INTEGRATED CHARGE SENSITIVE ELECTRON DETECTOR FOR ELECTRON SPECTROSCOPY

by

Mats Carlsson Göthe, Lars Asplund,
Carl-Johan Fridén and Måns Lundquist.

Institute of Physics, Box 530 S-751 21 Uppsala, Sweden.

Abstract

A new integrated, charge sensitive, multi-detector is presented. The detector consists of metal charge collecting plates, high gain amplifiers and coded, readout electronics, all integrated on the same chip. Design, simulations and layouts are discussed. The detection threshold is less then 10^6 electrons/pulse and the space resolutions is 0.15 mm.

1. INTRODUCTION

The use of position-sensitive-detectors, PSDs, has in the last years become an integral part in many spectroscopical methods. The PSD technology has diverged in a number of different measurement techniques [1]. In 1984 we presented a concept on how an integrated multiple-anode electron detector can be realized [2]. This PSD consisted of 5*10 mm² chips placed side by side forming a 120*12 mm² detector. A resolution of 150μm in two dimensions were suggested. Further work aiming to implement this circuit has been carried out by the Department of Electronics at the Institute of Technology in Uppsala [3]. This work has resulted in simulations, layout and manufacturing of a small test circuit.

The purpose of this project is to explore new concepts of PSDs, to improve the previous designs, and finally to evaluate the replacement of the existing detector equipment used at our institute for the electron spectrometers.

In this paper drawbacks in the previous designs are discussed. Suggested solutions, new simulations and designs are presented.

2. THEORY

2.1 The current detector used.

The PSD discussed in this report is intended for use in electron-, ion fragment- or Auger spectroscopy. In these applications, radiated samples ejects electrons by the photoelectric effect. The radiation may range from UV to soft X-ray. The energy of the emitted electrons are measured in an ESCA spectrometer by electron deflection in an electrostatic, hemispheric, analyser [4]. Figure 1 shows the principles of an ESCA spectrometer. Electrons with different energy are position resolved at the detector. The higher energy electrons are deflected towards the outer

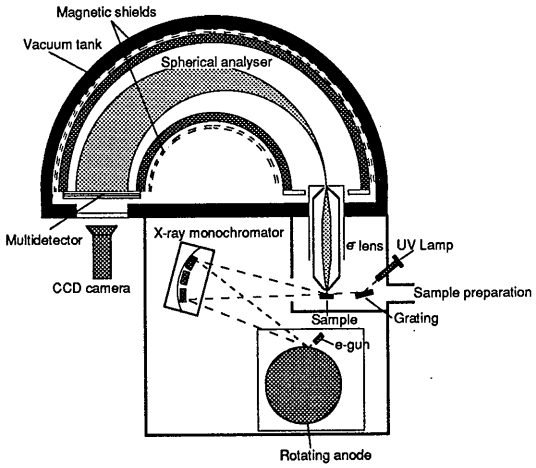


Figure 1. *ESCA spectrometer.*

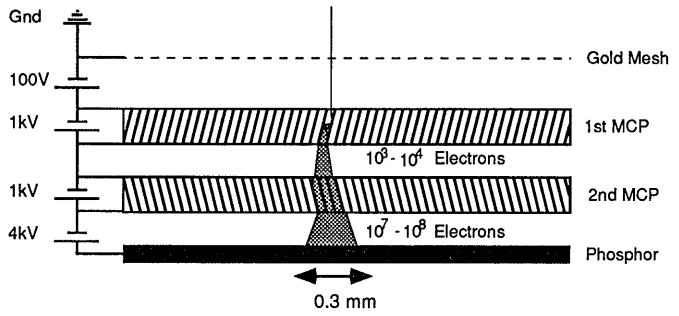


Figure 2. *The current used detector configuration.*

radius of the electrodes, the lower energy electrons towards the inner radius. It is difficult to directly detect single electrons, so they have to be multiplied.

In the currently used detector arrangement the Microchannel Plates, MCPs, in conjunction with a phosphor screen produces a visible scintillation for each electron [5]. Figure 2 shows the different parts in the current detector configuration. The MCPs in pair have a gain of 10^5 to 10^8. Higher gain requires longer recovery time, as the channels in the MCP can be regarded as charged capacitors. The gain therefore effects the dynamic range of the detector. Under normal gain conditions, 10^6, the recovery time for one channel is about 0.1s. One can see that the channels in the MCPs are tilted 13° relative to the normal of the MCP. The second MCP is rotated 180° with respect to the first plate so that the angle between the two plates becomes 26°. This is a standard way of mounting MCP pairs to reduce the ionic feedback from the second plate to the front side of the first plate. The phosphor screen is scanned by a CCD video camera so oriented that each scan line corresponds to a certain electronenergy. The peaks in the video signal are used to increment a counter. A dedicated processor reads the counter and adds the data to a spectrum buffer. The spectral data is later transferred to a host computer system [6]. Despite the isolated detector mounting and the high vacuum conditions in the spectrometer, flash-over between the MCPs or the 2nd MCP and the phosphor may occur. This is not critical in the current detectors but may be an important restriction when using the integrated detectors discussed. The currently used detector system allows a maximum overall counting rate of 10^4 - 10^5 counts per second, provided that the intensity is uniformly distributed over the detector surface. In a single channel the usable count rate is less than 1000 per second.

2.2 The integrated electron detector.

In detector technology it is desirable to integrate the detector system and the readout electronics onto the same monolithic chip. Integration gives the benefits of resolution enhancement, size and price reduction, and high reliability. The progress in integration techniques has made this possible. However, some silicon detection techniques are hard to integrate on the same chip due to compatibility problems. Substrate doping, lattice orientation are two examples where the silicon strip detectors and the circuit integration has opposite requirements for optimum performance [7]. Inter-connections between high density detector and amplifier chips introduce advanced bonding techniques [8]. To increase the active area in a

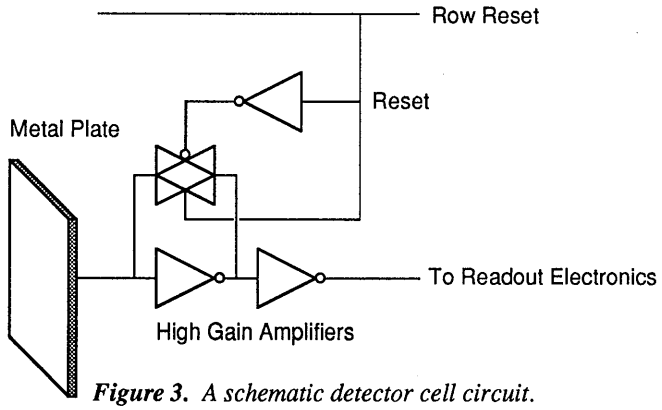


Figure 3. *A schematic detector cell circuit.*

detector circuit, and to minimize the interconnections, it is desirable to integrate the detector and readout electronics under the active elements. However, large detector areas, 120*12mm², would result in a very low yield when trying to manufacture these circuits. To increase the yield, detectors may be produced in smaller, identical, chips. Detector chips can then be mounted on a substrate in a matrix structure with repeated interconnections between the chips. If the connections are made at a higher coded level, rather than at detector cell level, the number of bonds may be reduced.

The new integrated detector is a charge sensitive , high gain, amplifier with digital, addressed, readout. The detector cell consists of a metal plate, a charge amplifier and readout electronics. Figure 3 shows a schematic detector cell circuit. The metal plate and the bulk material in the semiconductor forms a capacitance of about 0.02fF/µm². The 150*150µm metal plate gives a total capacitance, C_{plate}, of 0.45pF. When charged by an electron pulse, q_{pulse}, the plate drops in potential q_{pulse}/C_{plate}. This small voltage difference is amplified to digital signal levels by the high gain amplifiers connected to the plate.

A high gain amplifier can be constructed using two CMOS inverters in series. By adjusting the channel length or width in the inverters, the logic transition levels may be varied and the proper gain can be set. The transition level of the first inverter is set slightly below the second inverter. In figure 4 the characteristics of the two inverters are shown. The differences in transition levels are greatly exaggerated. A transmission gate connects the input and output of the first inverter and resets the amplifier into its active regions when opened. The first inverter enters a state where $V_{in}=V_{out}$. This state is in the high gain region of the inverter. The adjustment of the first inverter sets this state just below the transition region of the second inverter. A small voltage drop on the input of the first inverter is amplified which makes the output just pass the transition level of the second inverter. Figure 5 shows the amplification. The detector element is not active again until it has been reset.

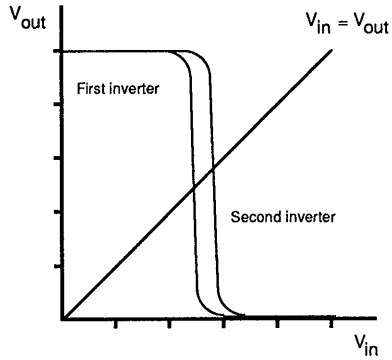


Figure 4. *The adjusted inverter characteristics.*

As previously mentioned the inverter modulation may be achieved by either adjusting the channel length or width in one of the transistors. The choice between length or width variation turns out to be important.

In normal digital CMOS application the power consumption is very low as the transistors are complementary and only one is open at a time. Current is therefore only flowing the short time when the gate is shifting from one state to the other. In the detector this is not the case. As shown in figure 5 the active region of thefirst inverter is where both the transistors are partly open. In this state the, constant flowing, current through the inverter must be considered. An increased channel width gives an increased current, while an increased channel length gives a decreased current. As a full detector circuit consists of a vaste number of detector cells, all in power consuming states, the overall current must be simulated and calculated when evaluating the usefulness of the circuit.

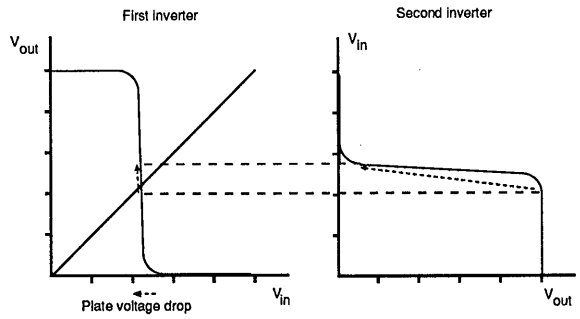


Figure 5. *Transition of the two inverters at reception of an electron pulse. After a reset opearation the firs inverter enters the $V_{in}=V_{out}$ state. This voltage is just below the transition level of the second inverter. A small input voltage change on the plate, due to an electron pulse, is amplified and raises the input of the second inverter just above the threshold.*

2.3 Readout Electronics

The principles of the readout electronics has been changed since the first proposals and implementations of the detector circuit [2,3]. Instead of a shift register, as in the previous designs, a coded bus is used. The bus is a 'wired OR'-bus. When a count is registered, a *READY* strobe signals the appearance of an unique row, Y, and column, X, pixel-address on the bus. As the detector circuits are identical and the readout electronics are regular, the circuit may easily be extended, both in hight and width. The present layout consists of a 16 * 16 matrix of detector cells, giving a 4 row and 4 column bit bus. This structure of the readout electronics gives fast output but introduces new problems. If two or more detector elements generates a ready strobe at the same time, the data is corrupted.

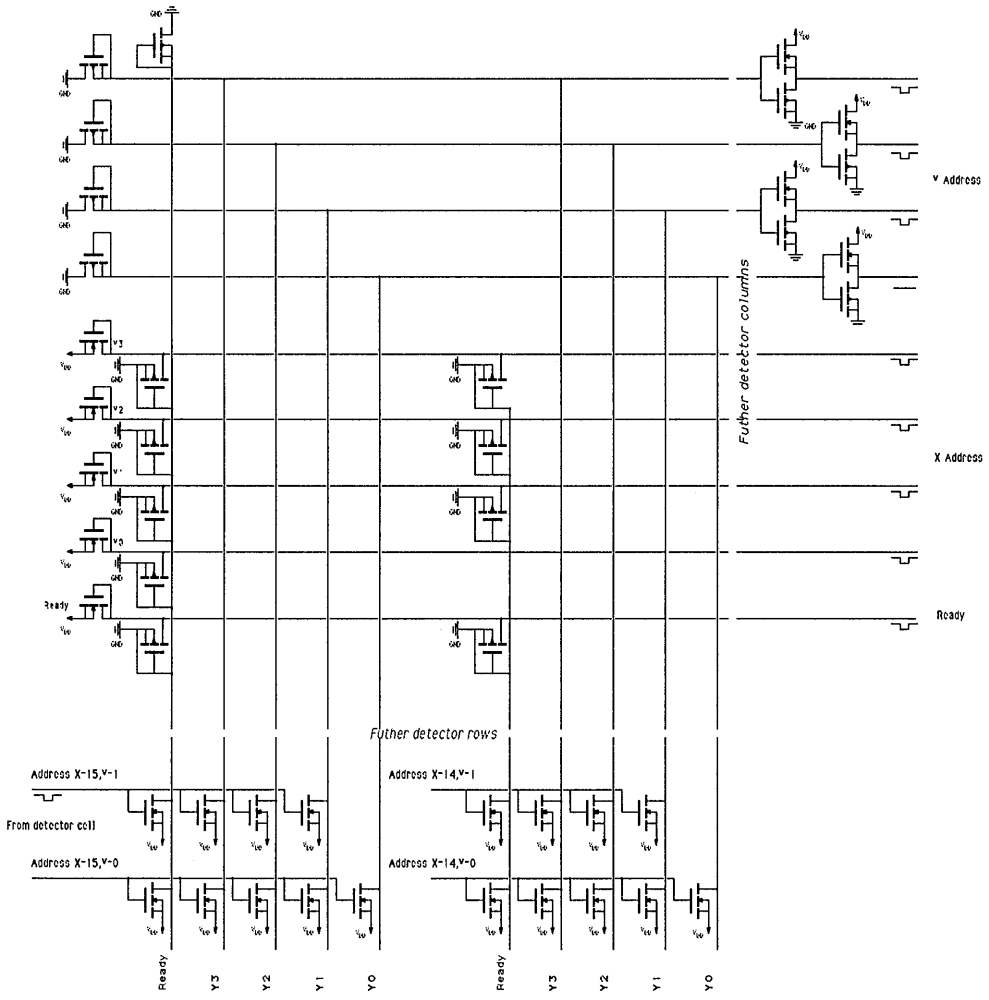


Figure 6. *The readout electronics.*

2.4 Reset network.

After a count has been detected, the detector cell must be reset. The reset pulse is given by the connecting hardware and is distributed by the, on chip, reset network.

The reset network performs two tasks, chip reset and row disable. Row disable may be used for disabling erroneous rows or to mask off parts of the detector.

The reset network consists of a static shift register, holding the row disable signal, and OR gates selecting either the row disable or the chip reset signal. Figure 7 and 8 shows the reset network. The external hardware controls the reset network through four connections, *CHIP_RESET, DATA_IN, CLOCK* and inverted *CLOCK*. The new detector gives the possibility of 2D detection. The connected hardware can convert the 2D information to energy lines though a 'look-up' table.

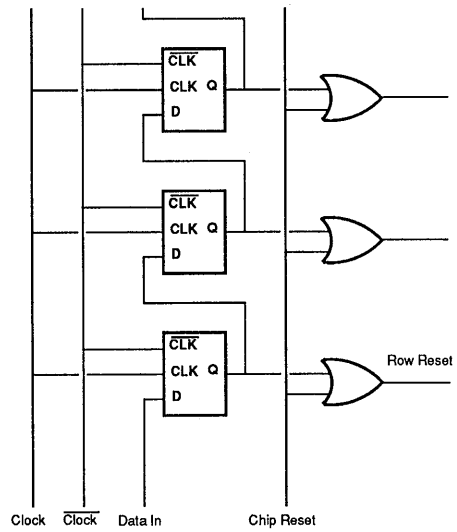


Figure 7. *The reset network.*

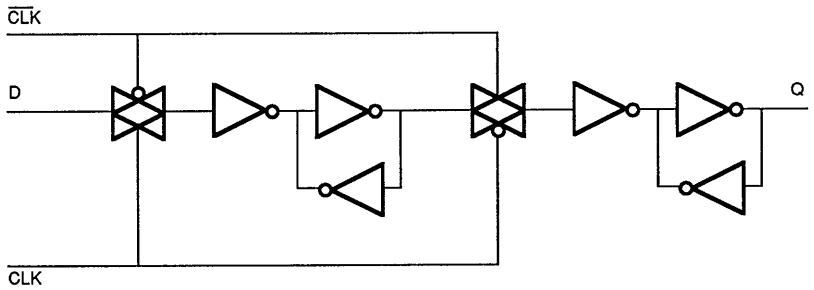


Figure 8. *The static D-latch design.*

2.5 Detector mounting.

The new integrated detector introduces a partly new mounting. As the new detector directly may detect the amplified electron pulses, the phosphor becomes obsolete. The new mounting is shown in figure 9. In the previous mounting, shown in figure 2, the accelerating field between the 2nd MCP and the phosphor have a small focusing effect on the electron pulse. The new detector does not tolerate this high voltage. The spacing between the MCP and the phosphor must be held to a minimum, reducing the divergence of the electron pulse. The spacing must, however, be large enough to allow for the bond wires.

3. RESULTS

3.1 Simulations

In order to gain knowledge on the behaviour and various parameters of the detector cell, we simulated the circuit using the SPICE simulation program [9]. The simulation has been modeled firstly by a pulsed voltage source connected via a small capacitor to the input plate capacitance and secondly by a pulsed current source with equal results. Figure 10 shows the pulsed current source model.

The design goal for the detection limit was 10^6 electrons. This gives a voltage drop of 300mV on the 0.5nF detector plate. In the simulation model a pulsed current source giving a 160μA, 1ns pulse simulating the 10^6 electrons.

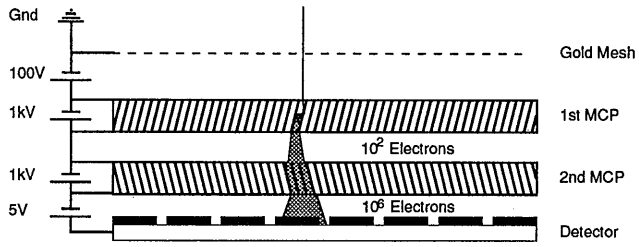


Figure 9. *The mounting of the new integrated detector.*

The amplification is set by adjusting the channel width or length of the transistors in the charge sensitive detector inverters. The degree of variation was determined by simulating the detector cell with various adjustments. The proper relation was found to be a complementary channel length of 15 to 3. The various lengths and widths are shown in figure 11. The current in one detector cell was found to be 13.5µA. This gives a power consumption of 65µW per cell. Thus, an 1024*16 detector would give a total power consumption of approximately 1W. Previously we stated that adjusting the width would increase the power consumption. Simulation shows that the current increases to 149µA. This would result in a 10W chip which is totally unacceptable and shows the importance in observing this property.

The simulation showed well known effects not discovered in the previous simulations performed [3]. When the transmission gate is opened the detector input is affected. The input may easily swing 60mV which is 20% of the minimum detection level. This is a capacitive effect produced by the voltage step applied to the two transistors, when opening and closing the transmission gate. The two transistors does however produce a complementary voltage swing. The swing amplitudes are not the same as the geometry may vary between the two transistors. By modulating the transmission gate, different amount of charge may be stored in the transistors and the effect may to some extent be balanced. Simulations showed a proper transmission gate width modulation of 2 to 1.

The simulations showed that the reset gives a time lag of 30ns before the detector is prepared for detection. A delay time of 50ns in the detector circuit and 65ns in the readout electronics. This gives a maximum counting frequency of 6MHz.

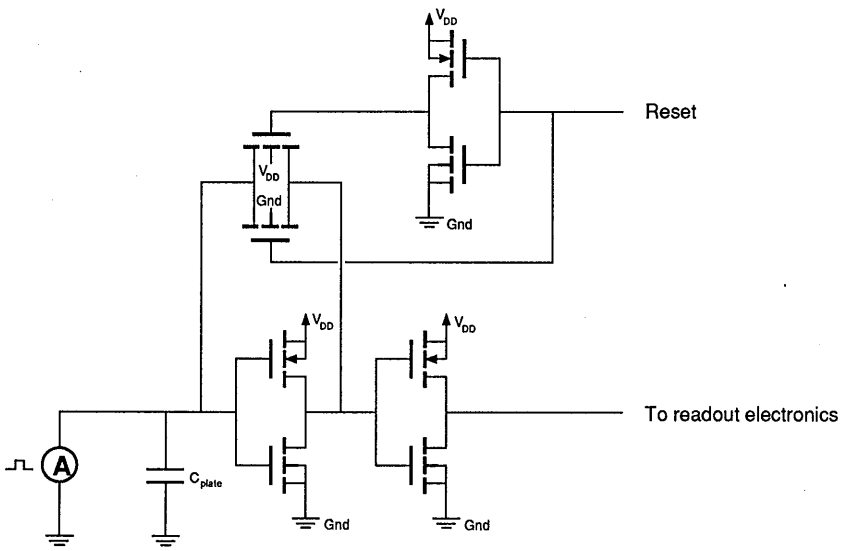


Figure 10. *The simulated model.*

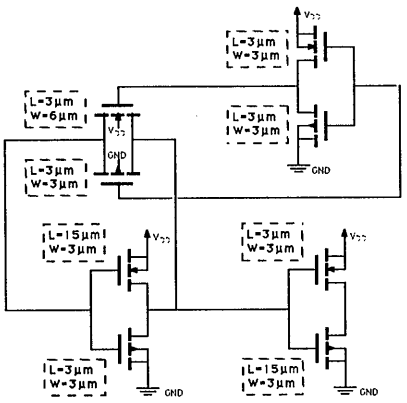


Figure 11. *Transistor length and width modulations.*

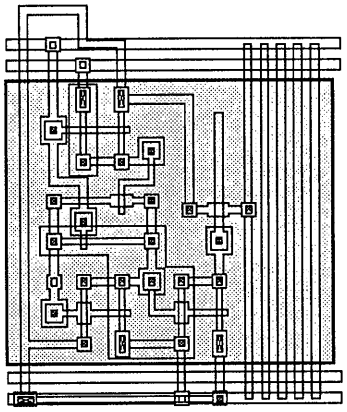


Figure 12. *The detector cell layout.*

Simulations were performed to determin the noise immunity in the voltage supply, V_{DD}. We wanted to find if a enabled detector was triggered by a change in the supply voltage. Simulations showed that a swing of 0.75V on the 5.0V V_{DD} line was accepted.

Simulation plots are shown in appendix A and the SPICE simulation file in appendix B.

3.2 Circuit layout

The detector was constructed using the 3µm, Si-gate, P-well and double metal CMOS process by NORDCHIP [10]. We used the SAGA and PAC layout programs [11,12].

Digital CMOS technology is fairly tolerant to batch variations in transistor parameters. However, using this technology for realizations of analog circuits may introduce faults. Large variations in the threshold voltage may be critical as this will affect the gain modulation of the amplifying inverters. The CMOS technology has, however, been used in previous detector designs [13].

When designing the detector cell layout the work was concentrated to minimize the used area. Minimizing the area has two major effects. It increases both the resolution and the sensitivity of the detector. The detector plate is made in the 2nd metal layer totally covering and protecting the underlying detector cell electronics. Interconnections within and between the detector cells were made using the 1st metal, doped tunnels and poly layers. Figure 12 shows the detectorcell layout.

It would be desirable to protect the input amplifiers against flash over, which easily may occur. Such a protection may consist of a diode or a transistor connecting the detector plate to ground [7]. Input protection circuits effects the input plate by increasing the capacitance and thus lowering the detection voltage step.

Grounded shielding plates are made in the 2nd metal layer covering the reset and readout electronics. These plates protects against flash over and electron pulses.

Two different chips have been constructed using the same detector cell layout. The first is a full 16*16 detector circuit with readout electronics and reset network, the other is a detector cell test circuit. On the test circuit two detector cells are integrated with access to the connecting buses and the the detector plate.

Plots showing the layout of the two chips and the detector call are supplied in appendix C. The full detector circuit is shown in a reduced 2*2 version.

4. CONCLUSIONS.

The new layout presented solves some problems but introduces new ones. One problem discovered is the incapability to handle, or detect, simultaneous electron pulses. A multiple pulse produces a erroneous cell address as the different addresses are overlaid on the bus. Another, more serious problem, is the fault tolerance. If one detector cell is 'constantly on', as an effect of an integration fault, then the whole chip is locked. Simulations has shown that the tolerance in supply voltage noise and temperature drift is good. Until fabrication, now underway, we will not know the exact measured specifications of the detector.

Further development of the detector circuit may include a new amplifier strategy. A differential charge sensitive amplifier would solve some of the drawbacks found.

5. REFERENCES

[1]. L. J. Richter and W. Ho.
Position-sensitive detector performance and relevance to time-resolved electron energy loss spectroscopy.
Rev.Sci.Instrum. 57, 1469, 1986.

[2]. L. Asplund, U. Gelius, P-A. Tove, S-Å Eriksson and N. Bingefors.
Position-sensitive focal plane detectors for electron (ESCA) spectrometers.
Nucl.Instrum.Methods.Phys. 226, 204, 1984.

[3]. J. Tirén, U. Magnusson, K. Bohlin and P-A. Tove.
A position sensitive LSI detector for low energy (ESCA) electrons.
12th Nordic Semicond. Meeting 255, 1986.

[4]. U. Gelius, L. Asplund, E. Basilier, S. Hedman, K. Helenelund and K. Siegbahn.
A high resolution multipurpose Esca instrument with x-ray monochromator.
Nucl.Instrum.Methods.Phys. 85, 1984.

[5]. E Basilier.
Multidetector systems for Electron spectroscopy
Uppsala University Institute of Physics, UUIP-1021, 1980.

[6]. L. Björnfot.
Microprocessorstyrt videokamera interface med GPIB kommunikation.
Uppsala University Institute of Physics, 1987.

[7]. G. Zimmer.
Technology for the compatible integration of silicon detectors with readout electronics.
Nucl.Instrum.Methods.Phys. 226, 175, 1984.

[8]. J. Walker, S. Parker, B. Hyams and S. Shapiro.
Development of high density readout for strip detectors.
Nucl.Instrum.Methods.Phys. 226, 200, 1984.

[9]. A. Vladimirescu, K. Zang, A.R. Newton, D.O. Pedersson,
SPICE Version 2G User's Guide.
A. Sangivanni-Vincentelli,
University of Califorina, Berkley.

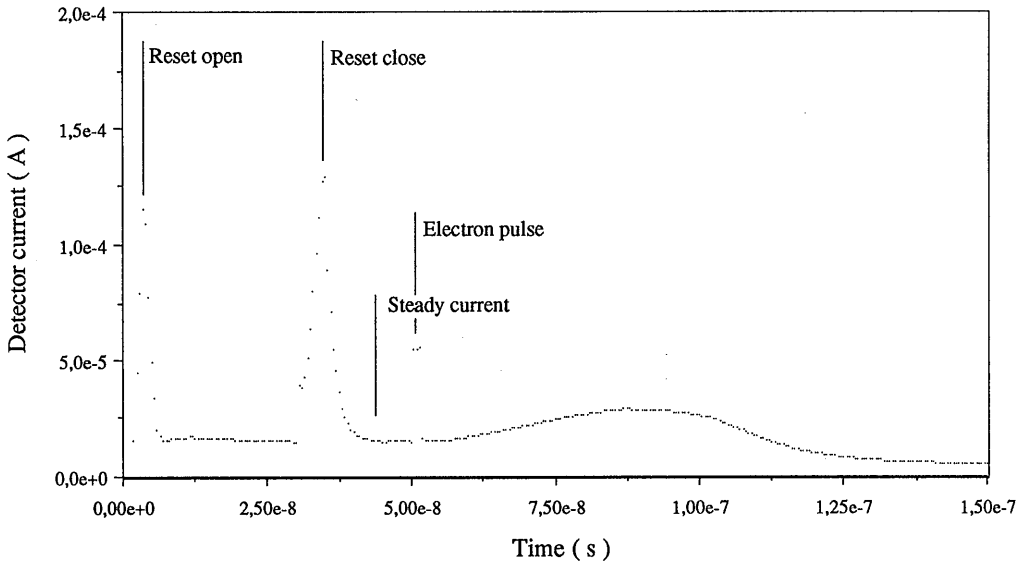
[10]. C. Svensson, University of Linköping,
L. Philipson and S. Mattisson,
University of Lund,
K. Jeppson, University of Göteborg,
Portable CMOS design rules for the Swedish Universities.

[11]. C. Hammar.
SAGA - A software package for LSI artwork design.
Institute of Microwave, Stockholm,
IM-report no. 183-1006.

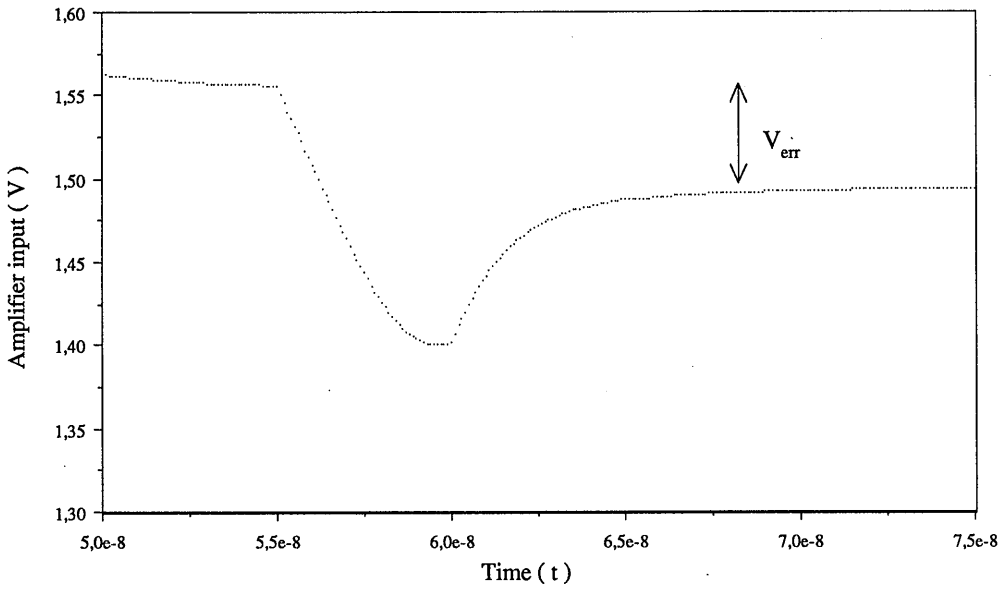
[12]. C. Hammar.
Pascal Artwork Compiler 4.1 Reference Manual
Institute of Microwave, Stockholm,
IM-report no. 184-1001.

[13]. R. Hofmann, G. Lutz, B.J. Hosticka, M. Wrede, G. Zimmer and J. Kremmer.
Development of readout electronics for monolithic integration with diode strip detectors.
Nucl.Instrum.Methods.Phys. 226, 196, 1984.

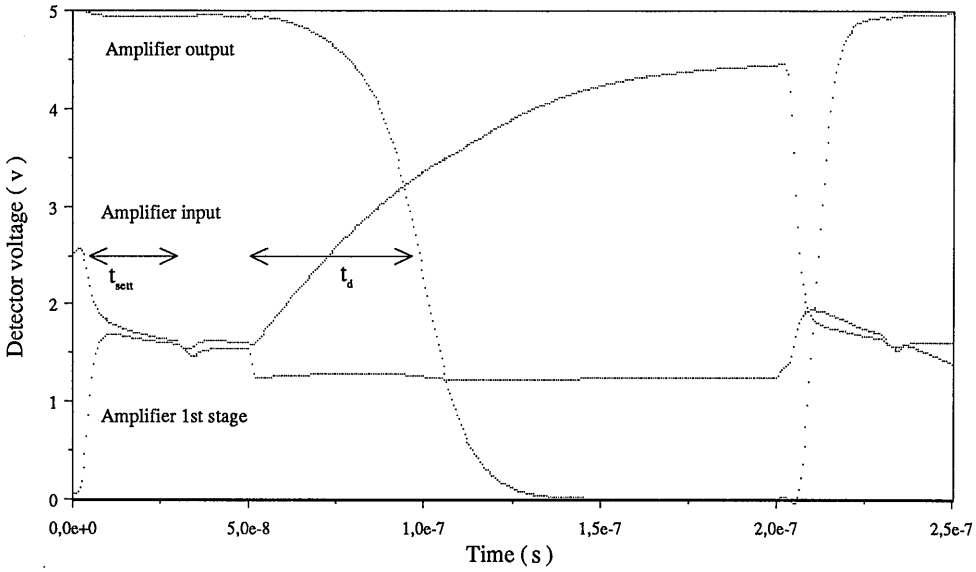
Appendix A.



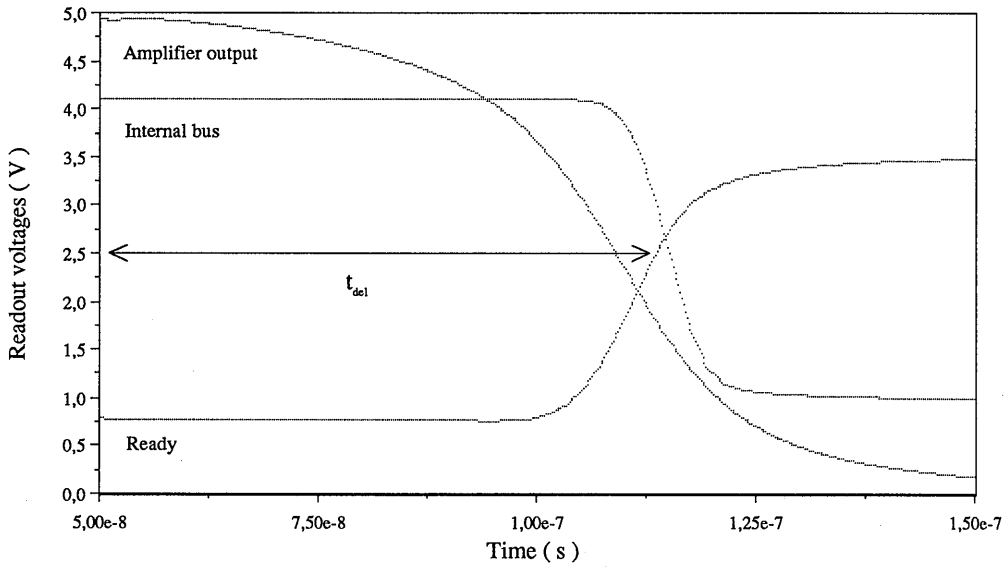
Simulation plot 1. *Simulation of the detector supply current following an event. Current peaks, up to 130μA, may occure during the reset cycle. The steady state current is found to be 15μA.*



Simulation plot 2. *Simulation of the error voltage, V_{err}, produced by the transmission gate. The static error voltage is found to be 62mV.*



Simulation plot 3. *Simulation of the detector amplifier. The voltage is simulated in three nodes, the amplifier input, the first amplifier stage and at the amplifier output.The detector setup time, t_{sett}, and the amplifier delay time, t_d, are shown. The setup time is found to be 30ns and the delay time 50ns.*



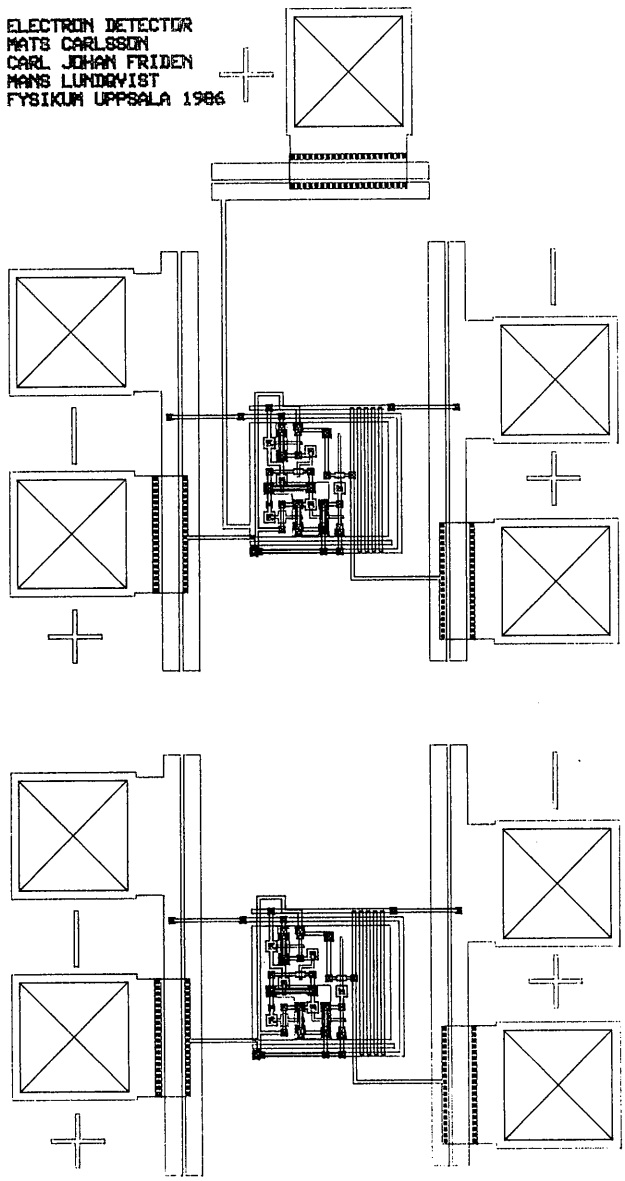
Simulation plot 4. *Simulation of the readout electronics. The simulation is performed in three nodes, the amplifier output, the internal bus, and at the READY output. The delay in the readout electronics, t_{del}, is found to be 65ns.*

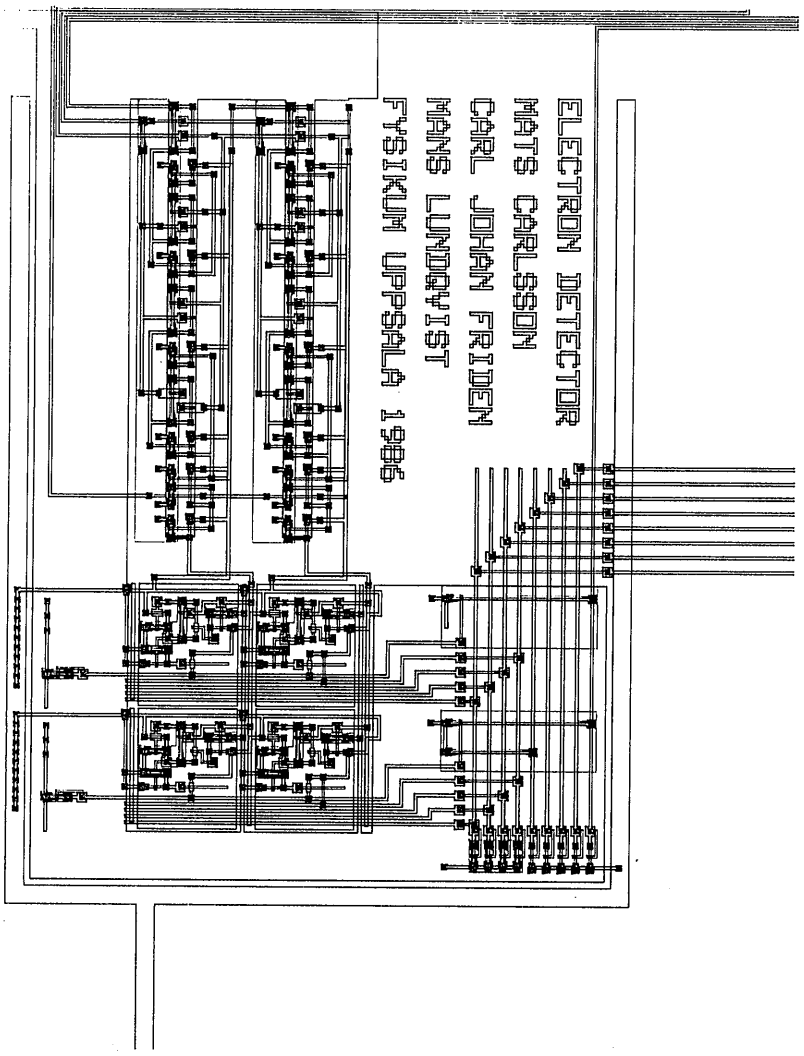
Appendix B.

```
*************************************************************
ELECTRONDETECTOR

* Translated from: det_cell  (.AF4)
* Process:         C2SG
*************************************************************
* Extracted transistors:
* ----------------------
*           drain    gate  source    bulk
Mos1           5       4       3       3  PMOS   W=3.00U   L=3.00U
+          AD=160.13P  PD=99.75U  AS=79.00P  PS=48.00U
Mos2           4       6       3       3  PMOS   W=3.00U   L=15.00U
+          AD=160.13P  PD=99.75U  AS=76.00P  PS=45.00U
Mos3           7       5       7       3  PMOS   W=6.00U   L=3.00U
+          AD=160.13P  PD=99.75U  AS=91.00P  PS=54.00U
Mos4           6       8       4       3  PMOS   W=6.00U   L=3.00U
+          AD=149.50P  PD=93.00U  AS=76.00P  PS=45.00U
Mos5           8       9       3       3  PMOS   W=3.00U   L=3.00U
+          AD=160.13P  PD=99.75U  AS=79.00P  PS=48.00U
Mos6           4       6      10      10  NMOS   W=3.00U   L=3.00U
+          AD=108.25P  PD=67.50U
Mos7           5       4      10      10  NMOS   W=3.00U   L=15.00U
+          AD=61.00P  PD=36.00U
Mos8           6       9       4      10  NMOS   W=3.00U   L=3.00U
+          AD=79.00P  PD=48.00U  AS=108.25P  PS=67.50U
Mos9           8       9      11      11  NMOS   W=3.00U   L=3.00U
+          AD=79.00P  PD=48.00U
* Extracted inter-layer capacitances:
* ------------------------------------
Cap1           9        7  5.00E-16
Cap2           9        3  5.90E-15
Cap3          11        7  1.00E-15
Cap4          11        9  1.00E-15
Cap5          11        3  2.00E-15
Cap6          10        7  5.00E-16
Cap7           6        9  3.38E-14
Cap8           6       10  1.34E-14
Cap9           6       11  5.48E-15
Cap10         10        3  1.30E-14
Cap11          6        5  2.43E-14
Cap12          6        7  2.96E-14
Cap13          6        4  3.88E-14
Cap14          6        8  2.23E-14
Cap15          6        3  6.60E-14
* Extracted capacitances (areas & perimeters):
* ----------------------------------------------
Cap16         10        0  4.31E-14
Cap17          6        0  4.67E-13
Cap18         11        0  3.93E-14
Cap19          9        0  3.60E-14
Cap20          3        0  1.16E-13
Cap21          4        0  1.66E-14
Cap22          5        0  1.18E-14
Cap23          7        0  1.41E-14
Cap24          8        0  1.03E-14
* Put Your control cards here.
VDD1  3  0  5V
VSS2  10  0  0V
VSS3  11  0  0V
```

```
.MODEL NMOS NMOS LEVEL=1
+KP=40E-6 VTO=0.9 GAMMA=0.3 LAMBDA=0.05 PHI=0.6
+LD=0.45E-6 TOX=50E-9 CGSO=3.0E-10 CGDO=3.0E-10
+CGBO=1.0E-9 CJ=0.3E-3 CJSW=0.5E-9

.MODEL PMOS PMOS LEVEL=1
+KP=15E-6 VTO=-0.9 GAMMA=0.4 LAMBDA=0.05 PHI=0.6
+LD=0.45E-6 TOX=50E-9 CGSO=3.0E-10 CGDO=3.0E-10
+CGBO=1.0E-9 CJ=0.2E-3 CJSW=0.4E-9

*DEFW compensated for oxide encroachment
.OPTIONS DEFL=3UM DEFW=1.6UM
.OPTIONS LIMPTS=2500

*  DISABLE1     9
*  DISABLE2 (INV)     8
*  OUT     5
*  INV1    6
*  INV2    4

IPULS 6 0 PULSE (0 0.1602E-3 50NS 0NS 0NS 1NS 2000NS )
VENABLE   9 0 PULSE (0 5V 0.0 5NS 5NS 25NS 200NS)
.TRAN 0.5NS 250NS

.PRINT TRAN V(6) V(4) V(5) V(9) I(VDD1)

.END
```

Appendix C.


ELECTRON DETECTOR
MATS CARLSSON
CARL JOHAN FRIDEN
MANS LUNDQVIST
FYSIKUM UPPSALA 1986


ELECTRON DETECTOR
MATS CARLSSON
CARL JOHAN FRIDEN
MANS LUNDQVIST
FYSIKUM UPPSALA 1986

VI

The Distributed Ada Run-Time System, DARTS

by

M. Carlsson Göthe, D. Wengelin and L. Asplund.
Institute of Physics, Box 530
S-751 21 Uppsala, Sweden.

Abstract

A Distributed Ada Run-Time System, DARTS, is presented. The system can be used in conjunction with a pre-partitioning as well as a post-partitioning paradigm. A single program can be partitioned to run on a loosely coupled multiprocessor system. The distributed units are tasks, task objects, packages, variables, procedures, and functions. Task objects can be dynamically distributed. High fault tolerance is assured by unit redistribution. Design decisions, implementation details and ideas are presented.

INTRODUCTION

Distribution of hardware and software for scientific, industrial, and military computer systems has significantly increased system performance. Several methods, practices, and standards exist for designing distributed hardware, but only a few such improvements have been made in the software area. The effect of the software crisis, the cost of software overwhelming the cost of hardware, is more accentuated for distributed systems in embedded applications than for other types of applications.

Hardware for distributed systems is either tightly coupled or loosely coupled. A system where processors share a common memory is defined as a tightly coupled system. Interprocess communication is performed by shared data structures in the common memory. In a loosely coupled system the memory is localized to each node, and communication is performed over some interconnecting network.

The goal of this project is to develop a distributed Ada run-time system for loosely coupled distributed systems. High fault tolerance is required. The system will be used in a testbed for embedded scientific and military applications.

Partitioning Ada programs.

Partitioning of programs is performed by pre-partitioning or post-partitioning. In pre-partitioning, the software structure is based on the hardware topology [1], [2], [3]. Hence, the software may be designed to improve the fault tolerance without considerable run-time overhead.

In post-partitioning, the design process is divided into two phases: functional design (application) and functional distribution (hardware mapping) [4], [5], [6], [7]. The partitioning information and source code are usually kept separate. Strong requirements are put on the run-time system for efficiency, especially when high fault tolerance is required.

Considerable work has been made to evaluate the proper units of distribution in the Ada language [8]. Five methods may be found; partitioning on Ada programs [9], on tasks [2], on packages [1], [10], on any part of an Ada program [4], [6], or by adding new partitioning rules and mechanisms into the Ada language [11].

Breaking a system into separate programs is the prevailing approach in the design of distributed systems [9], [12]. The method is used for Ada as well as other programming languages. The communication between the various programs is performed by calls to some added IO-packages. As the partitioning decisions are taken early in the system life-cycle, the partitioning tends to become static during the following phases and changes may require redesign of the entire system.

Another shortcoming is that Ada only performs type checking within a program, and no compiler support is given for the compatibility of the data types between programs. Furthermore, Ada defines a rich variety of mechanisms for data-flow and process control within a program. With the use of IO-packages for inter-program communication, we are restricted to subprogram calls.

The partitioning on tasks is widely spread and accepted in the scientific community [2], [13], [14]. The task is the basic structure for concurrency in Ada and therefore suitable for partitioning. The Ada Language Reference Manual [15] states that tasks may execute in a multicomputer environment, but there are only limited language facilities for distributing these tasks on different processors. Furthermore, the task is not a compilation unit and must therefore be encapsulated in a package. The task also lacks the declarative part and must therefore use the encapsulating unit for its declaration purposes. Finally, some severe problems, such as task termination dependencies to non local nodes, will arise.

The partitioning on packages is also widely accepted [1], [10], [16]. Several arguments may be put in favor for this method. First, Ada packages are library units. Also, the package is the main unit of logical program decomposition. However, constraints are often put on the declarations in package interfaces. When distributing on library packages, only minor changes are required to allow for distribution on library subprograms as well [17].

The method of partitioning on any part of an Ada program is developed in the APPL project [18]. The aim is to supply an application independent system that supports the execution in a distributed environment. The functional mapping of the application is described in a separate language, APPL (Ada Program Partitioning Language). The development of a system may start on a uniprocessor and may later, in the final integration phase, be transferred to the distributed target. The APPL approach gives the application no knowledge of the distribution. Hence, all fault tolerance has to be implemented within the underlying run-time system.

Another method for partitioning is to add new mechanisms for partitioning and distribution to the Ada language. Several authors note the absence of abstraction of a virtual node in the language. The required compilation unit should combine the declarative ability of a package and the parallel and stand-alone ability of a task. A suggestion have been made which involves the combination of a main procedure and a package into the compilation unit *partition* [11].

Fault tolerance.

One of the main purposes of distributed systems, beside increased performance, is fault tolerance. Fault tolerance involves error detection, error signaling, and error recovery by means of controlled system degradation and redistribution. The error recovery may be transparent [4], [5] or non-transparent [14], [6]. In transparent recovery, the run-time system handles the recovery and reconfiguration. The application is not aware of an error state in the system. In non-transparent recovery, the error is signaled to the application and it may take any appropriate steps to degrade the service. A full description of the actions required after a processor failure is given by Knight *et al* [14].

Ada does not fully support error detection and error signaling [6], neither does the LRM [15] define the state of a distributed program after the loss of a portion of the computing environment. The exception mechanism does not allow error states to be transferred between parallel activities asynchronously since exceptions may only be transferred during a rendezvous.

The loss of hardware will result in the loss or malfunction of software components, such as variables, subprograms or tasks. However, the only predefined exception for the detection of lost software resources is TASKING_ERROR, which is raised by the run time system in the caller of an abnormal task. Exceptions for the detection of other lost software resources can be added to the package SYSTEM. These exceptions should be raised by the run time system at the access of the lost resource, analogously with TASKING_ERROR.

Kamrad *et al* [5] state that too many software designs include unnecessary details of the hardware configuration in its reconfiguration strategy. From the software point of view it is of no interest that a processor is lost, but rather the loss of the software operations which that processor supported. Furthermore, introducing hardware details prevents the software from being reusable.

Various authors illustrate the statement above. Kamrad *et al* [5] specifies a mechanism, in a separate partitioning language, that makes it possible to raise a user defined asynchronous exception into a list of named task when a defined state is set. Arévalo *et al* [2] defines a death-notice mechanism; if a task wants to be informed of the death of another, it gives directions to the run-time system to get calls through an entry point at these events. These two examples show error signaling that does not involve hardware information. Another example is given by Knight *et al* [6] where two exceptions, NODE_FAIL and COMM_FAIL, are raised in every process that survives a processor failure or network failure respectively.

THE DISTRIBUTED ADA RUN-TIME SYSTEM, DARTS.

The Distributed Ada Run-Time System, DARTS, is developed by the Measurement and Data Acquisition group at the Department of Physics, Uppsala University, in collaboration with the Swedish Defence Research Establishment. The system is one example of solving some of the problems described in the introduction above. The intention is to implement the entire run-time system software in Ada, to test the behavior of Ada in real time applications, and to examine portability and reuse of software components.

Partitioning and distribution in DARTS.

DARTS can be used for pre-partitioning as well as post-partitioning. Using the pre-partitioning approach, the partitioning information is added to the application source code as pragmas. In case of a post-partitioning approach, the partitioning could be performed by some CASE tool generating the transformed Ada code.

DARTS aims to support the distribution of :

- tasks
- task objects
- packages
- variables
- subprograms

A software component that is selected for distribution is called a *distributed unit* (DU).

A program is partitioned into distributed units. A virtual node (VN) consists of a set of DUs that can execute on the node. The subset that actually executes is defined at run-time. Performance and fault tolerance implies that a DU may be a member of several virtual nodes, and subprograms may even execute on several VNs simultaneously. Virtual nodes are assigned to physical nodes (PN). Currently, DARTS can only map a single virtual node onto each physical node.

The state of a DU on a given VN depends on the preparations for execution. A *local* DU is currently executing on the node. The node must be prepared to receive calls to the DU from other nodes. A *remote and idle* DU is idle on the local node. All calls to the DU are forwarded to a remote node where the DU is currently executing. In case of a node failure, the state of the DU may be changed from *remote and idle* to *local*. The *remote* state implies that the DU is executing on a remote node. In case of a node failure the unit may be redistributed to another node, but not to this node. The states of the DUs on a node are set at startup using a configuration file.

The distribution is performed by source code transformation. This involves the insertion of additional code into the application source code. The inserted code interfaces the application to the distributed run-time system.

A number of global exceptions are declared to handle failure states. Recovery is transparent in the case of stateless units. Only when a lost DU is referenced, exceptions signal the permanent or temporary inaccessibility of the DU, as in the case of TASKING_ERROR in a non-distributed environment. This mechanism has been chosen since a process needs no information that a DU is lost if no communication or synchronization is required.

The DARTS is designed to put callers into a hibernating state during remote calls [19]. No busy-wait in communication routines, as described by Eisenhauer *et al* [20], is needed.

Syntax for partitioning.

Other work [17] in the distributed Ada field indicate that pragmas are a proper base for distribution and reconfiguration information. This is partly due to the fact that Ada does not prohibit the adding of pragmas. One drawback is that the partitioning information is spread over the source code.

Two pragmas are used for program partitioning; pragma Distribute and pragma Redistribute. Pragma Distribute is used to identify a distributed unit and associate the unit to its executing virtual node(s). Pragma Redistribute is used to enumerate the possible target nodes for redistribution. In some cases, such as calls to a subprogram DU executing on several nodes, the allocation of a task object DU, or during a redistribution, a choice of node has to be made. The choice is ruled by a distribution criterion for the DU. The distribution criterion is specified as one of the following:

- CURRENT_LOAD
- AVERAGE_LOAD
- MAILING_LOAD
- SPACE.

System overview.

The DARTS consists mainly of three parts; the communication layer, the distribution layer, and the application layer.

The communication layer is the lowest level of the distributed Ada run-time system. The layer handles the abstraction of the network and supports the upper layers with functionality for node event handling and low level byte transfers.

The distribution layer contains logic for handling the current configuration of the distribution, high level internode handshaking, node load monitoring, and redistribution. It supplies the application layer with primitives for message transfers to distributed units, and provides a means to determine if a given DU is executing locally or remotely.

The application layer consists of adapted application code, generated by a source code transformer. The transformation is made by adding passive server units (SU), i e alternate bodies, to the distributed units. The server unit forwards any call to the actual DU, which may execute on the same node or on a remote node. Server units are sometimes named 'local agents' [17], or 'client stubs' [21].

One SU exists on a node for each DU that executes on the node, may execute on the node, or is used on the node. Some modifications may be necessary in the application code to adapt calls to the SU. All such modifications are performed by the source code transformer.

The communication between SU and DU is made by passing command messages, constituting remote calls and rendezvous, as well as unit creation, abortion, and elaboration.

SOURCE CODE TRANSFORMATION.

Identification of Distributed Units.

A unique unit number is generated for each distributed unit. The number is given at compile time by the source code transformer for static units. For dynamically created units the number is generated at run-time. Information about each unit is held in a unit identification record (UI). Figure 1 shows the UI declaration.

It is not allowed to separate a DU from the scope of the used non-local entities, unless these are made distributed. To obtain the necessary visibility from the distribution layer, all nested SUs identifiers are given a unique extension and put in the scope of the distribution layer.

Transformation of Procedures and Functions.

The distribution of a subprogram is simply performed by replacing the body of the subprogram with a server body forwarding the calls to the DU on the executing node. The subprogram UI is included in the server unit body as a constant. The actual subprogram code is included in the SU, if the DU is selected for local execution. The implementation of the SU is described in Figure 2. It is not allowed to separate a subprogram DU from the scope of the used global variables, unless these variables are DUs.

Transformation of Distributed Variables.

The DARTS concept comprises two different paradigms for distributed variables. The first is based on a totally distributed ownership of the variable, the second defines a single owner with all others using that one instance. Both paradigms use the same support from the distribution layer.

In the first case, the pragma DISTRIBUTE is used to identify all owners of the variable. A local copy is maintained in the distribution layer on each node and each variable update will be transformed to an update of the local copy in conjunction with a broadcast to update all other instances in the network. A variable reference is transformed to a reference to the local copy of the variable.

In the second case, the pragma DISTRIBUTE is used to identify the owner of the variable. Any update or reference to the variable is transformed to a call to update or obtain the value held by the owner. A pragma REDISTRIBUTE indicates an alternate owner of the variable.

Transformation of Packages.

The transformation of a package involves the creation of a procedure to contain the package executable part. This initiation procedure is called when DARTS elaborates the package during system startup or reconfiguration. Also, all entities declared in the package specification is automatically regarded as distributed units. Figure 3 shows the transformation of a sample package.

Transformation of Tasks and Task types.

Tasks and task type objects are replaced by server units implemented as packages, with all entries declared as procedures using the entry identifiers as procedure names. Two additional parameters are added to each entry procedure. The first parameter is used to identify the called distributed task and the second parameter is used for sending the time value in timed entry calls. Additional subprograms are used for initiation and abortion, and for obtaining task attributes. All task objects derived from a task type are handled by the same SU package. Figure 4 shows a task declaration and the corresponding server unit package.

All tasks that are selected for distribution are transformed to task types, as suggested by Bishop *et al* [22]. The transformed code handles the creation of the task object. A *new* statement is transformed to a call to the NEW_UNIT function. Figure 5 and 6 show the transformation of a declaration of a static task object and a declaration of a dynamic task object with a *new* statement. Note that the activation of the task, in Figure 5, is delayed until the beginning of the parent block, while the activation of the task object, in Figure 6, is performed in the *new* statement. An *abort* statement is transformed to a call to the ABORT_UNIT procedure in the SU package. The task attributes T'CALLABLE and T'TERMINATED are obtained by calls to corresponding functions.

The parent-child synchronization is made possible by adding an additional entry, AWAIT_-TERMINATION, to the task type declaration. This entry is accepted at the completion of the task executable part. Figure 7 shows the transformation of a task type into a DU.

The entry calls in the application source code are transformed to fit the SU package. Figure 8 shows the transformation of a basic entry call. Note that the task object is provided as a parameter, and that the package name is used for clarity only, since the use clause makes the SU directly visible.

A timed entry call is rewritten into a block containing a call to the server package and an exception handler containing the time-out executable code. Figure 9 shows the transformed call. The conditional entry call is transformed into a timed entry call with a delay time of 0.0 seconds, in accordance to the functionality specified in the LRM [15]. The basic entry call is implemented as a timed entry call with infinite time.

When the call is made, the time-out parameter is used in a timed entry call on the remote node. If a time-out occurs, a time-out error message is returned to the calling node, resulting in the raising of a TIME_OUT_ERROR exception. This exception will be caught in the exception handler shown in Figure 9 and 10.

The termination mechanism in Ada is based on the block structure [15]. If a block, task, or subprogram has dependent tasks, it terminates when it has completed and all dependent tasks have terminated or are ready to terminate. An algorithm for termination in a multiprocessor environment is described by Flynn *et al* [23].

However, any efficient implementation of task termination requires access to the run-time system. The aim of the DARTS project was to keep the system portable and, hence, DARTS only supports immediate termination after completion. Synchronization is performed as remote or local rendezvous, between the parent and, in sequence, each child. The transformation of the terminate alternative is not addressed.

KEY MECHANISMS.

Remote calls.

The DARTS implements a remote call mechanism that avoids busy waits [19]. Unlike Volz *et al* [17], who uses a distribution package with a pool of call agents tasks for each distributed unit, DARTS uses only one pool of general and reusable call agent tasks and a single distribution package. This minimizes the storage needed for task agents in the distributed run-time system. To decouple the application from the lower layers, a pool of agent tasks is used on the calling node. This facilitates an orderly recover of a communication failure.

The distribution package consists mainly of the FORWARD_TO procedure. This procedure, called by the call agents on an executing node, interprets the DU identification number, unpacks the parameters and performs a call to the identified DU. As the call is completed, the return parameters are packed and sent to the calling node.

Exception handling.

The exception handling in DARTS may be divided into two parts;

- System Exceptions,
- User Exceptions.

The system exceptions are TIME_OUT_ERROR, used for the distributed timed rendezvous, DU_LOST_ERROR, used for signaling the loss of a distributed unit, and DU_INACCESSIBLE_ERROR, used for signaling the temporary inaccessibility of a DU during redistribution. For user exceptions, a simplified exception handling is used. All user exceptions are mapped into a single USER_ERROR exception. The same method is used by Atkinson *et al* in the DIADEM project [1].

Initiation and redistribution.

Distributed units are initiated in a uniform manner at system startup and during reconfiguration. A unit is idle until initiated. The initiation is handled by the distribution layer.

The initiation is performed by sending a command message to the DU to be initiated. At the remote node a call agent calls the initiate-entry, or -procedure, of the DU.

The redistribution logic is contained in the distribution layer and is implemented as a task. The application continues its execution during redistribution, although all references to DUs currently redistributed are signaled by the predefined DU_INACCESSIBLE_ERROR exception. If a DU with several simultaneously executing copies is redistributed, calls are simply redirected to the remaining DU copies.

The redistribution task is activated by the detection of a node failure. At redistribution, one of the the remaining nodes is selected to be master of the redistribution. The master node accepts *redistribution requests* from the other nodes, and, using a *redistribution acknowledgement*, signals the acceptance of the redistribution mastership. The master then evaluates all DUs that were executing on the failed node. For each such DU, the master selects a target node and sends an initiation message to the DU on the selected node. When all DUs have been processed, the master sends a *redistribution completed* message to all nodes in the network. This message releases the redistribution state in the system and puts all redistribution handlers to sleep.

Project status and performance tests.

DARTS was originally implemented on a VAX-cluster using the DEC Ada compiler. The communication layer was first based on mailboxes where the nodes were simulated as processes on a single machine. In a second version of the communication layer, Ethernet communication was used between workstations.

Currently DARTS is revised for increased performance and adapted to bare MC68030 boards (Force CPU37ZBE) with Ethernet communication. The compiler used is the TeleGen2 cross compiler, version 3.23 [24].

The performance data, on the MC68030 boards, available at this stage are not complete, nor fully analyzed. However, it has been found that a complete remote procedure call takes 12.2 ms, using an unoptimized version of DARTS. One observation is that rendezvous times are not very expensive using modern Ada compilers. A 'seize' operation on a semaphore implemented as a task takes less than 80 µs, and the Ethernet interrupt task handles an interrupt and buffers the incoming packet in another rendezvous in approximately 400 µs. However, packing and unpacking parameters, and transferring parameter blocks to and from Ethernet buffers, is time consuming.

CONCLUSIONS.

The Distributed Ada Run-Time System represent one approach to distribute Ada programs. We have found that some restrictions must be put on the language for use in distributed applications. High portability requirements on DARTS impose restrictions on the use of the underlying run-time system, preventing an efficient solution to the exception transferring problem and the distributed task termination problem.

We have also found that error recovery may, in the case of stateless DUs, be invisible to the application. The application can be informed of a failure at a reference to a lost program part. Hence, there is no need for an asynchronous exception mechanism to transfer failure states to the application.

Most Ada mechanisms are maintained in DARTS. This includes:

- remote and local, timed entry calls,
- remote and local subprogram calls,
- shared variables,
- exceptions at remote and local calls,
- dynamic creation and abortion of distributed tasks,
- limited task termination.

The ability to execute several instances of procedures, and the possibility to allocate tasks of the same task type on any number of nodes, make DARTS a vehicle to achieve high performance.

REFERENCES.

[1] C. Atkinson, T. Moreton and A. Natali,
Ada for Distributed Systems, Cambridge University Press, 1988.

[2] S. Arévalo and A. Alvarez,
Fault tolerant distributed Ada,
2nd International Workshop on Real Time Ada Issues, 1988.

[3] A.D. Hutcheon and A.J. Wellings,
Supporting Ada in a Distributed Environment,
2nd International Workshop on Real Time Ada Issues, 1988.

[4] D. Cornhill,
A Survivable Distributed Computer System For Embedded Applications Written In Ada, Ada Letters, Vol 3, Number 3, 1983.

[5] M. Kamrad, R. Jha and G. Eisenhauer,
Reducing the Complexity of Reconfigurable Systems in Ada,
2nd International Workshop on Real Time Ada Issues, 1988.

[6] J. Knight and M. Rouleau,
A New Approach to Fault Tolerance in Distributed Ada Programs,
2nd International Workshop on Real Time Ada Issues, 1988.

[7] M. Kamrad, R. Jha and D. Cornhill, *Distributed Ada.* ACM, 1987.

[8] D. Cornhill,
Four approaches to partitioning of Ada programs for execution of distributed targets, IEEE Computer Society Conference on Ada Applications and Environments, 1984.

[9] R. Fors, U. Olsson and G. Larsson,
The use of Ada in large shipborne weapon control system,
Ada in Industry, Cambridge University Press, 1988.

[10] R.M. Clapp and T. Mudge,
Ada on a Hypercube, Ada Letters, Vol 9, Number 2, 1989.

[11] A.B. Gargaro, S.J. Goldsack, R.A. Volz and A.J. Wellings,
Supporting Reliable Distributed Systems in Ada9X,
Proc. of the symposium Distributed Ada 1989.

[12] R. V. Scoy, J. Bamberger, and R. Firth,
An overview of DARK, Ada Letters, Vol 9, Number 7, 1989.

[13] J. Armitage and J. Chelini,
Ada Software on Distributed Targets: A Survey of Approaches,
Ada Letters, vol 4, Number 4.

[14] J. Knight and I. A. Urquhart,
On the implementation and use of Ada on fault-tolerant distributed systems, IEEE Transactions on Software Engineering, Vol SE-13, No 5, May 1987.

[15] *Reference Manual for the Ada Programming Language,*
(ANSI/MIL-STD-1815A),Ada Joint Program Office,
Department of Defence, Washington, D.C. 20301, 1983.

[16] T. Mudge, *Units of Distribution for Distributed Ada.*ACM, 1987.

[17] R.A. Volz, P. Krishnan and R.J. Theriault,
Distributed Ada- A Case Study.
Proc. of the symposium Distributed Ada 1989.

[18] D. Cornhill,
Distributed Ada Project. Honeywell,1982.

[19] D. Wengelin, M. Carlsson-Göthe and L. Asplund,
A System Structure to Avoid Busy Wait, Ada Letters, Vol 10, Number 1, 1990.

[20] G. Eisenhauer, R. Jha and J.M. Kamrad II,
Targeting a Traditional Compiler to a Distributed Environment,
Ada Letters, Vol 9, Number 2, 1989.

[21] A.D. Hutcheon and A.J. Wellings,
The York Distributed Ada Project,
Proc. of the symposium Distributed Ada 1989.

[22] J.M. Bishop, S.R. Adams and D.J. Pritchard,
Distributing Concurrent Ada Programs by Source Translation,
Software-Practice and Experience, Vol 17, December 1987.

[23] S. Flynn, E. Schonberg and E. Schonberg,
The Efficient Termination of Ada Tasks in a Multiprocessor Environment,
Ada Letters, Vol 7, Number 7,1987.

[24] L. Asplund, M. Carlsson Göthe, D. Wengelin and G. Bray.
Real time compilers for the 68020.
Ada Letters, Vol 9, Number 7,1989.

```
-- Declare the distributed unit kinds.
type UNIT_KIND is (     TASK_TYPE_KIND,    DERIVED_TASK_OBJECT_KIND,
                        VARIABLE_KIND,     SUBPROGRAM_KIND,
                        PACKAGE_KIND);

-- The id of a distributed unit.
type UNIT ( KIND : UNIT_KIND := SUBPROGRAM_KIND) is
   record
     UNIT_NO : NATURAL;
     case KIND is
       when DERIVED_TASK_OBJECT_KIND =>
         DERIVED_FROM : NATURAL := 0;
         DERIVED_TASK_OBJECT_SUBUNIT : NATURAL := 0;
         THE_TASK_OBJECT : ACCESS_TYPE := ( others => 0 );
       when TASK_TYPE_KIND => TASK_SUBUNIT : NATURAL := 0;
       when  PACKAGE_KIND =>  PACKAGE_SUBUNIT : NATURAL := 0;
       when others =>         null;
     end case;
   end record;
```

Figure 1. The Unit Identification record, UI.

```
-- The original source code.
procedure INC(X: in out INTEGER) is
begin
  X:= X + 1;
end INC;
pragma DISTRIBUTE( INC, TO => MACHINE_2);
pragma REDISTRIBUTE( INC, TO => MACHINE_1);

-- The transformed SU on machine 1.
procedure INC(X: in out INTEGER) is
  use DISTRIBUTED_MAIL;
  MY_D_U: constant DISTRIBUTED_UNIT(SUBPROGRAM_KIND) :=
    (KIND => SUBPROGRAM_KIND, UNIT_NO => 1);

  -- Local DU.
  procedure INC_LOCAL(X: in out INTEGER) is
  begin
    X:= X + 1;
  end INC_LOCAL;

begin
  if UNIT_MODE(MY_D_U) = LOCAL then INC_LOCAL(X);
  else -- REMOTE or REMOTE_AND_IDLE
     declare
      OUT_C, IN_C: COMMAND_TYPE;
    begin
      OUT_C.KIND:= MESSAGE_REQUEST;
      INTEGER_HANDLER.PACK(X, OUT_C.PARAMETERS); -- Pack parameters.
      SEND(MY_D_U, OUT_C, IN_C); -- Send and block caller.
      INTEGER_HANDLER.UNPACK(X, IN_C.PARAMETERS); -- Unpack parameters.
    end;
  end if;
end INC;
```

Figure 2. An implementation of a procedure server unit. The generated code allows both local and remote operation depending on the unit state. Note the unit identification declared as a constant and used when performing a remote call.

```
-- The original source code.
package MY_PACKAGE is
 procedure SOME_PROCEDURE;
 procedure SOME_OTHER_PROCEDURE;
end MY_PACKAGE;

package body MY_PACKAGE is
 -- Package declarative and
 -- implementation part.
begin
 -- Package executable part.
end MY_PACKAGE;


-- The transformed source code.
package MY_PACKAGE is
 procedure SOME_PROCEDURE;
 procedure SOME_OTHER_PROCEDURE;
 -- Added subprograms.
 procedure INITIATE_UNIT;
end MY_PACKAGE;

package body MY_PACKAGE is
 -- Package declarative and implementation part.
 procedure INITIATE is
 begin
    -- Package original executable part.
 end INITIATE;
-- Empty executable part.
begin
 null;
end MY_PACKAGE;
```

Figure 3. *The transformation of a package into a DU.*

```
-- The original source code.
declare
 task THE_SERVER is
    pragma DISTRIBUTE( TO => MACHINE_1);
    entry FIRST_ENTRY;
    entry SECOND_ENTRY;
 end THE_SERVER;
 task body THE_SERVER is separate;
begin
 -- Executable code.
end;


-- The transformed source code.
-- Extracted to the uttermost application
-- level due to visibility reasons.
package THE_SERVER_001 is
 subtype THE_SERVER_T001 is UNIT;
 procedure FIRST_ENTRY(
    TASK_OBJECT : THE_SERVER_T001;
    TIMEOUT : DURATION := DURATION'LAST );
 procedure SECOND_ENTRY(
    TASK_OBJECT : THE_SERVER_T001;
    TIMEOUT : DURATION := DURATION'LAST );
 -- New unit.
 function NEW_UNIT(
    THE_BLOCK_ID : BLOCK_ID )
    return THE_SERVER_T001;
 -- Abort procedure.
 procedure ABORT_UNIT(
    TASK_OBJECT : in out THE_SERVER_T001);
 -- Task attributes.
 function UNIT_CALLABLE(
    TASK_OBJECT : THE_SERVER_T001)
    return BOOLEAN;
 function UNIT_TERMINATED(
    TASK_OBJECT : THE_SERVER_T001)
    return BOOLEAN;
 procedure AWAIT_TERMINATION(
    TASK_OBJECT : THE_SERVER_T001);
end THE_SERVER_001;

-- The transformed code.
declare
 THE_SERVER :THE_SERVER_001.THE_SERVER_T001;
begin
 THE_SERVER := THE_SERVER_001.NEW_UNIT;
 begin
    -- Executable code.
 end;
 -- Added to the end of the block declaring
 -- the task or the body of the package to
 -- synchronize task termination.
 THE_SERVER_001.AWAIT_TERMINATION( THE_SERVER);
end;
```

Figure 4. *The transformation of a task declaration into a SU declaration and the transformation of the declaring block.*

```
-- The original source code.
declare
 A_SERVER : THE_SERVER;
begin
 A_SERVER.FIRST_ENTRY;
end;

-- The transformed source code.
declare
 use THE_SERVER_002;
 A_SERVER : THE_SERVER;
begin
 A_SERVER := THE_SERVER_002.NEW_UNIT;
 FIRST_ENTRY( A_SERVER);
 THE_SERVER_002.AWAIT_TERMINATION( A_SERVER);
end;
```

Figure 5. *A declaration of a static task object.*

```
-- The original source code.
declare
 A_SERVER : THE_SERVER_POINTER
    := new THE_SERVER;
begin
 A_SERVER.FIRST_ENTRY;
end;

-- The transformed source code.
declare
 use THE_SERVER_002;
 A_SERVER : THE_SERVER_POINTER
    := new THE_SERVER'(THE_SERVER_002.NEW_UNIT);
begin
 FIRST_ENTRY( THE_SERVER.all);
end;
```

Figure 6. *A declaration of a dynamic task object.*

```
task type THE_SERVER is
 entry FIRST_ENTRY;
 entry SECOND_ENTRY;
end THE_SERVER;

task body THE_SERVER is
 -- Task body declarative part.
begin
 -- Task body executable part.
end THE_SERVER;


-- The transformed source code.
task type THE_SERVER is
 entry FIRST_ENTRY;
 entry SECOND_ENTRY;
 -- Entry added by the transformer.
 entry AWAIT_TERMINATION;
end THE_SERVER;

task body THE_SERVER is
begin
 declare
    -- Task body declarative part.
 begin
    -- Task body executable part.
 end;
 accept AWAIT_TERMINATION;
end THE_SERVER;
```

Figure 7. *The transformation of a task DU.*

```
-- The original source code.
THE_SERVER.FIRST_ENTRY;

-- The transformed source code.
THE_SERVER_002.FIRST_ENTRY( THE_SERVER);
```

Figure 8. *The transformation of a basic entry call.*

```
-- The original source code.
select
 THE_SERVER.FIRST_ENTRY;
or
 delay 10.0;
 -- Time-out executable part;
end  select;

-- The transformed source code.
begin
 THE_SERVER_002.FIRST_ENTRY( THE_SERVER, TIMEOUT => 10.0);
exception
 when DISTRIBUTED_EXCEPTIONS.TIME_OUT_ERROR =>
    -- Time-out executable part;
end;
```

Figure 9. *The transformation of a timed entry call.*

```
-- The original source code.
select
 THE_SERVER.FIRST_ENTRY;
else
 -- Time-out executable part;
end  select;

-- The transformed source code.
begin
 THE_SERVER_001.FIRST_ENTRY( THE_SERVER, TIMEOUT => 0.0);
exception
 when DISTRIBUTED_EXCEPTIONS.TIME_OUT_ERROR =>
    -- Time-out executable part;
end;
```

Figure 10. *The transformation of a conditional entry call.*

VII

Daniel Wengelin
Swedish Defence Research Institute
S-102 54 STOCKHOLM, Sweden

Mats Carlsson Göthe, Lars Asplund
Uppsala Unversity
S-751 21 UPPSALA, Sweden

Abstract Using a source code transformation approach to Ada in a distributed environment will give some implementation difficulties. This paper presents an all Ada, portable, solution to the problem of suspending a caller on one node during a call to a remote node. The solution is based on two sets of tasks on each node, making it possible for a caller to hang on an entry during the call. Algorithms are presented in pseudo-Ada.

1 Introduction

During recent years, much effort has been put into the area of Ada on distributed targets. Several papers [Cor84, AMN88, CWA89, BAP87] focus on the possibility to use standard compilers. This can be accomplished by means of a preprocessor, that translates one Ada program into a set of Ada programs. The transformation is controlled by some partitioning information.

One difficulty observed [EJK89] in the transformation is how to avoid the use of busy waiting during calls to a remote node.

2 A system structure to avoid busy wait

Consider the following example.

The hardware consists of a two node network. The program to be run is basically a task on one node calling a procedure on the other node. The task is to be suspended during the execution of the procedure.

The transformation of the source code will include the adding of code to handle the distribution and interface the network. We assume that there is a package DISTRIBUTED_MAIL dealing with the interface. The package will declare a task type, MESSAGE_HANDLER (M_H), and a resource pool to hold objects of the task type. The package will also declare a RECEIVE procedure and a SEND procedure. A generic package, declaring a RECEPTOR task type, will be used. A pool of RECEPTORs is also held on each node.

The MESSAGE_HANDLER algorithm is

```
loop
      - Get a call from application
      accept FORWARD (inparameters, addressee, mypointer)
      - Send message over the network
      SEND (inparameters, addressee)
      - Wait for a RECEPTOR to call on reply message to me
      accept REPLY (outparameters)
      - Accept final call from application
      accept READ_REPLY (outparameters)
end loop
```

The RECEPTOR is implemented as

```
loop
      - Queue on network port for (request) message
      RECEIVE (inparameters, addressee, frompointer)
      - Perform actual call
      FORWARD_TO (inparameters, addressee, outparameters)
      - Put reply message back on network
      SEND (outparameters,frompointer)
end loop
```

The RECEIVE procedure gets a message from the network. It is implemented to return control to the RECEPTOR only when a request from another node arrives. If the message received is a reply to some earlier call from a message handler on the local node, this message handler will be called. The following statements are found in the RECEIVE procedure.

```
GET_REQUEST : loop
      GET_FROM_NETWORK (message)
      if message.IS_A_REPLYthen
            message.FROM_HANDLER.REPLY(message.OUTPARAMETERS)
      else
            exit GET_REQUEST
      end if
end loop GET_REQUEST
```

The transformation of the original program will include substituting the procedure body on the calling node. The stub will perform the following algorithm.

```
GET_A_MESSAGE_HANDLER (apointer)
PACK_INPARAMETERS (..., inparameters)
apointer.FORWARD (inparameters, addressee, apointer)
apointer.READ_REPLY (outparameters)
UNPACK_OUTPARAMETERS (outparameters,...)
```

The RECEPTOR task will be used as follows. On each node, a package DISTRIBUTE is added during the transformation. The package specification is empty, but the body includes several vital components. First, a procedure TO, which will take a call, identify the addressee, unpack the inparameters, perform the call, and pack and return the outparameters. Second, an instantiation of the generic RECEPTOR package, using the TO procedure as the generic actual to the FORWARD_TO procedure. Finally, the package includes a resource pool for objects of the RECEPTOR task type.

The structure of a remote procedure call can be seen in fig. 1. The numbers denote the data flow sequence . Also in the figure are numbers indicating the order in which the M_H accepts rendezvous and the RECEPTOR makes its calls.

During a call, the following happens; the task will make an ordinary procedure call(1), executing the substituted procedure body. A M_H task will be obtained, and the actual parameters of the call will be transferred by a rendezvous(2). The calling task will then suspend itself, by means of a call to the READ_REPLY entry denoted "3". Meanwhile, the M_H passes the call onto the other node(3,4), where the call is caught by a waiting receptor(5,6). The receptor recognizes the call and performs the actual call through the FORWARD_TO procedure(7). At return(8), a reply message will be created and sent(9) back to the requesting node. There, another receptor will pick up the message through a call to RECEIVE. In RECEIVE, the message is recognized as a reply(11). Hence, the M_H pointer is extracted and the waiting M_H is called(12). This will cause the release of the M_H, the acceptance of the READ_REPLY entry(13), and hence, the release of the application.

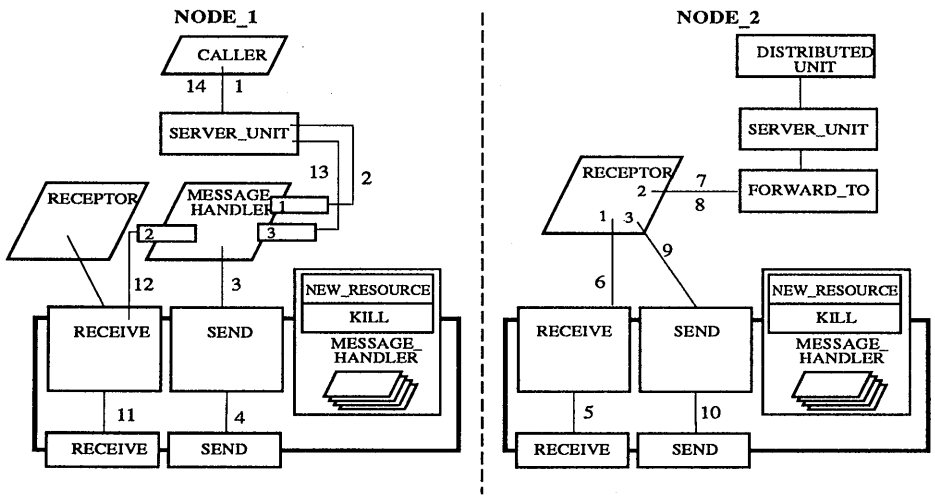


Figure 1. *Data and control structure in a remote call in DARTS*

3 Conclusions

There is a not very complex, all Ada, portable, way of having callers on one node suspended while their request is processed on a remote node. It has been tested and proven feasible, and is now undergoing further refinement and testing.

4 References

[Cor84] D Cornhill
Four approaches to partitioning of Ada programs for execution on distributed targets
IEEE Computer Society Conference on Ada Applications and Environments

[BAP87] J M Bishop, S R Adams, D J Pritchard
Distributing Concurrent Ada Programs by Source Code Translation
Software- Practice and Experience, Vol 17(12), 859-884 (Dec-87)

[AMN88] C Atkinsson, T Moreton, A Natali
Ada for Distributed Environments
Cambridge University Press, 1988

[CWA89] M Carlsson, D Wengelin, L Asplund
The Distributed Ada Run-Time System, DARTS
Uppsala University Institute of Physics Report, UUIP-1213
Uppsala University, Institute of Physics, P.O.Box 530
S-751 21 Uppsala, Sweden

[EJK89] G Eisenhauer, R Jha, J M Kamrad II
Targeting a Traditional Compiler to a Distributed Environment
Ada Letters, Vol IX(2), 45-51 (Mar/Apr-89)