LiTH-IDA-R-88-23 ISSN-0281-4250

INSTITUTIONEN FÖR DATAVETENSKAP
UNIVERSITETET OCH TEKNISKA HÖGSKOLAN
LINKÖPING

Two Contributions on Debugging Distributed Systems

by

Johan Fagerström
Yngve Larsson

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-88-23 ISSN-0281-4250

Two Contributions on Debugging Distributed Systems

by

Johan Fagerström

Yngve Larsson

Abstract: This report presents two extended abstract also published in a workshop on parallel and distributed debugging. The first by Johan Fagerström presents a general framework for distributed system design and tetsing. The second paper, by Yngve Larsson, presents ideas around a set of debug tools.

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

Enabling Structured Debugging of Distributed Systems (extended abstract)

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

Background

Debugging a parallel program can be a frustrating experience, in particular, using a system without a symbolic debugger [Fagerström 86]. The programmer is forced to use primitive techniques such as manually installed trace commands. This abstract reports on how programming environments for parallel and distributed systems can be designed to allow the programmer to think in problem terms when debugging. It is part of a larger project, called **PEPSy** (**P**rogramming **E**nvironments for **P**arallel **S**ystems) [Fagerström 88a]. The general goal of this project is to study methods and tools for design and implementation of software for distributed systems. There are basically three problems with manually installed tracing as a debugging tool:

1. it takes too long time to insert the code and re-compile the system
   -> tracing (and other tools) must be used without re-compiling the system
2. even if one tries to build and debug the system in a hierarchical fashion, bugs tend to creep up from lower levels when one is testing higher-level protocols
   -> it must be possible to move quickly between abstraction levels. It should also be possible to have a multi-abstraction view of the system
3. it is not interactive enough, one would like to test modules interactively
   -> the system must provide tools which can be used (on varying abstraction levels) to test modules

We have designed a structural model of distributed applications. It results in hierarchical designs. It is a very general model. However, it is specific enough to allow us to integrate it with a programming environment. By doing so, the programmer, and the programming environment, can use structural knowledge for debugging purposes. The model resembles both HPC [LeBlanc 85] and CONIC [Sloman 85]. A complete description of the model and its implementation and integration with a programming environment is given in [Fagerström 88b].

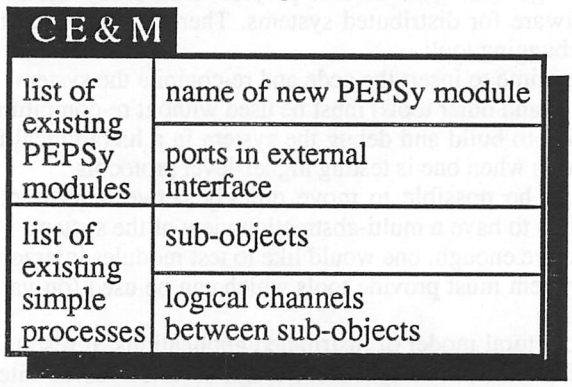
A Structural Model

Processes: The first architectural component in the model is the process. Our processes correspond to the usual concept of logical process. The model is language independent. The only requirement is that processes must be able to "communicate" with the outside world. A process 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 made. At least, the process is given a globally unique name.
Interfaces and Ports: Processes communicate with each other using message passing through well-defined 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. It 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. Conceptually, an interface controller is associated with the interface. It will intercept messages sent to the interface itself, e.g. a request to link a port to some other interface.
Encapsulations: Abstraction is an important tool often used in design and implementation. The model introduces encapsulations as an abstraction mechanism. The user can specify a border around an arbitrary set of objects. An encapsulation is not an object in the model, it is a concept used in the programming environment. We describe this idea here since it is reflected in the model by our next type of object.
PEPSy modules: 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, encapsulations can be turned into a black box by introducing a control unit. This new type of object is called a PEPSy module. A control unit is thus part of a PEPSy module. It will set up and tear

down logical channels between interfaces in the PEPSy module, so called internal interfaces. The control unit also controls the interface to the environment (the external interface). A PEPSy module is opaque. Its internal structure can not be observed or manipulated from the outside. It can not be distinguished from a single process. 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, and 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.

A System

We use two editors to build the system, one at a process level (language dependent) and one configuration editor which is used to create PEPSy modules (language independent). The structure of the latter is shown below. (Referred to as 'configuration editor and manager, CE&M).



When entered the configuration 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 (by pointing and using menus) the appropriate configuration code executed by the control unit when it is started. 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. The programmer can incrementally add, change, and delete 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 PEPSy module and the control unit as sub-classes will be stored in the class hierarchy. The programmer can create numerous instances of the configuration 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. The tool can then be used to monitor and control the PEPSy module and its sub-components. One can also zoom in on sub-objects, this will generate a new CE&M window.

Debugging tools: The backbone of any debug system is a traditional debugger for sequential processes. 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 static and dynamic 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 e.g., encapsulations, control units, and interfaces. For debugging purposes we have integrated the following tools with the configuration 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 and putting it inside a test-bed. We have developed a bottom-up debugging methodology based on protocol testing in the user-defined hierarchy [Fagerström 88b].

- tracing & monitoring
  Information flowing on channels and interfaces can be recorded. The statically compiled code for tracing installed with the tool can be lifted out or completely new code can be dynamically installed. (The starting time for this kind of tracing is of course dependent on the time it takes to send the command across the network).
- automatic surveillance
  Traditionally the term 'demon' has been used for a process that observes a database. It 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, in our case, are part of the debug tools watching and controlling objects. (As in [Smith 81]). The user specifies the conditions which must be fulfilled for an action to occur, and the action itself.
- control over the system
  Single-stepping, peeking and poking on processes can be done via a remote sequential debugger.
- probing
  A software probe can be placed on objects in the hierarchy. It will give a status report on the object.

The programmer can easily change abstraction levels in the hierarchy using the CE&M tool. Debug tools can be placed on meta-levels defined in the system. Debugging is thus done on the meta-level using structural knowledge introduced by the designer.

Implementation status

A prototype implementation has been implemented in Smalltalk-80 [Goldberg 83]. It was used as a tool when studying the model. Present work includes refinement of the model and the tools, and porting the prototype onto a distributed system (using CONIC [Sloman 85] on a set of SUN workstations).

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

References

[Fagerström 86] Johan Fagerström
"Experiences with Occam: A Simulator for Asynchronous Processes"
Hawaii International Conference on System Sciences, HICSS-19, Honolulu, Hawaii, 1986
[Fagerström 88a] Johan Fagerström and Lars Strömberg
"A Paradigm and System for Design and Test of Distributed Systems" *COMPCON spring'88*
[Fagerström 88b] Johan Fagerström
"A Paradigm and System for Design of Distributed Systems"
Forthcoming Ph.D. Thesis, Department of Computer and Information Science, Linköping University
[Fritzon 84] Peter Fritzon
"Towards a Distributed Programming Environment based on Incremental Compilation"
Linköping Studies in Science and Technology. Dissertations. No. 109
[Goldberg 83] Adele Goldberg and David Robson
"Smalltalk-80: The language and its implementation"
Addison-Wesley Publishing Company, 1983
[Hibbard 82] 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
[LeBlanc 85] Thomas J. LeBlanc and Stuart A. Friedberg
"Hierarchical Process Composition in Distributed Operating System"
Computer Science Department, University of Rochester
[Sloman 85] M. Sloman, Jeff Kramer, and Jeff Magee
"The CONIC Toolkit for Building Distributed Systems"
6Th IFAC Distributed Computer Control System Workshop, Monterey, California, may 1985
[Smith 81] Edward Tucker Smith
"Debugging Techniques for Communicating, Loosely-Coupled Processes"
Tr-100, December 1981 Computer Science Department. University of Rochester, N.Y.

A Testbed Environment for Debugging Distributed Systems

Yngve Larsson
Programming Environments Laboratory
Department of Computer and Information Science
Linköping University, Sweden

Abstract

Writing distributed programs in a hierarchical way suggest bottom-up debugging. By providing powerful tools for configuring a distributed system and mechanisms for isolating suspect sub-systems we facilitate two-phase debugging in which a possibly malfunctioning subset of processes (ideally a single process) is identified in a first phase and debugged in a testbed environment in the second phase. It is then incrementally reinserted into the executing system.

1. Introduction

Debugging and testing distributed systems poses large problems for the programmer. Reasons for this include non-determinism, name-space problems and undefined (or at least extremely complicated) global states.

The PEPSy project [Fagerström 86] is aimed at assisting the programmer in these tasks. The paradigm used for programming a distributed system draws heavily upon (among others) HPC [LeBlanc 85] and Conic [Sloman 85].

While bottom-up testing (verifying all components before composing them) is fine, experience shows that not all bugs are caught this way. Unfortunately, this is the most direct way of debugging in a distributed environment. We therefore strive for a method known as *two-phase debugging* [Garcia-Molina 84].

The task of debugging a distributed system is in many areas analogous with that of debugging a large and complex serial program, e.g. the difficulty of getting an overview of the system configuration. In a distributed environment this is further complicated by the impossibility of step-wise debugging of subsystems (except in the special case of a single process), no strict control of process behaviour (specifically the impossibility of halting all processes at a given time) and the difficulty of observing a global state.

2. The PEPSy Project

PEPSy stands for **P**rogramming **E**nvironments for **P**arallel **Sy**stems. It is an on-going research project within the Programming Environments Laboratory at the Department of Computer and Information Science, Linköping University.

The paradigm used is based upon hierarchical configuration. The basic building-blocks are processes which communicate via ports. These ports allow asynchronous communication and has infinite buffers for unprocessed messages.

A PEPSy Module is constructed by connecting processes (in a sense, "atomic modules") and/or other modules and by defining an external interface to this module. Within this module further life is handled by an agent known as the Control Unit [Fagerström 88], e.g. in order to terminate a module, a termination request must be sent to the Control Unit of that module.

The modules represent a *conceptual* grouping of functionality within a system. This means that a module may very well span several physical nodes.

3. Pinpointing the error

There are several ways of localizing a suspect error. The methods envisioned by us are primarily by tracing. We use a trace generation mechanism for writing traces at each physical node within the system. A trace inspection and verification tool is used for examining these traces. A alternative but more costly tracing facility would permit debugging with instant replay [LeBlanc 87].

Another possible way of identifying malfunctioning subsystems is to use debugging demons. These are a distributed analogy of breakpoints in a serial debugger. However, instead of halting the affected process, the demon executes its associated action, which may or may not stop the affected processes. For instance, the action may simply deliver a message to the master debugger.

4. Debugging in isolation

Once a subsystem (or singular process) is suspected to cause the observed error, phase one of the two-phase debugging is completed. The task is now to find out why and when the error occurs. There are basically two possible causes for an error in a (distributed) application: There may be an error within a process (in which case this process must be identified), or the interprocess relations may be incorrect (in which case the process connections must be corrected).

The problem of debugging a serial process is well-known, and will not be covered here. If, however, the cause of the bug is within the inter-process communication, other tools must assist the programmer.

Our tool for this task is the PEPSy TestBed, into which one may insert a PEPSy Module. This tool gives the programmer explicit control over input and output channels from the process. This enables the programmer to try out possible error cases (albeit labourously).

A crucial feature of this tool is the ability to sever the communication links between the parts being debugged and the remaining system. This is necessary in order to avoid strange effects in the untouched parts of the system while debugging. Manually constructed test cases should not propagate outside the debugged subsystem. This isolation of a subsystem is facilitated by the design paradigm of PEPSy since a hierarchical organization implicates a small interface between a subsystem and its environment.

The TestBed automatically adapts itself to the interface of the debugged module by examining the types and number of ports in the interface. Also, when invoking the TestBed on a subsystem, simple queueing processes are attached to the ports left disconnected by the separation of the debugged parts from the system. This is done in order to minimize the impact of debugging on the entire application. I.e. messages sent from the environment to the debugged system are not lost, merely delayed (this may be thought of as stretching the concept of non-deterministic communication delays).

Note that the debugged system is not physically moved. It still executes on the intended hardware, only with a different communication partner.

This tool is currently beeing implemented in Conic. It is intended to run upon a network of Sun-3 workstations.

5. Reconstructing the final system

In order to complete the two-phase debugging the corrected sub-system must be reinserted into the original system. This requires the configurations to be incremental, unless one would have to reconstruct the entire system from scratch. This is supported both by the PEPSy paradigm and by the implementation language used.

Not every distributed system takes kindly to being separated from one of its sub-components. In most cases, however, it will merely block gradually as the subsystem fails to answer. These configurations would gracefully unblock as the incrementally inserted (and hopefully correct) subsystem starts to process the pending communication.

6. Conclusions

The presented tool supports minimal impact debugging of selected subsystems in a distributed environment in the sense that the parts not being debugged may stay alive. Debugging of interprocess relations with this tool is fully consistent with the hierarchical paradigm of PEPSy.

References.

[Fagerström 86] Johan Fagerström, Yngve Larsson and Lars Strömberg.: *"Debugging Techniques for Distributed Environments"*, Research Report, LiTH-IDA-R-86-35, Linköping University.

[Fagerström 88] Johan Fagerström: *"A Paradigm and System for Design of Distributed Systems"*, PhD Thesis, To be presented, Linköping University.

[Garcia-Molina 84] Hector Garcia-Molina, Frank Germano, Jr. and Walter H. Kohler: *"Debugging a Distributed Computing System"*, IEEE Transactions on Software Engineering, Vol. SE-10, No 2, March 1984, pp 210-219.

[LeBlanc 85] Thomas J. LeBlanc and Stuart A. Friedberg: *"Hierarchical Process Composition in Distributed Operating Systems"*, Computer Science Department, The University of Rochester.

[LeBlanc 87] Thomas J. LeBlanc and John M. Mellor-Crummey: *"Debugging Parallel Programs with Instant Replay"*, IEEE Transactions on Computers, Vol. C-36, No 4, April 1987, pp 471-482.

[Sloman 85] M. Sloman, J. Kramer and J. Magee: *"The Conic Toolkit for Building Distributed Systems"*, 6th IFAC Distributed Computer Control System Workshop, Monterey, California, USA, May 1985.

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

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-88-23

Upplagans storlek:
Number of copies:

Datum:
Date: August 1988

Projekt:
Project:

Titel:
Title: Two Contribution on Debugging Distributed Systems

Författare:
Author: Johan Fagerström and Yngve Larsson

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):

~~Abstract~~ This report presents two extended abstract also published in a workshop on parallel and distributed debugging. The first by Johan Fagerström presents a general framework for distributed system design and tetsing. The second paper, by Yngve Larsson, presents ideas around a set of debug tools.

Nyckelord (högst 8):
Keywords (8):

Bibliotekets anteckningar

A Selection of Previous Research Reports.

LiTH-IDA-R-88-16	**Lin Padgham:** A Model and Representation for Type Information and Its Use in Reasoning with Defaults. Also in *Proc. of AAAI'88, American Association for Artificial Intelligence*, 1988.
LiTH-IDA-R-88-15	**Lin Padgham, Ralph Rönnquist:** LINCKS: An Imperative Object Oriented System. Also in *Proc. of the 20th Annual Hawaii Int. Conf. on System Sciences*, Hawaii, 1987.
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.

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 Laboratory*, 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-88-23 **Johan Fagerström, Yngve Larsson:** Two Contributions on Debugging Distributed Systems. Also in *Proc. of ACM SIGPLAN/SIGOPS, Workshop on Parallel and Distributed Debugging*, Madison, Wisconsin, May 5-6, 1988.

LiTH-IDA-R-88-22 **Johan Fagerström:** Design and Test of Distributed Applications. Also in *Proc. of the International Conference on Software Engineering, ICSE*, Singapore, April 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 Co

LiTH-IDA-R-88-17 **Be** itectures - Industrial Ex is a revised version of a p *nd International W ing.* Cambridge, Ma