tony Larsson:** On the Specification and Verification of VLSI Systems. 1986. |
| No 73 | **Ola Strömfors:** A Structure Editor for Documents and Programs. 1986. |
| No 74 | **Christos Levcopoulos:** New Results about the Approximation Behaviour of the Greedy Triangulation. 1986. |
| No 104 | **Shamsul I. Chowdhury:** Statistical Expert Systems - a Special Application Area for Knowledge-Based Computer Methodology. 1987. |
| No 108 | **Rober Bilos:** Incremental Scanning and Token-based Editing. 1987 |
| No 111 | **Hans Block:** SPORT-SORT Sorting Algorithms and Sport Tournaments. 1987. |
| No 113 | **Ralph Rönnquist:** Network and Lattice Based Approaches to the Representation of Knowledge. 1987. |
| No 118 | **Mariam Kamkar, Nahid Shahmehri:** Affect-Chaining in Program Flow Analysis Applied to Queries of Programs. 1987 |
| No 126 | **Dan Strömberg:** Transfer and Distribution of Application Programs. 1987 |
| No 127 | **Kristian Sandahl:** Case Studies in Knowledge Acquisition, Migration and User Acceptance of Expert Systems. 1987 |
| No 139 | **Christer Bäckström:** Reasoning about Interdependent Actions. 1988. |
| No 140 | **Mats Wirén:** On Control Strategies and Incrementality in Unification-Based Chart Parsing. 1988. |

LiTH-IDA-R-88-21 ISSN-0281-4250

# A Paradigm for Distributed System Design and Test

by

Johan Fagerström

Lars Strömberg

Abstract: It is well known that programming and testing distributed systems can be extremely difficult. Reasons for this include non-deterministic behaviour, non-reproducability of events, complex timing of events, and complex states. This paper presents a paradigm and a system that will support a programmer when designing, programming, and testing a distributed application. Our main point is that the structure introduced by the paradigm must be kept and exploited during design, programming, and testing. One is not helped by a programming method which introduces a structure just to break it down again. We have developed a structural model based on common structures in distributed systems. This structural model is used to describe the logical relationsships between components in a system. It is also integerated into the programming environment.

*This work is supported by the Swedish Board for Technical Development, STU.*

*Postadress:*
Institutionen för datavetenskap
Universitetet i Linköping och
Tekniska Högskolan
581 83 Linköping

*Mailing address:*
Department of Computer and
Information Science
Linköping University
S-581 83 Linköping, Sweden

# A Paradigm for Distributed System Design and Test

*Johan Fagerström and Lars Strömberg*
*Programming Environments Laboratory*
*Department of Computer and Information Science*
*Linköping University, S-581 83 Linköping, Sweden*

Presented at COMPCON'Spring 88
San Fransicso

## Background

Designers of large programs traditionally start with high-level structural design. People introduced to programs also generally try to understand global structures before looking at sub-components. Structures and understanding of structural relationships is thus of vital importance for design and programming. Enforcing programs to conform to the designed structures not only guarantees up to date documentation but it also simplifies testing. This paper presents a structural model of distributed applications based on a few observations of common structures in such systems. The model is very general but specific enough to allow us to integrate it into a programming environment. The programmer can thus use structural relationships introduced by the designer when debugging. For instance, he can change viewpoints of the system consistent with abstraction levels in the design. This 'meta-level' debugging is a necessary methodology since debugging of distributed systems is complicated by introduced and natural non-deterministic behavior, complex causal relationships between events, complex timing of events, and complex states. Our approach concentrate on what we belive to be the toughest problems: large (and distributed) system states and change of design and environment.

## Introduction

The **PEPSy** project (**P**rogramming **E**nvironments for **P**arallel **S**ystems)[1,2] is an attempt to provide programmers with a suitable paradigm and appropriate tools for design and test of distributed applications. By "paradigm" we understand a conceptual framework for structuring ideas and concepts when solving problems. Our main point is that the structure introduced by the paradigm must be kept and exploited both during design and testing. One is not helped by a programming method that introduces a structure in the design phase just to break it down again later. Several formal models for describing parallel systems exist,

for instance CCS[3] and CSP[4]. Both can be used to describe communicating processes, concepts like encapsulation, observation and tracing, and even system design. However, they are primarily models to be used in the specification and formal analysis of systems. They do not support later phases, such as testing, especially well. We have developed a structural model that explain distributed systems in terms of processes, logical channels, interfaces and abstract objects. This structural model is used to describe the logical relationships between components in the system. At the same time the model is integrated into a programming environment. It supports distributed system construction in several ways:

- Documentation of system components and their relationships can be done in a consistent and intuitive way. It is focused around a graphical representation of the design.
- The paradigm can be used to enforce a limited and structured mechanism for incremental changes in the system. There are two reasons for this. First, the paradigm results in hierarchical designs. Secondly, this hierarchy is enforced. We have made 'change' a design issue.
- Programming environment tools can be improved since more knowledge of the system can be built into them. The main reason for this is the fact that knowledge introduced by the designer and the programmer is kept and exploited by the programming environment.

Our model strongly resembles the HPC - model[5,6]. Both models have the same purpose: to use process structures explicitly in the design of distributed systems.

**A paradigm for design and test of distributed applications**

Summary

We base our model on processes which are combined into so called software ICs. (We acknowledge Brad J. Cox[7] for this appropriate name, we have previously used various other names). The nature of processes is language dependent. They have an interface and they communicate via logical channels. These channels are uni-directional. A software IC is a set of communicating processes encapsulated into a black box by means of a so called control unit. The control unit is responsible for linking and un-linking logical channels inside the software IC. It is also used to construct well-defined states locally inside the Software IC, so called clean points[8]. A complete description of the model and its implementation and integration with the programming environment is given in [9].

Processes

The first architectural component in the paradigm is the **process**. Our processes

corresponds to the usual concept of logical process. The paradigm is language independent. The only requirement is that processes must be able to "communicate" with the outside world. This can be done either explicitly, using for instance SEND and RECEIVE primitives, or implicitly, dependent on the language used. A processes is created as an instantiation of a class. A class is a template that describes a generic process. On instantiation a few specializations on the process are always done. At least, the process is given a globally unique name. This name is known to its creator and the process itself. Names are used for resource management, testing, and monitoring the system, not for binding communication partners. It is also possible to initialize local state or instantiate types. For instance, a queue template can be used to generate a queue-process with buffer elements of a certain type (SIMULA-syntax[10]):

```
ref (queue) queueInstance#1;
queueInstance#1 :- new queue (numberOfElements,typeOfElements);
```

Interfaces & channels

Processes communicate with each other using message passing through well-defined **interfaces**. The message passing is between interfaces, the coupling between the interface and the associated process is where the language independent model meets a language. An interface is always created together with a process (or more generally, together with an object). An interface consists of a fixed number of **ports**. From the outside a port can be seen as an abstraction of a function implemented by the associated object. Logical channels can be set up between ports. Messages are sent on these channels asynchronously. Ports are named. The extra level of indirection introduced by using portnames instead of direct naming simplifies process migration and location transparency. This is not an implementation issue since entries can by simulated by direct naming and calculated GOTOs.

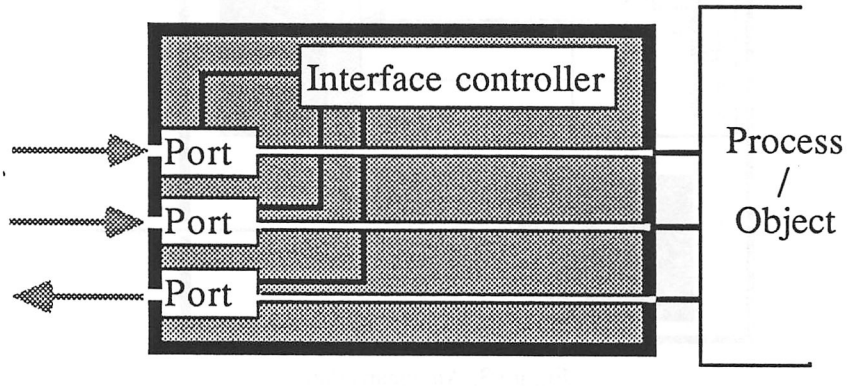


Figure 1: An interface

Figure 1 shows our view of an interface. This particular interface consists of three ports and an interface controller. The interface controller will intercept messages sent to the

interface itself, e.g. a request to link a port to some other interface. We have also indicated that two ports are used for input and one for output. Keeping to the queue example, the three ports can be named `controlPort`, `queuePut`, and `queueGet` respectively. From the process point of view, entries (defined inside the process and connected to the interface) are used for sending and receiving messages. Processes have thus no direct control over communication paths, this simplifies testing, process migration and fault-handling. An example of a physical realization of a port is shown in figure 2.

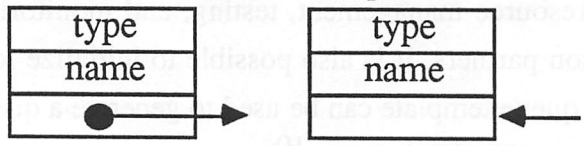


Figure 2: A port implementation

The port consists of three fields: a type, a name, and a register. The name is a constant. The register can either be used as an input register or as a placeholder for an output pointer. The output pointer either points to another port or it is `nil`. Incoming data will temporarily be stored in an input register. If it is not used it will contain `nil`. Values sent or received on `nil`-valued ports are lost. If the output pointer is writeable it will be possible to dynamically change communication partners.

Encapsulations

Abstraction is an important tool often used in design and programming. The model introduces **encapsulations** as an abstraction mechanism. The user can specify a border around an arbitrary set of objects. This border together with a name defines a new object. Figure 3 is an example of an encapsulation with three component object, one of which is itself an encapsulation.

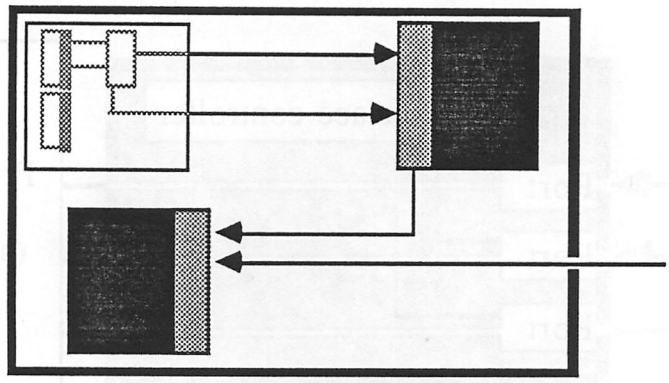

Figure 3: An encapsulation

An encapsulation is not an object in the model, it is a tool used in the programming environment. This mechanism is very influential on the debugging of the system. We

describe this idea here since it is reflected in the model by our next type of object. One can trace and examine an encapsulation as a single object. (Within reasonable limits.)

Software ICs

An encapsulation is transparent. One can observe and even change sub-components. It is also passive (as an object). In order to make it opaque we must transform it into an active object. Logically, an encapsulation can be turned into a black box by introducing a **control unit**. (In practice one does not start from an encapsulation, since they represent a user-defined categorization of instances, not a class). This new type of object is called a **Software IC**. A control unit is thus a part of a Software IC. It will set up and tear down logical channels between interfaces in the Software IC, so called **internal interfaces**. The control unit also controls the interface to the environment (the **external interface**). A software IC is opaque, its internal structure can not be observed or manipulated from the outside. It can not be distinguished from a single process. (Our implementation guarantees this by enforcing scope rules for process naming consistent with the design). In figure 4 an encapsulation (figure 3) has been turned into a software IC by introducing a control unit. We have also included a new logical channel from the environment, a channel directly to the control unit. It could represent a control channel used for sending interrupts, debug commands, etc.

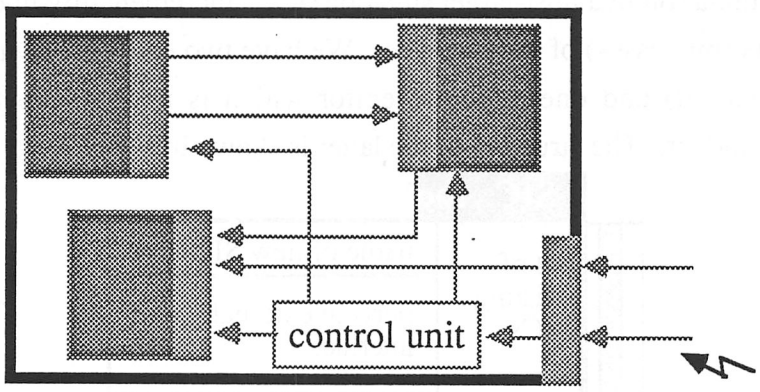


Figure 4: A software IC

The control unit has the following responsibilities:

- creating new instances of objects and their interfaces
- requesting termination or suspension of objects
- setting up and tearing down logical channels between objects
- setting up and tearing down logical channels between objects and the external interface.

The control unit is the only object in the system that can directly manipulate other objects. In this way we allow limited structured changes in the system. Unlimited creation,

suspension and termination of objects in a distributed system very quickly leads to chaos.

**Programming environment tools**

Distributing systems introduce a number of new dimensions to the system design problem. For instance, designers must take time delays and limited bandwidth into account. Distributed systems also introduce a number of new types of possible bugs (in particular bugs introduced by failing nodes), and debugging is further complicated by non-reproducibility of events, non-determinism and complex timing of events. If robust, reliable, and correct programs are to be constructed, we must handle this extra complexity compared to sequential systems. We will examine a number of programming environment tools in light of our model in particular and distributed systems in general.

Editor, compiler, interpreter, linker, and loader

Traditional tools like these play an important role for program development. They must be re-evaluated in the distributed case. For example, the optimizing pass in compilers might try to add code to increase parallelism and thus performance[11]. "Objects" also provide an appropriate component for editing, compiling, and distributing. Incrementality can thus be provided at varying granularities. A related project has presented results on statement level incremental compilation in a host-target situation[12,13]. Incrementality in our case must be on larger objects (processes) of varying sizes. We have two editors, one at a process level (language dependent) and one structure editor which is used to create software ICs (language independent). The structure of the latter is shown in figure 6.

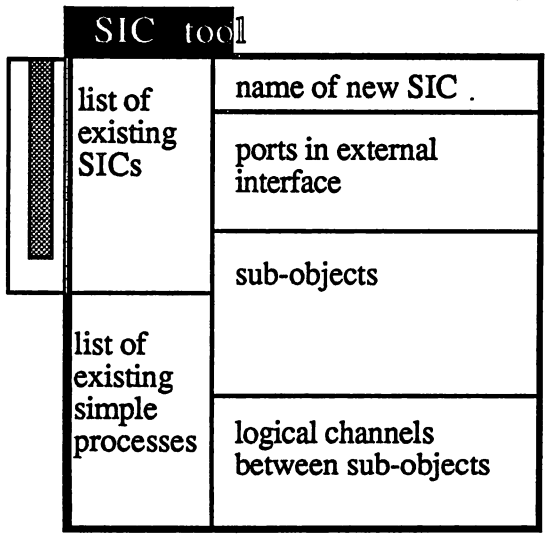


Figure 6: Editor for SIC

When entered the structure editor automatically creates a sub-class of the class **control unit**, an external interface and various control channels to the external world. The programmer

can then use the editor to record the appropriate configuration code executed by the control unit when it is started (by pointing and using menus). Commands include creating and deleting sub-objects, and linking and un-linking channels between interfaces. The code for this (the configuration code) is generated and stored in the control unit. For sub-units, the editor will automatically create the appropriate interface and link it to the control unit via control ports (using symbolic names). The programmer can incrementally add, change and delete the configuration code. He/she can also inform the editor to generate a template (or a set of templates) for sending or receiving data on channels and insert data into the configuration code. Code for tracing (conditional or unconditional) and demons can be installed from the editor on ports, interfaces and channels. The appropriate code for this will then be generated. When the editing session is finished, the software IC and the control unit as sub-classes will be stored in the class hierarchy. (One can, of course, also start with an old sub-class of software IC and change it.) The programmer can create numerous instances of the structure editor and the language editor so that various part of the design can be designed and accessed in parallel using the window system. Finally, he/she can load and start the system from the editor. This is implemented as an instantiation of the control unit which executes its configuration code. Space does not allow us to discuss resource allocation. However, the programmer has the freedom to specify (or not specify) resource requirements consistent with the model structure.

Debugging tools

The backbone of any debug system is a traditional debugger for sequential processes. It provides services such as conditional breakpoints, single-stepping and tracing. Sequential debuggers can be allocated on a one per object basis, together with a master debugger residing on the host from where the user interacts with the system. In our case we can provide more structure than this. The debug system will have knowledge about component relationships. The most important aspect of a debugger is that it presents the system in a way consistent with the users conceptual view. The debugger must thus know about encapsulations, control units, and interfaces. To support this we need a kernel which keeps track of logical structures and a name server which maps logical names into addresses. The model itself also simplifies debugging by introducing centralized control at various places: the control unit. For debugging purposes we have integrated the following tools with the structure editor:

- module interface testing
  protocols can be tested by monitoring, inserting, changing, or adding messages. This is done by dynamically "lifting" the object out of its environment from the debug tool.

- tracing & monitoring
  Information flowing on channels and interfaces can be stored on a file. The statically compiled code for tracing installed with the SIC tool can be lifted out or completely new code can be installed dynamically. (The starting time for this kind of tracing is of course dependent on the time it takes to send the command across the network). We also offer tools for browsing through the trace. For instance, information about ordering of events can be displayed (based on [14]).
- automatic surveillance
  Traditionally the term 'demon' has been used for a process that observes a database. The demon consists of a trigger and an action. The action associated with an instance of a demon is performed when the trigger condition is fulfilled. Demons recognize database changes that occur after their activation. Demons, in our case, is part of the debug tools watching and controlling objects. The user specifies the conditions which must be fulfilled for an action to occur, and the action itself. Typically, demons are used for testing relations between structures in the system (i.e., "protocols"). They can be considered as part of the debugger distributed into the system itself.
- control over the system
  Single-stepping, tracing, peek and poking on processes can be done via a traditional sequential debugger, allowing the user to introduce his own debugging techniques. This is similar to the probe-mechanism used in[15]

These tools (except the last) can be used on processes, encapsulations and software ICs. For instance, by tracing the external interface of a software IC the programmer avoids generating the (possibly large) trace of all its sub-components. Debugging can thus be done in a bottom-up fashion where single components are tested before integrating them into a software IC. When all sub-components are believed to function correctly one can study their combined behavior. All control units have built-in methods for cooperating with the debug tool. Interfaces are also prepared for tracing and demon handling. We can thus manipulate objects within the system in several ways. It is also possible to manipulate logical channels in similar ways.

**Implementation status**

A prototype implementation has been constructed in Smalltalk-80[16]. It was used as a tool when studying the model. All objects, e.g. interfaces, are represented as processes making the system very slow.Present work includes refinement of the model and the tools, and

porting the prototype onto a distributed system (using CONIC[17] on a set of SUN workstations). This version will use tables operated on by name servers to implement several model objects. This will improve performance considerable.

## Conclusions

We have presented a unifying view of distributed computations based on a structural model and briefly discussed the consequences for programming environments. The model is based on communication for process composition, and encapsulation as an abstraction tool. The programming environment benefit since more knowledge is introduced and exploited. This provides a conceptual adequate view of the system to the user. Structures are enforced by the system and protocols can be tested in a hierarchical way. The model supports both structuring (interfaces, logical channels, and processes) and layering (encapsulations and software ICs).

This work is supported by the Swedish Board for Technical Development, STU.

## References

[1] Johan Fagerström, Yngve Larsson, and Lars Strömberg
"Distributed Debugging - Collected ideas"
*Technical Report, LITH-IDA-R-86-21*, Linköping University

[2] Johan Fagerström, Yngve Larsson, and Lars Strömberg
"Debugging Techniques for Distributed Environments"
*Proceedings of the Workshop on Programming Paradigms and Programming Environments, Roskilde, Denmark*, October 1986, pp. 65-73

[3] Robin Milner
"A Calculus of Communicating Systems"
*Lecture Notes in Computer Science 92, Springer - Verlag* , 1980

[4] C.A.R. Hoare
"Communicating Sequential Processes"
*Prentice Hall International Series in Computer Science*, 1985

[5] Thomas J. LeBlanc and Stuart A. Friedberg
"Hierarchical Process Composition in Distributed Operating System"
Computer Science Department, University of Rochester

[6] Thomas J. LeBlanc and Stuart A. Friedberg
"HPC: A Model of Structure and Change in Distributed Systems"
Computer Science Department, University of Rochester

[7] Brad J. Cox

"Object Oriented Programming - An Evolutionary Approach"
*Addison - Wesley Publishing Company*, 1986
[8] J. Bryan Lyles, Zebo Peng, and Johan Fagerström
"Naming Services in a Distributed Computer Architecture"
*Nordic Symposium on VLSI in Computers and Communications in Tampere, Finland*, June 1984
[9] Johan Fagerström
"A paradigm and system for design of distributed systems"
*Forthcoming thesis, Linköping University*
[10] O-J. Dahl, B. Myhraug and K. Nygaard
"Simula Common Base Language"
*Norsk Regnecentral*, 1970
[11] P.G. Hibbard and T.L. Rodeheffer
"Optimizing for a Multiprocessor: Balancing Synchronization Cost against Parallelism"
*Proceedings of the 5th International Symposium on Programming*, Torino, Italy, 1982
[12] Peter Fritzson
"Symbolic Debugging through Incremental Compilation in an Integrated Environment"
*The Journal of System and Software* © Elsevier Publishing Co. Inc., 1983 pp.285-294
[13] Peter Fritzson
"Preliminary Experience from the DICE System - a Distributed Incremental Compiling Environment"
*Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Pennsylvania*, April 1984, pp. 113-123
[14] Leslie Lamport
"Time, Clocks and the Ordering of Events in Distributed Systems"
*CACM*, Vol.21, No.7, (1978), pp.558-565
[15] Partha Dasgupta
"A Probe-Based Monitoring Scheme for an Object-Oriented Distributed Operating System"
*Proceedings of OOPSLA '86*, pp.57-66
[16] Adele Goldberg and David Robson
"Smalltalk-80: The language and its implementation"
*Addison-Wesley Publishing Company*, 1983
[17] J. Kramer et al
"The CONIC Programming Language, Version 2.4"
*Research Report DOC 84/19*
Department of Computing, Imperial College of Science and Technology, London

Avdelning, institution, fakultet
Division, department, faculty

Department of Computer and
Information Science
Linköping University

ISBN:

ISSN: ISSN-0281-4250

Rapportnr:
Report no: LiTH-IDA-R-88-21

Upplagans storlek:
Number of copies:

Datum:
Date: August 1988

Projekt:
Project:

Titel:
Title: A Paradigm for Distributed System Design and Test

Författare:
Author: Johan Fagerström and Lars Strömberg

Uppdragsgivare:
Commissioned by:

Dnr.:
Call no:

Rapporttyp:
Kind of report:

- ☐ Examensarbete/Final project
- ☐ Delrapport/Progress report
- ☐ Reserapport/Travel report
- ☐ Slutrapport/Final report
- ☒ Övrig rapport/Other kind of report

Rapportspråk:
Language:

- ☐ Svenska/Swedish
- ☒ Engelska/English
- ☐ _____

Sammanfattning (högst 150 ord):
Abstract (150 words):

A
Abstract: It is well known that programming and testing distributed systems can be extremely difficult. Reasons for this include non-deterministic behaviour, non-reproducability of events, complex timing of events, and complex states. This paper presents a paradigm and a system that will support a programmer when designing, programming, and testing a distributed application. Our main point is that the structure introduced by the paradigm must be kept and exploited during design, programming, and testing. One is not helped by a programming method which introduces a structure just to break it down again. We have developed a structural model based on common structures in distributed systems. This structural model is used to describe the logical relationsships between components in a system. It is also integerated into the programming environment.

Nyckelord (högst 8):
Keywords (8):
Distributed systems, programming environments, debugging, structural models

Bibliotekets anteckningar

# A Selection of Previous Research Reports.

LiTH-IDA-R-88-13 **Erik Tengvald:** Ett kartorienterat planeringssystem för autonoma farkoster, en design diskussion.

LiTH-IDA-R-88-12 **Erik Sandewall, Johan Hultman, Erik Tengvald:** Software Architecture and Programming Paradigms for Robotics Applications.

LiTH-IDA-R-88-11 **Mats Wirén:** An Incremental Chart Parser for PATR.

LiTH-IDA-R-88-10 **Mats Wirén:** A Control-Strategy-Independent Parser for PATR. Revised version of paper appearing in *Proc. of First Scandinavian Conf. on Artificial Intelligence*, Tromsö, Norway, March 9 - 11, 1988.

LiTH-IDA-R-88-09 **Peter Fritzson:** Incremental Symbol Processing.

LiTH-IDA-R-88-08 **Erik Sandewall:** Formal Semantics for Reasoning about Change with Ramified Causal Minimization.

LiTH-IDA-R-88-07 **Erik Sandewall et al:** IDA Annual Report 1987.

LiTH-IDA-R-88-06 **Christer Bäckström:** A Representation of Coordinated Actions Characterized by Interval Valued Conditions.

LiTH-IDA-R-88-05 **Christer Bäckström:** Keeping and Forcing: How to Represent Cooperating Actions.

LiTH-IDA-R-88-04 **Wlodek Drabent, Simin Nadjm-Tehrani, Jan Maluszynski:** Algorithmic Debugging with Assertions. Also in *Proc. of META 88 - Workshop on Meta Programming*, Bristol, June 21-24 1988.

LiTH-IDA-R-88-03 **Staffan Bonnier, Jan Maluszynski:** Towards a Clean Amalgamation of Logic Programs with External Procedures. Accepted to the *Joint Fifth Int. Logic Programming Conf. and Fifth Logic Programming Symp.*, Seattle Washington, August 15-19, 1988.

LiTH-IDA-R-88-02 **Ulf Nilsson:** Inferring Restricted AND-Parallelism in Logic Programs using Abstract Interpretation.

LiTH-IDA-R-88-01 **Jacek Leszczylowski, Staffan Bonnier, Jan Maluszynski:** Logic Programming with External Procedures: Introducing S-Unification. A shorter version is to appear in *Information Processing Letters*, North Holland.

LiTH-IDA-R-87-26 **Jonas Löwgren:** Applying a Rapid Prototyping System to Control Panel Dialgues.

LiTH-IDA-R-87-24 **Sven Moen:** Drawing Dynamic Trees.

LiTH-IDA-R-87-22 **Lars Ahrenberg, Arne Jönsson:** An Interactive System for Tagging Dialogues. Also in *Proc. of 14th Conference of the Association for Literary and Linguistic Computing*, Göteborg 1-5 June, 1987.

LiTH-IDA-R-87-21 **Harold W. Lawson, Jr.:** Challenges and Directions in Computers and Education.

LiTH-IDA-R-87-20 **Krzysztof Kuchcinski, Zebo Peng:** Parallelism Extraction from Sequential Programs for VLSI Applications. This paper is to appear in *Microprocessing and Microprogramming, the Euromicro Journal*, 1988.

LiTH-IDA-R-87-19 **Zebo Peng:** Construction of Asynchronous Concurrent Systems from their Behavioral Specifications. Also in *Proc. of the 10th World Computer Congress*, Dublin, Ireland, Sept. 1-5, 1986.

LiTH-IDA-R-87-18 **Henrik Nordin:** Reuse and Maintenance Techniques in Knowledge-Based Systems.

LiTH-IDA-R-87-17 **Tony Larsson:** Specification and Verification of VLSI Systems Actional Behaviour This is a close version of a paper presented at *the 8th international conference on Computer Hardware Description Languages, CHDL*, 87.

LiTH-IDA-R-87-16 **Shamsul I. Chowdhury:** State of the Art in Statistical Expert Systems. Also in *Proc. of the Conference on Expert Systems and their Applications*, Avignon, May 1987.

# The Department of Computer and Information Science
# Linköping University

organizes undergraduate and graduate studies in *Computer Science*, *Telecommunication and Computer Systems*, and *Administrative Data Processing*. Research activities have an emphasis on advanced software technology and computer systems design and are organized in a number of research laboratories:

- *ACTLAB - Laboratory for Complexity of Algorithms*, which is concerned with the design and analysis of efficient sequential and parallel algorithms, and complexity theory, especially in the areas of computational geometry, data structures on bounded domains and graph algorithms.
- *ASLAB - Application Systems Laboratory,* which studies design of advanced support systems for interactive use of computers, including tools for automated construction of applications software.
- *CADLAB - Laboratory for Computer-Aided Design of Electronics*, which concentrates its research activities around tools for integrated development of hardware and software, graphics-based modelling and simulation techniques.
- *LIBLAB - Laboratory for Library and Information Science,* which studies methods for access to documents and the information contained in the documents, concentrating on catalogs and bibliographic representations, and on the human factors of library use.
- *LOGPRO - Laboratory for Logic Programming*, which concentrates its research activities around foundations of logic programming, relations to other programming paradigms and methodology.
- *NLPLAB - Natural Language Processing Laboratory*, which conducts research related to the development and use of natural language interfaces to computer software.
- *PELAB - Programming Environments Laboratary*, which works with design of tools for software development, specific functions in such tools and theoretical aspects of programs under construction.
- *RKLLAB - Laboratory for Representation of Knowledge in Logic*, which covers issues and techniques such as non-monotonic logic, probabilistic logic, modal logic and truth maintenance algorithms and systems.

## Research Reports 1988

LiTH-IDA-R-88-21 **Johan Fagerström, Lars Strömberg:** A Paradigm for Distributed System Design and Test. Also in *Proc. of COMPCON'SPRING 88*, San Fransicso, Febr-March 1988.

LiTH-IDA-R-88-19 **Lin Padgham, Ralph Rönnquist:** From a Technical to a Humane Environment: A Software System Supporting Cooperative Work. Also in *Proc. of the GDI Internationa Conference on USER INTERFACES,* Rüschlikon, Switzerland, October 20-21, 1986.

LiTH-IDA-R-88-18 **Nils Dahlbäck:** Mental Models and Text Understanding - a Commented Review.

LiTH-IDA-R-88-17 **Bengt Lennartsson:** Multi-Level Software Architectures - Industrial Experiences and Research Directions. This report is a revised version of a paper in *Advance Working Papers of The Second International Workshop on Computer-Aided Software Engineering.* Cambridge, Massachusetts. July 12-15, 1988.

LiTH-IDA-R-88-16 **Lin Padgham:** A Model and Representation for Type Information an [illegible] *Proc. of AAAI'88,* A[illegible] 1988.

LiTH-IDA-R-88-15 **Li**[illegible] An Imperative Object Or[illegible] *al Hawaii Int. Conf. on* [illegible]