Viper : a visualisation tool for parallel program construction

Viper

A Visualisation Tool for Parallel Program Construction

René Schiefer

Viper, a Visualisation Tool for Parallel Program Construction

PROEFONTWERP

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
Rector Magnificus, prof.dr. M. Rem, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op 14 december 1999 om 16.00 uur

door

René Schiefer

geboren te Rotterdam

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr. P.A.J. Hilbers
en
prof.dr. R.W. Dobinson

Copromotor:
dr. P.D.V. van der Stok

Stan Ackermans Instituut, Centrum voor Technologisch Ontwerpen.

The work in this thesis has been carried out under the auspices of CERN, the European research laboratory for high energy physics.

Print: Universiteitsdrukkerij Technische Universiteit Eindhoven

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN
Schiefer, René

Viper : a visualisation tool for parallel program construction /
by René Schiefer. - Eindhoven: Eindhoven University of Technology, 1999.
Proefontwerp. - ISBN 90-386-0711-3
NUGI 859
Subject headings: parallel computer systems / high energy physics / software systems
CR Subject Classification (1998) : D.1.3, C.1.2, D.2.6, D.2.5, J.2

Table of Contents

Preface

In 1993, the Institute for Continuing Education of the Eindhoven University of Technology (now called the Stan Ackermans Institute, SAI) started a stimulation program for designers theses ("promotie op proefontwerp"). Through this program, a number of final projects from the two-year post-masters programmes in technological design were funded to enable a two-year extension. It was then considered that the resulting time scale of two years and 9 months was in principle sufficient to produce results that would merit a PhD.

The project described in this thesis is one of these projects and started off as the first OOTI[1] graduation project abroad. CERN, the European research laboratory for high energy physics, situated between Geneva and the Franco-Swiss border, acted as employer in this project. The work has been performed at the laboratory over the period March '93 - January '96 and has been jointly funded by the SAI and CERN.

Although the role of the thesis in a designers PhD is still a topic of discussion, a number of requirements have been identified as described in [Bco94]. The main purpose of the thesis in this context is to provide a description of the project result ("product"), and of the design process that has led to this result, i.e. which design alternatives and choices have played a role in the development process. This is in contrast with the "traditional" thesis, where more emphasis is put on fundamental issues.

In this project a variety of topics have influenced the design process. On the one hand we have the complexities and dynamics of visualisation of parallel processing - an ongoing area of research to which this project contributes. On the other hand we have the inherent complexities that computing in high energy physics is faced with. Although we will touch upon many of these aspects, a thorough discussion of all of them is not possible within the scope of this thesis.

Furthermore, we have tried to find a balance between completeness and relevance in the description of the design process. This means that we address bottlenecks - i.e. important issues, but not every small detail. To allow for the above whilst not jeopardizing the homogeneity and readability of the thesis, we sometimes had to adopt a rather "loose" style of writing. Nevertheless, the following chapters provide a concise description of the Viper project, both in terms of project result and development process.

1. Ontwerpers Opleiding Technische Informatica, loosely translatable as Designers Programme for Software Technology.

Chapter 1

Outline of the thesis

The potential benefits of parallel computing in high energy physics (HEP) is a topic of ongoing research. Visualisation tools assist the software developer in managing and reducing the complexity of (parallel) program development. The subject of this thesis is the design of a visualisation tool for parallel program construction. One of its applications is the parallelisation of existing image reconstruction software in high energy physics.

Over the last decades, parallel computing has been a vigorous area of research, both generally and specifically aiming at high performance computing. As a result, a variety of machine architectures, programming languages, operating systems, tools and applications that support parallel computing are currently commercially available. However, a general recipe for the design of a parallel program based on a given specification does not (yet) exist. The Viper tool supports an improvement of the design process of parallel programs.

The fact that parallel computing is not a consolidated area is witnessed by a severe lack of standardisation in both hardware and software. Examples are the proprietary processor interconnections and the lack of standard software development tools. Standards such as MPI and HPF do exist, but are only just emerging. Ongoing research activities as in the ESPRIT[1] programme address, amongst other issues, the ease of exploitation of parallel computers.

In the high energy physics community, the potential power of parallel machines has generated great interest [Pea93]. The performance requirements on the software supporting HEP experiments is continuously increasing. For future CERN[2] experiments, e.g. to be carried out at the Large Hadron Collider (LHC), computing needs will increase by up to three orders of magnitude [Map93]. Parallel computing will be used to meet these demands. Amongst other developments and collaborations with industrial partners, research into parallel computing is carried out via ESPRIT projects. This includes the GP-MIMD P5404 project, the context in which this work has been performed.

The main goal of the GP-MIMD project has been to build a parallel machine on which to run parallelised applications [Gpm94]. One of the underlying aims is to investigate and demonstrate the potential benefits of parallel computing for *existing* HEP applications. The applications developed at CERN deal with data-acquisition, real-time decision making, simulation and image reconstruction. The enormous amount of HEP

1. European Strategic Programme for Information Technology
2. CERN is one of the major HEP research centres in the world

software that has been developed sofar represents an investment of hundreds of man-years. On the whole, these applications are written in Fortran77. This is a relatively old language and does not have the expressive power and structure to allow for a simple transition to parallel machines.

High Performance Fortran (HPF) [Koe94] is a successor of Fortran77 which does provide constructs to express parallelism, but the evolution of the language constructs has been at the cost of compatibility between Fortran77 and HPF. Therefore, the transformation of existing Fortran77 software to HPF for execution on a parallel machine does not present a viable option, given the amount of re-engineering required.

The term *parallel programming paradigm* is generally used to refer to a framework for designing parallel programs. Within GP-MIMD a programming paradigm called Mona Lisa [Schn93] has been developed, specifically designed to provide the functionality needed for constructing parallel HEP applications. This paradigm allows a large part of the Fortran77 software to be maintained in its present form while migrating sequential applications to parallel machines.

Developers of parallel software are faced with a complexity of program behaviour that goes beyond that of sequential programming. A parallel computation can be considered as a collection of co-operating (sequential) sub-computations that proceed in parallel and interact to produce the desired result. The higher the performance requirements, the higher the degree of parallelism that has to be introduced in the computation, and the more complex the interaction patterns become[1]. The interaction pattern need not be deterministic, i.e. subsequent executions of the same program may follow a different interaction pattern.

The interaction pattern is of interest to us for two reasons. Firstly, it is a potential source of programming errors, as the interaction pattern is often explicitly designed by the programmer[2]. Especially in the case of non-deterministic interaction, the programmer must take care that the interaction pattern eventually chosen during program execution does not produce undesirable program behaviour, such as producing the wrong result or no results at all. Secondly, the interaction pattern can reveal the limitations of the amount of parallelism that actually occurs in the execution. Sequential sub-computations interact because of data dependencies and a sequential sub-computation can be stalled temporarily when the requested data is not yet available. The more sub-computations are stalled, the less parallelism is effectively present, and the longer the overall computation takes to complete. Optimising the interaction pattern therefore plays an important role in the performance tuning of a parallel application.

Hence, the visualisation of interaction patterns is an important support tool in the construction of parallel programs. The tools that emerge as a result of research in this area help programmers to debug parallel programs and tune their performance. At the basis of these activities lies a program's event history: data recorded during the program's execution for debugging and/or performance evaluation purposes. The tool's task is to present this often large amount of data in graphical form to allow the user to deal with its complexity. To support the development of Mona Lisa programs, the Viper visualisation tool has been designed.

1. This is only true for non-regular interaction patterns. In the case of grid structures, the complexity of the interaction pattern remains constant as we increase the degree of parallelism.
2. There are also cases as described in Section 2.1, where the interaction pattern is generated by a tool.

Parallel computing has been introduced successfully in HEP, the most notable example being the so-called *event farming* technique (see Section 2.2). However, there is still a large amount of potential parallelism at a finer level (i.e. *within* the event analysis) that is currently not being exploited. This is because of the significant controversy that exists regarding the benefits and exploitability of this parallelism: many people believe the benefits do not weigh up against the programming effort. In an attempt to quantify some aspects of this qualitative discussion (how much benefit can we get for how much effort) we investigated a representative HEP application called CPREAD. CPREAD is the event image reconstruction package of the CPLEAR experiment at CERN.

CPREAD's potential for parallelisation in unexplored areas has been evaluated using Viper and Mona Lisa. We concluded that a strategy of minimal code change limits the amount of parallelism that can be introduced and gives only a small gain in reconstruction time. Higher gains would have to come progressively from significant code rewriting. The overall conclusion is therefore that future HEP applications provide a more profitable area for parallel computing where the parallel perspective can be taken into account at an early design stage.

The use of Mona Lisa in designing a parallel CPREAD version has been successful. The communication performance of the paradigm's current implementation however, proved to be disappointing when the parallel execution was analysed with Viper. As a result, Mona Lisa communications have been replaced by message passing operations that better exploit the machine architecture. This demonstrates an important issue: visualisation tools are severely restricted in usage if only a single paradigm is supported.

Various visualisation tools have already emerged over the years [TU94]. However, most of these tools have a restricted application domain. They either address a specific programming paradigm, are machine oriented as opposed to paradigm oriented, or operate correctly only under specific conditions. The Viper visualisation tool has been developed addressing the following issues: the consistent observation of a parallel computation; support of programming paradigms with a high level of abstraction; and paradigm independent design. The latter is illustrated by the tool's use in a completely different context from Mona Lisa: the investigation of traffic patterns in very large switching networks (>1000 nodes) within the Macramé ESPRIT project [OMI95].

The following chapters will go deeper into the design issues of Viper, its effectiveness in meeting its design goals, and other aspects of the work carried out. The thesis structure reflects Viper's evolution in the design process. Chapter two describes the environment in which the project has been carried out. Following a discussion on the contribution of visualisation tools to parallel program development, we conclude with the rationale for developing Viper. Viper's design is extensively described in chapters three, four and five. Chapter six is a validation chapter. First, we discuss the scope for parallelisation of existing image reconstruction applications in high energy physics, examining CPREAD. Then, we analyse the extent to which Viper and Mona Lisa have contributed in this and have met their design goals. Chapter seven is devoted to Viper's detachment from Mona Lisa, and its application in the Macramé project. Finally, chapter eight provides the conclusion of this thesis.

Chapter 2

Project background

This chapter gives some background information to describe the context in which the work has taken place. Firstly, some aspects of parallel computing are introduced which are relevant to this thesis. Secondly, an introduction to the computing requirements within high energy physics is given, showing where parallel computing is of interest. Thirdly, we introduce the Mona Lisa programming paradigm that was designed to support the introduction of such parallel computations. Finally, we describe how visualisation tools such as Viper support the development of parallel programs, and particularly Mona Lisa programs. The chapter concludes with the rationale for developing Viper, initial aims and external constraints as identified at the beginning of the project.

2.1 Some aspects of parallel computing

The essence of a parallel computer is the presence of multiple processors that can work together on one computation. The programmer strives for an efficient execution of this computation to achieve the shortest overall execution time. This includes the desire that each processor can proceed with its part of the computation without interruption as much as possible. In particular, the computations on the different processors should be as independent as possible, to reduce the amount of data exchange between processors to a minimum.

Data exchange between processors can be point-to-point (one source, one destination) or broadcast (one source, many destinations) based. Point-to-point data exchange can be initiated by either processor at any time, but can only be completed when this data is actually available (calculated). If data is requested at a time when the other computation cannot yet deliver, the computation that initiated the request is stalled. Furthermore, even if computations are fully synchronised and data is always available, communication delays can still cause a processor to stall.

On parallel machines where the communication has to be dealt with explicitly by the processor itself (as opposed to special purpose hardware), there is an additional disadvantage associated with inter-processor communication: any time a processor spends on communication is not spent on computation.

We will now look at a parallel machine's hardware architecture and how the issue of communication is addressed here. For a comprehensive overview of parallel computer architectures we refer to [Cul99]. In this thesis we focus on the main class of parallel computer architectures, the Multiple Instructions Multiple Data (MIMD) architecture. With this type of architecture each processor has its own set of instructions to carry out on its own data. The computer memory (that contains the instructions and data for each processor) can either be *shared* by all processors, or *distributed*, such that each proces-

sor has its own local memory. In the latter case, processors can only get access to non-local memory via an additional interconnection structure.

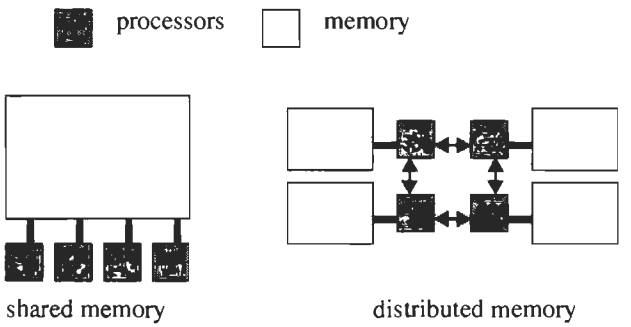

FIGURE 1. Two types of parallel computer architectures

The shared memory architecture allows each processor to access any part of the memory directly. As processors can use the memory to exchange data, the cost (delay) associated with inter-processor communication is comparable with that of memory access. This is not the case for the distributed memory, where non-local memory access has to make use of a separate interconnection structure. Here, the cost of inter-processor communication (non-local memory access) can significantly exceed that of local memory access. The performance penalty depends on latency and throughput characteristics of the interconnection structure, and whether this structure allows direct access to the non-local memory, or only indirectly via the owning processor.

The primary disadvantage of a shared memory architecture is its limitation to a small number of processors. The main scalability bottleneck with this type of architecture is the interconnection between memory and processor (for example a shared bus), a resource shared by all processors. As the number of processors in the system increases, contention between processors because of simultaneous memory access can lead to excessive stalling and hence performance degradation.

Large scale shared memory architectures also pose an interesting engineering problem. The physical size of a large number of processors prohibits a short communication path between processor and memory. Given the speed of an electrical signal (30 cm per 10 ns., a typical clock cycle time corresponding to a 100Mhz clock), this means that memory access times increase drastically, destroying the processing speed of the individual processors and thus killing the overall computing performance of the machine[1]. The distributed memory architecture does not have the above mentioned scalability problems. Current parallel machines of this type, such as the Intel Paragon system, have been built comprising up to 6,768 processors.

Distributed memory machines can be looked upon as more difficult to program because of the separate address space per processor, but current technological developments have already provided distributed memory machines that appear to the programmer as a (virtual) single address space, supported through dedicated hardware that implements a non-local memory access as a Direct Memory Access (DMA) data trans-

1. Although this argument may grossly overlook techniques such as caching and pipelining, it does illustrate the engineering problem.

fer. These machines are said to have a Non-Uniform Memory Access (NUMA) architecture because access times range from fast for a DMA transfer to very fast for a local cache access.

As far as programming parallel computers is concerned, two fundamental types of computational models can be identified: *algorithmic* parallelism places the different tasks within the parallel program on different processors; *data* parallelism involves the partitioning of data rather than code. An intuitive example of data parallelism in everyday life is the supermarket, where multiple cash registers are used to handle one dataset (the supermarket's clientele) so that the average queuing time of a customer remains acceptable. In car manufacturing, algorithmic parallelism is applied at the production line: multiple workers assemble cars in parallel to increase production throughput[1].

With data parallelism, the algorithm that is normally used to process the data sequentially, can sometimes simply be replicated over multiple processors, such that each instance works on a local data partition. One speaks of farming if there are no computational dependencies between instances, otherwise the term geometric parallelism is used [Hey89].

The farming model will re-appear in the next chapters, so we will explain this model in more detail. With farming, a master-slave configuration can be used for workload distribution and result collection, as illustrated in Figure 2. Farming is a popular model because of its simplicity and the mechanisms it provides for achieving good performance. Firstly, extra parallelism can be introduced by adding extra slaves, as long as the number of jobs is large enough. Secondly, as slaves can work completely independently from each other, the main issue in minimising the overall execution time is making sure all slaves get the same amount of work so that they finish around the same time (called load balancing). One load balancing technique is distributing jobs on request: a slave that gets large jobs will ask for a new job less frequently than a slave that gets small jobs.

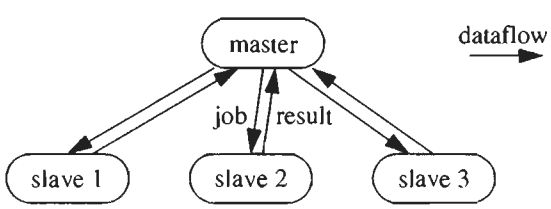

FIGURE 2. The farming computational model

Restructuring tools (compilers) can analyse a sequential program, detect potential parallelism and apply the appropriate transformations to obtain a parallel execution. Not only does this alleviate the programming effort for new applications, it also allows parallelisation of existing sequential applications, without any need for modification.

1. These useful analogies and others that turn parallelism from an abstract concept into an everyday life reality can be found in [Hil95].

Significant progress has been made in building parallelising compilers [ZC91]. However, fully exploiting parallelism using this approach remains difficult. Due to the amount of analysis involved, compilers often have to use approximating heuristics in their dependency analysis. Also, the way sequential code is written can introduce (unintentional) dependencies. Finally, the compiler sometimes lacks the information required to decide on optimal parallelisation (for instance because parameters are only known at run-time). This limits the speed-up obtainable via automatic parallelisation.

If a programmer is not satisfied with the performance increase of automatic parallelisation, he can decide to explicitly introduce parallelism in the program design. The programming language then has to provide constructs that express parallelism, and support the interaction (data exchange) between the sequential components, preferably in the form of high-level primitives. Traditional languages such as Fortran77 do not provide these.

The advent of parallel computers has introduced new languages, specifically designed for parallel machines. One example is Occam [Inm93], a programming language based on Hoare's CSP [Hoa86] and designed for the Transputer. Although a new parallel language may provide a coherent approach to programming parallel machines, there are some clear disadvantages if the new language does not conform well to existing languages. A more natural approach is to extend sequential programming languages with new constructs that express parallelism. We will discuss three examples.

High Performance Fortran (HPF) is an extension of Fortran 90 [Koe94]. Whilst designing a single sequential program, the programmer indicates with hints (incorporated in the program text) how the compiler can exploit parallelism. These hints define abstract processors and data distributions of data structures (for instance matrices) over these processors. This provides the programmer with an elegant means of defining computational models of the data parallelism type. It is up to the compiler to decide how abstract processors are mapped to physical processors, depending on the number of processors available. In the ultimate case all abstract processors could be mapped to one physical processor, resulting in a sequential application.

Parallel Virtual Machine (PVM) [Gei93/1] provides a set of message passing primitives for sequential languages such as C and Fortran. Using this approach, the programmer writes multiple sequential programs, that together make up the parallel program. The interaction between sequential programs occurs through the use of the *send* and *receive* primitives. This approach is evidently inspired by the distributed memory architecture where processors have to communicate with each other in order to gain access to non-local memory. An important limitation within PVM is the 'single process per processor' model. This means that the paradigm does not allow a mapping of multiple sequential programs onto the same physical computing node. PVM is only one instance of a whole class of message passing paradigms, and recent standardisation work has culminated in the Message Passing Interface standard (MPI).

The concept of Linda [CG90] is more closely related to the shared memory architecture. The Linda paradigm uses a common data pool called the *tuple space*. Sequential programs cannot interact directly but have to use the tuple space. The source places the data to be communicated in the tuple space using the *out* primitive. Once the data is in the tuple space, any correlation with the source is lost. The destination can extract the data from the tuple space by specifying a format or matching content in an *in* primitive. Linda's communication mechanism decouples the source and destination completely

and provides a high level of abstraction. However, the decoupling of source and destination is not always convenient: sometimes it is useful to know where the data is coming from. This has to be done by tagging the data explicitly with a source identifier.

In this section we have discussed a number of languages (and language extensions) with constructs to express parallelism. These constructs determine how the modules of a parallel program are defined and how they interact from the programmer's point of view. The model that a programmer uses within a given programming language to express parallelism is commonly referred to as a parallel programming paradigm. A useful classification of parallel programming paradigms is given in [Schn93].

To summarize this section, a programmer uses a parallel programming paradigm to apply a parallel computational model to the computation task at hand, so that high performance parallel machines can be used to obtain a solution in a reasonable timespan. One of the areas where high performance computing is frequently used, is high energy physics. The next section will provide a global introduction into the computing activities at CERN, and the current and future use of parallel computing at this institute in particular.

2.2 Computing requirements in High Energy Physics

High energy physics (HEP) studies the behaviour of *elementary particles*, the constituents of matter. This is done empirically using accelerators, in which particles are accelerated to increase their energy and subsequently collide with other particles. The collision produces a set of new particles that are scattered into various directions, possibly decaying into other particles. The trajectory of a particle is called a *track*. The set of particles and their tracks that result from one collision is called an *event*. In any experiment, the events generated that are of interest need to be recorded on a mass storage medium (e.g. tape) for analysis purposes.

Events are observed and registered with *detectors*. As a particle passes through detector material, it leaves traces that, with the help of electronics, can be translated into so-called *raw event* information: particle positions (hit patterns) that indicate a possible track, and indicators for the presence of a particle of a certain type (particle signature). However, a complete description of involved particles and their trajectories (the event image) is only obtained after considerable data treatment, a process called *event (image) reconstruction*.

Between raw event data production and physics analysis of events, a complicated process of *data reduction* takes place. Not all raw data represents interesting events, and the data quality may sometimes be too low to allow full event reconstruction. Such events are *rejected* (filtered out) as soon as possible, preferably before writing data to tape to reduce storage cost and unnecessary data management. It is important to realize though that the physicist wants to keep as much of the data as is reasonably possible.

Event data reduction is based on the application of rejection/acceptance criteria (referred to as a *trigger*), for example the presence of a particular particle. These criteria can often only be applied after some event reconstruction has taken place; e.g., to recognise a particle we need to know its mass, which is derived from speed and momentum, etc. Therefore, data reduction and event reconstruction go hand in hand.

To illustrate the computing requirements for future experiments at CERN, we will discuss the ATLAS[1] experiment in more detail [Atl94]. Within the ATLAS detector, one bunch crossing occurs every 25 ns. A bunch can be described as a cloud of accelerated particles, and when two bunches moving in opposite directions are crossed, particle interactions occur. Each bunch crossing produces 20 to 30 overlapping events, resulting in an event rate of 1GHz, or 10^9 events per second! This high event rate is required for two main reasons. If the experiment's goal is to produce a rare event, a high interaction rate is required to find the event within a reasonable timespan. Furthermore, to obtain statistically meaningful results on measuring small effects one needs large data samples, hence again a large number of events.

The raw event data size will be in the order of 1 MB. The ATLAS experiment is to be equipped with an I/O infrastructure that allows a data storage rate up to 100 MB/s, thus enabling the experiment to keep the 100 "best" events each second. Despite the huge data recording rate of 100 Mbyte per second, there is still a data reduction factor of 10^7 to be achieved, as 10^9 events are generated each second. The full event reconstruction that implements this data reduction, is estimated to take in the order of 1 s processing time for a single event, orders of magnitude more than the bunch crossing interval time of 25 ns. Both the amount of computing power and data transfer bandwidth required imply parallel and distributed processing of the event data in the process of real-time event filtering.

The ATLAS trigger system implements a so-called *multi-level trigger*. The current design is composed of three subsequent stages. Trigger level 1 is an integral part of the detector, implemented in dedicated hardware, and taking in data from over 10,000 channels. The implementation in dedicated hardware is necessary because of the timing demands: a decision on whether or not an "interesting" event has taken place has to be made within nanoseconds. The main task of the level 1 trigger can be summarised as twofold: 1) retain detector read-out data for further analysis (the detector itself is a memory-less device!) and 2) discard uninteresting events so that the remaining event rate is low enough to be input to trigger level 2, a distributed network of micro-processors.

The micro-processors of trigger level 2 run reprogrammable algorithms to perform additional event rate reduction. Whereas the design emphasis for level 1 is on speed, the design for level 2 is concerned with flexibility (reprogrammable components) and cheaper commodity components (as opposed to custom built hardware). The full input data rate of 100 GB/s for level 2 (corresponding to a 100 KHz event rate) would put too much strain on the data network that collects data from the output buffers of level 1 and distributes it to the level 2 processors. For this reason, the level 2 trigger reads in and examines only the data associated with so-called *regions of interest* (i.e. those parts of the detector that have been identified by level 1 as containing interesting information). The analysed data amounts to ~5% of the total data. Level 2 further reduces the event rate by a factor 100.

If the level 2 trigger decides not to reject an event, all event data (~1MB) is transferred to trigger level 3: a processor farm of commodity micro-processors, performing full event reconstruction at a rate of 1 KHz. This farm selects the best events (100 per second) and writes them to tape.

1. ATLAS is one of the experiments of the future Large Hadron Collider accelerator ring at CERN.

Apart from experimental analysis, a substantial amount of simulation work is carried out within HEP. Monte Carlo event simulation is used to verify the physics models of particles and cross check their predictions with the particle behaviour as observed in experiments. This type of simulation is also used to optimise detector design, and in combination with cross checking experimental results, allows detector performance monitoring.

Monte Carlo event simulation is essentially the inverse process of event reconstruction. The simulation input is based on a description of the event type. Using models of particle behaviour and the detector materials, as well as the detector's geometry description, the simulation produces a potential detector read out (raw event data). Although detector geometries differ from experiment to experiment, the models for particle and detector material behaviour are common to all experiments. This has stimulated the development of a standard core simulation package called GEANT [Gea95]. Using the GEANT program library, a simulation program for a particular experiment can be produced with reduced development effort.

Apart from the above mentioned event-oriented computing activities, there are also other activities that rely on high performance computing. We mention here accelerator simulation and theoretical HEP computing such as quantum chromodynamics (QCD). However, event data analysis and Monte Carlo simulation can be considered to be the main high performance computing activities. An illustration of this is the millions of lines of reconstruction code that have been written, representing an investment of hundreds of man years. In the area of event-oriented computing there is interest in parallel computing to reduce the time to solution for two distinct reasons:

- increase event throughput.

  Both Monte Carlo event simulation and event reconstruction involve huge amounts of events. By processing individual, independent events in parallel, event throughput is increased and total time to solution reduced.

  Much of CERN's parallelisation efforts so far have gone into the class of event reconstruction software, in order to improve the event throughput of these packages. Event farming has proven to be an effective technique to boost the performance of event reconstruction applications [Mic92]. It scales well in terms of throughput, on distributed memory machines as well as workstation networks, as the ratio between communication and computation is low; the reconstruction of one event takes in the order of seconds, whereas the communication overhead for one event is in the order of ms or even μs.

  Also parallelisation of GEANT has been investigated, both on the event level and track level [MS95].

- reduce single event analysis time (event latency).

  LHC experiments will all use parallelism in the area of on-line data reduction. For the level 2 trigger in the ATLAS experiment, the parallel analysis of detector regions of interest will be essential in obtaining the required data reduction within the time frame available: around 1 GB of data has to be analysed within a maximum of ~10 ms) [Moo95].

  Parallel event reconstruction implementations with reduced execution time and memory requirements could also allow the introduction of (currently off-line) event analysis techniques into the real-time domain, e.g. closed loop feedback detector control systems based on reconstructed event data (such as a beam alignment system). Both average and maximum event latency are of concern here.

Finally, reduced single event analysis time is beneficial in the area of interactive analysis (where event simulation and reconstruction are used in combination).

When discussing the potential benefits of parallelism, several characteristics of high energy physics analysis need to be taken into account:

- Detectors produce data that is output via I/O channels that are geographically distributed. The traditional grouping of data on a per event basis (event building) can be complex if not impossible due to bandwidth and latency requirements. Doing distributed event analysis first (hence allowing distributed input) so that the data rates are drastically reduced might prove to be a more natural and cost effective approach. The level 2 trigger design for the ATLAS experiment is an example of this.
- Triggers often use multiple rejection criteria for data reduction. The order in which criteria are applied can have a crucial effect on a trigger's performance. The optimal order is a function of event data characteristics, which may vary throughout an experiment's lifetime. A parallel analysis that uses competing algorithms (each algorithm implementing a different rejection criterion) may give a more robust trigger.
- Finally, the output side of the event analysis can also benefit from parallelism. The physicist's desire to write to tape as many events as possible, can sometimes only be satisfied by parallel I/O.

2.3 The Mona Lisa parallel programming paradigm

One of the tasks in the GP-MIMD project was the investigation to what extent the event latency and throughput characteristics of existing (off-line) event reconstruction software packages can be improved using parallel computing. To put it simply, how could existing applications benefit from parallelism? HEP applications are generally characterised as:

- data intensive (low ratio of computation to size of manipulated data)
- mostly written in Fortran77.
- using a memory management package for dynamic data structuring as Fortran77's standard data structures are limited (only fixed size, static arrays). For example, the Zebra package [Zeb95] provides variable sized array structures called *banks*, that can be organised in tree-like structures. The dynamic data structuring is important due to the varying event data size and characteristics.
- executing repetitive tasks, such as fitting each track to extract the physics parameters (e.g. momentum of the particle). Some of these tasks are implemented by more than one algorithm, as there is not always a "best" one (e.g. pattern recognition).

Looking at the structure of the manipulated data (many events, many tracks per event) and the repetitive nature of the data treatment, the potential for data parallelism appears to be high. But before investigating this any further, another question needed to be answered: what programming paradigm would be suitable for the task at hand? The following criteria were identified:

- The paradigm must be suitable for developing new HEP applications *and* parallelisation of existing sequential HEP applications. HPF for example does not score on this criterion, because it requires much code rewriting to adapt old Fortran code to the HPF standard.

- The paradigm must be machine independent, to reduce rewriting effort when migrating software to other machines. This is especially relevant if we take into consideration that the average HEP software lifetime is between 10 and 20 years! Paradigms that rely on the uniform memory access times of a shared memory architecture do not score on this criterion: the migration to (increasingly popular) distributed memory machines can have a negative impact on performance due to the larger overheads involved in non-local memory access.
- The paradigm must support parallel program structures that are likely to be useful for parallel HEP applications (data parallelism, farming in particular).
- The paradigm should have expressive language constructs, allowing the implementation of the relevant computational models (data parallelism!) with few constructs. As a result, the paradigm should be easy to use.

In principle, message passing is a very flexible paradigm that suffices most of the criteria. However, it was felt to be too close to the machine architecture of distributed memory. On a shared memory architecture, send/receive primitives can be an awkward way of looking at shared data access. Also, there exist computational models that are awkward to implement with message passing.

Another possible candidate is the Linda paradigm (see Section 2.1). However, when using Linda for distributed memory architectures the implementation of the paradigm tends to be relatively inefficient. This is due to the conflicting goals of efficient use of memory (resulting in a distribution of the tuple space over local memories), and efficient search algorithms for retrieving a requested tuple from the tuple space (as it can be in any local memory).

It was therefore decided to develop a new paradigm called Mona Lisa [Schn93] that would combine the best of both message passing and Linda[1]. Mona Lisa allows us to take a Fortran77 program, split it up in several sequential Fortran programmes (modules), add a communication structure and then run the collection of modules as a parallel program. The *module replication* concept makes it easy to implement farming strategies. A Mona Lisa module is declared as a replicated module by providing it with a replication variable and a replication range. A set of instances of the replicated module exists at run-time, with each instance uniquely identified by a replicator ID.

The program manager module is an integral part of the Mona Lisa run-time environment. To execute a Mona Lisa program, the program manager is downloaded to an appropriate computing node and started up. The program manager then downloads the other modules to their respective computing nodes and launches the parallel execution. As the modules execute the application code, they are monitored by the program manager. In the event of a deadlock situation, the program manager terminates the program cleanly. In chapter 3 we will go into more detail on this functionality. In particular, we will define which computational states are regarded by the program manager as deadlock states (see page 32). Appendix A illustrates the Mona Lisa concept with an example program.

In Linda, all modules share the same tuple space. In Mona Lisa, we create one space per module, called the global variable space. The variables in a module are split into

1. The development of Mona Lisa has been the PhD subject of André Schneider, GP-MIMD project member.

two categories: the *global* variables and the *local* variables. The global variables reside in the global variable space. A global variable is visible throughout the whole Mona Lisa program: modules, other than the declaring module, can access this variable. Local variables cannot be referenced by other modules.

In Linda, two modules synchronise on the exchange of data by using the tuple space as a data carrier: if the requesting module cannot find a tuple in the tuple space, it has to wait for the other module to insert a tuple in the tuple space. To implement this synchronisation in Mona Lisa, the concept of global variable *state* was introduced. A global variable can be in two states, *exposed* and *hidden*. If a global variable is hidden, its value cannot be read or changed by an external module. An outstanding write or read operation on a global variable by an external module is only completed when the variable becomes exposed. Global variables can be moved explicitly between the two states by the declaring module only. Furthermore, when a module wants to reference a global variable from another module, it has to explicitly reference that module.

So, the global variable space and the tuple space are used in different ways: the tuple space stores variable *values* which are *always* accessible. The tuple space can store multiple values that come from the same variable. The global variable space stores *variables*, which are *sometimes* accessible. Also, global accesses are always directed at a particular module (as opposed to Linda). One should note that these differences between Linda and Mona Lisa may have resulted in a paradigm that does not have the same power of expression as Linda, but a further investigation of this is not in the context of this thesis. The rationale for the Mona Lisa approach is that it addresses some of the problems that are associated with the efficient implementation of Linda on a distributed memory platform:

- All global variables have a fixed size which is known at compile time. Linda programmes on the other hand can cause excessive tuple generation on one processor, resulting in an "overflow" of the tuple space on that processor's local memory. The resulting data management issues that need to be addressed at run-time have a performance impact.
- The explicit reference of a module and global variable in every data access uniquely defines the location where the data are stored, as opposed to Linda where the tuple space has to be searched at run-time. These search algorithms have a worst case behaviour that involves more communication then required for Mona Lisa access primitives.

Similar to the *in* and *rd* primitives in Linda, Mona Lisa provides the *rdglb* and *inglb* primitives to read global variables: *rdglb* is appropriate for data broadcasting (replication), whereas *inglb* is appropriate in the case of data distribution (e.g. jobs in a processor farm). The *inrglb* primitive is provided to deal with replicated modules: the *inrglb* reads a global variable from one non-deterministically chosen instance. This is useful in a farming situation where the master has to collect data, produced by a set of slaves. As the master does not know which slave has finished first, it wants to read from any slave. Also in the situation of competing algorithms, where two modules each calculate the same result but using different techniques, the *inrglb* can be used to select the result from whichever module is fastest. Finally, the *wrglb* primitive is provided for cases where *writing* to a global variable is more appropriate than *reading* it.

Table 1 shows the Mona Lisa primitives and their operational semantics. The *glob* parameter represents the global variable which is being manipulated. The *mod* parame-

ter stands for the name of the module owning the global variable. The *loc* parameter represents the local variable in the module executing the primitive. The execution of a Mona Lisa primitive by a module is referred to as a *primitive call.*

TABLE 1. Overview of the Mona Lisa primitives

primitive	description
exposeglb(*glob*)	Set state of hidden variable *glob* to exposed.
hideglb(*glob*)	Set state of *glob* to hidden.
mod.rdglb(*glob*,*loc*)	Read value of exposed variable *glob* from module *mod* into *loc*.
mod.inglb(*glob*,*loc*)	Read value of exposed variable *glob* from module *mod* into *loc*. changing *glob*'s state to hidden (*mod* is a non-replicated module).
mod.inrglb(*glob*,*loc*)	Read value of exposed variable *glob* and change its state to hidden, *mod* is a replicated module. The instance which supplies the value is chosen non-deterministically.
mod.wrglb(*glob*,*loc*)	Write the value of *loc* to exposed variable *glob* (a variable from module *mod*), changing *glob*'s state to hidden.

In comparison with point-to-point message passing, the program modules are less coupled in Mona Lisa: the concept of first sending and then receiving data does not exist, as data are just made available without an explicit destination. This is especially useful in the case of conditional data flows: with message passing, data has to be sent to all modules that possibly need the data. Mona Lisa allows data transfer to occur only when a module explicitly requires it to continue its computation. This can allow a more transparent program structure and module re-usability.

On the other hand, Mona Lisa is by definition a heavier programming model on distributed memory platforms than message passing, as each (external) global variable read access requires at least two communications: one to request the data, and one to receive it. This compares to a single communication with message passing. This becomes particularly relevant in situations where inter-processor communications are expensive in terms of imposed latency and processing overhead.

The suitability, limitations and implementation aspects of the Mona Lisa paradigm will be further addressed in later chapters, in particular in relation to the Chorus distributed operating system on which the current implementation has been built. Suffice to say that, in summary, Mona Lisa can be regarded as a compromise between message passing and Linda in two respects, namely the hardware abstraction level and the associated cost in execution efficiency that needs to be paid to obtain this abstraction within the context of distributed memory platforms.

During this project only one type of hardware is supported. This parallel machine is based on the Transputer, a single-chip computer which has its own local memory. Four bi-directional communication links allow it to communicate with a neighbouring Transputer or other devices such as storage devices. The architecture of the (distributed memory) parallel machine consists of a network of interconnected Transputers, with one Transputer connected to a SUN workstation. This workstation is used to control the network, and also functions as main input/output facility via keyboard and screen.

The Mona Lisa development environment provides the tools that are necessary for compiling and running a Mona Lisa program on a parallel machine. The Mona Lisa pre-processor transforms Mona Lisa source code into a set of sequential Fortran pro-

grams, one for each module. It also performs type checking on the Mona Lisa variable declarations and primitive calls to signal programming errors of this nature.

In the generated sequential Fortran programs, the primitive calls have been replaced by library calls, such that they can be compiled with a conventional (sequential) compiler, linking in the Mona Lisa run-time libraries. The resulting binaries, together with the standard binary for the program manager module, constitute the parallel program binary.

During the parallel program execution, interaction with the user (if incorporated in the application's functionality) takes place via the SUN screen and keyboard. The system does not provide monitoring information on the application status. There is no feedback directly related to the interaction between the program modules, nor any related to the status of the processors (idle, busy). This makes it extremely difficult, if not impossible, to analyse why a program does not execute as fast as expected, or not at all! The Viper visualisation tool has been developed to address these issues. Visualisation tools help a programmer in performing:

- behavioural analysis (gain insight into parallel program behaviour). What interaction pattern occurs when the program executes?
- performance analysis (how much parallelism is present in the program execution, and how efficiently are the processors used).
- program debugging (finding and correcting programming errors).

In the next section we discuss in more detail these roles of visualisation tools in the process of parallel software development.

2.4 Visualisation tools

In this overview we present a summary of parallel program visualisation - details are more extensively discussed in [BH92][Com95][TU94]. We start with a definition of the terms for which a concise description is relevant in this section. For terms with more than one possible interpretation, we will adopt the most common definition.

A *program* is a computation specification, written down in a textual format. A *computation* is the execution of a program. A program makes use of *algorithms*. An algorithm can be seen as a recipe for the structure of the computation in order to achieve a predefined functionality. For instance, if the desired functionality is sorting of array elements, the programmer can choose from various sorting algorithms such as bubble sort, quick sort etc. to write the program.

A *parallel program* explicitly expresses concurrency, allowing its execution on a parallel computer to be performed as a *parallel computation*. A parallel computation is composed of multiple sequential computations (components) that proceed in parallel and interact to obtain a desired result. The operational characteristics of a parallel computation, such as the interaction between the sequential computations, are in this thesis referred to as *parallel program behaviour*.

The textual format in which computer programs are written does not lend itself to interpretation of the operational aspects of the algorithm that the program implements.

Research in this area has produced tools that use graphical animation to better express the program's behaviour. The use of sound has also been explored.

Although originally intended for sequential programs, graphical animation is even more useful for parallel program behaviour analysis. In particular the interaction between the sequential computations lends itself to visualisation. In the next section we will discuss how visualisation also contributes to the following issues:

- developing a correct parallel program
- developing a high performance parallel program (achieving high throughput and low latency, in a scalable way)

2.4.1 Developing a correct parallel program

Through formal specification, automatic code generation and other related methods and techniques, software engineers try to develop software that behaves correctly, i.e. according to specification. However, these methods have so far been successful in a restricted application domain. In addition, we still have to live with bugs in compilers, operating systems, hardware and the persisting possibility of human error in the seldom fully automated path from specification to running application.

Therefore, one of the almost unavoidable activities of software development involves tracking down programming errors and correcting them, described by the term *software debugging*. The classical approach in debugging a sequential program is by stepping through the program's execution and examining the program state at various points along the way. This repetitive process, which is assisted by a debugging tool, is commonly referred to as *cyclic debugging*.

The intuitive approach for debugging parallel programs would be to extend this to a process where we have one classical debugger per sequential computation. Some additional hierarchy of control needs to be added, such as the possibility to simultaneously start or stop all computations at once at a specific break point. An example of such a parallel debugger is TotalView, from BBN Systems and Technologies [Bbn95].

However, debugging of parallel programs differs from sequential debugging in a number of ways:

- In a parallel program execution, each sequential computation has to be considered in the context of the other computations. The program state changes that are observed when stepping through one sequential computation are a function of the interaction with the other sequential computations.
- A parallel program can exhibit non-deterministic behaviour. For example, if two sequential computations share a variable for communication purposes, then the communication result may depend dramatically on the order in which writes and reads take place.

  Tracking down the erroneous interaction pattern that results in undesirable program states (for example, one that results in a deadlock) is not the focus of debuggers such as TotalView; the focus there is still on the program state changes *inside* one sequential computation. In addition, the approach of cyclic debugging is not appropriate because non-deterministic bugs often tend to be non-reproducible.

It should be noted that non-deterministic behaviour is sometimes introduced on purpose. Consider for instance optimisation algorithms based on simulated annealing. The interaction pattern of a parallel implementation of such an algorithm can be so closely linked to the randomisation process, that it is intentionally non-deterministic.

Also, a farming application may exhibit non-deterministic behaviour in the sense that the job distribution over the workers can be such that it depends on the (beforehand unknown) job sizes, which in turn depend on the program input parameters.

- Any attempt to observe the behaviour of a distributed system may change the behaviour of that system. In particular, it is not uncommon for bugs that result from non-determinism to disappear when debugging instrumentation is added to the program.

  This phenomena is commonly referred to by the term *probe effect*. The probe effect shares characteristics with the Heisenberg uncertainty principle in physics, which explains the term Heisenbugs that is jokingly used by programmers for bugs that seem to appear or disappear at will.

To address the above issues, techniques additional to cyclic debugging have been developed:

- Static analysis tools can detect certain types of problems such as potential deadlock by inspection of the parallel program text. So, instead of analysing an erroneous execution of the program, static analysis techniques aim to prevent erroneous execution.
- The recording of relevant data of a parallel program execution for detailed analysis at a later stage partially decouples the debugging activity from the parallel program execution. This can be used to minimise the probe effect (disturbance of program execution). The data recorded are *events*, that are of interest to the tool user, usually because they establish the relationships between the sequential computations, such as the sending of a message, a memory access etc.

  The occurrence of an event is registered by a trace message, and trace messages are collected in a trace file. Trace messages contain a time stamp that relates to the local clock of the processor. In establishing a global time frame where we can compare the time stamps of trace messages from different processors, the alignment of the local clocks plays an important role.

  Trace message generation is usually implemented by adding extra statements to the parallel program (called *program instrumentation*) or it is built into the libraries that are used by the program (for example a message passing library).

  To minimise the probe effect, considerable effort has been spent on developing techniques that minimise the overhead, involved in generating the trace data. This includes the amount, size, generation and storage of trace messages.

An event history often involves large amounts of data. This, and the complexity of the data, prohibit a textual representation as an efficient means for analysis. Visualisation tools provide a graphical animation of the parallel execution that is based on an interpretation of the event history. The spatial and the temporal view of the interaction pattern are examples of well-known representations, provided in one form or another by many of these tools.

2.4.2 Developing a high performance parallel program

Making a correct implementation that satisfies a problem specification is essential in both sequential and parallel software development. Another important characteristic of a parallel program is a satisfactory execution time. Sometimes, the performance requirements are such an explicit part of the problem specification, that not achieving a certain performance represents a failure of the program. An illustrative example is weather forecasting, where parallel computing is used to obtain a forecast of reasonable precision within a reasonable time frame. Performance is an integral part of the specification: a weather forecasting system that takes one week to compute the expected temperatures for the next day is completely useless! Two major categories of tools for performance analysis and performance tuning currently exist:

- modelling tools such as P^3T [Com95] have predictive capabilities on program performance, based on a static analysis of the parallel program and a database that characterises performance factors as a set of parameters and constraints.

  A major advantage of these tools is the possibility of analysing what-if scenarios (trying out a different machine architecture or run-time environment). However, the model's accuracy and applicability define an upper limit on the quality of the results these tools give. This may reduce their application domain considerably.

- monitoring tools such as ParaGraph [HE91] interpret the detailed event history of a parallel program execution, and present this data to the user in a form that allows an investigation of performance aspects. Performance indicators relate to load balancing, data distribution, network traffic, message queues etc.

  Of course, the probe effect plays an important role here. A correct balance has to be found between the number of observations needed for sufficient detail/statistics and the level of disturbance of the program's execution.

In both cases, the information is presented preferably in a graphical form.

2.4.3 Differences between visualisation tools

Existing visualisation tools distinguish themselves on the following issues:

- The parallel platforms they support.
- The programming languages or programming paradigms they support.
- The way the instrumentation of the parallel program is carried out. Some tools rely on a third party (such as the user) to carry out the instrumentation, other tools can do this automatically.
- The support of multiple abstraction levels. Examples are a processor view on the machine architecture level, a task or process view on the programming level, etc.
- Requirements of the tool on the trace data if externally supplied.
- Support of real-time visualisation and/or post-mortem visualisation.

We will now look at one of the more prominent tools in detail, notably ParaGraph.

2.4.4 Description of ParaGraph

ParaGraph [HE91] has been developed at Oak Ridge National Laboratory and was first publicly available in 1990. The tool has undergone continued development efforts since, and here we will discuss the status of the tool as it was in March 1993 at the start of this project.

ParaGraph provides some 25 different graphical views, each showing a different interpretation of the event history that is recorded in the trace file. The easiest way to obtain this trace file is by making use of the Portable Instrumented Communication Library (PICL) [Wor92]. This library provides message passing operations on a variety of distributed memory platforms, and has the instrumentation necessary for trace message generation already built in. ParaGraph does not depend on PICL; any trace file that conforms to the same format can be analysed. ParaGraph requires the trace messages in the file to be sorted on a strictly increasing time stamp. This means the time stamping mechanism requires a clock resolution that is high enough, and in the event of a missing global system clock (as is often the case), a correct local clock alignment. ParaGraph has (conveniently) delegated the time stamping task to PICL.

ParaGraph preprocesses the trace file to determine some parameters like time scale and number of processors, before the graphical animation begins. After selecting the graphical views that need to be displayed, the user can start the animation. ParaGraph interprets the trace messages one by one, using the trace file as a script to be played out. Pause/resume/restart buttons in a control window give the user control over the way in which ParaGraph scans the trace file.

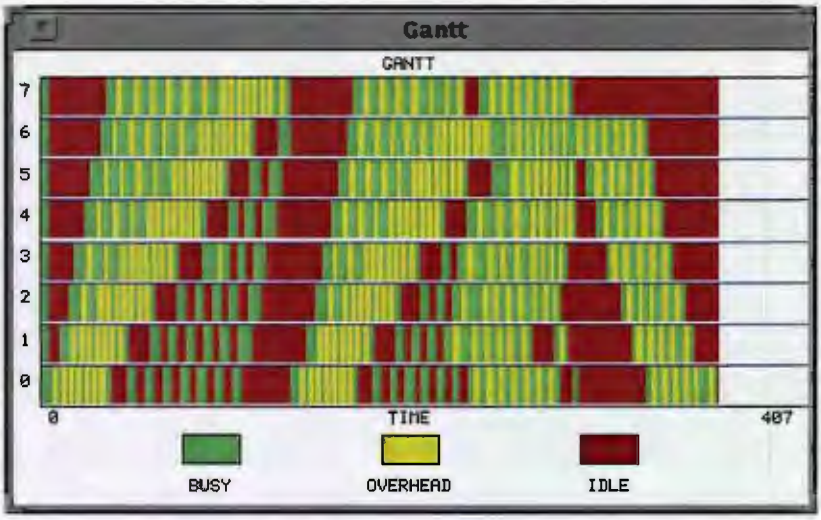

FIGURE 3. Example of a processor utilization view in ParaGraph

Most of the views fall into one of three categories: processor utilization, communication and task information. ParaGraph provides task views to relate performance indicators to parallel program behaviour. The user defines tasks within a program by using special PICL subroutine calls to mark begin and end of each task and assign it a task

number. The task activity over time can be displayed for example in gantt chart form, showing which tasks are executed by which processors at what time.

Figure 3 shows an example of a processor utilization view, characterising each processor as busy, communicating or idle over time. Other views present utilization information in an accumulated form, such as percentage of processors idle as a function of time. Utilization views allow an analysis of work load distribution over the processors.

Communication views focus on various performance indicators related to the interaction between processors: message queues, communication patterns etc. Figure 4 shows the hypercube view. The processor topology is depicted as a graph whose nodes represent processors and arcs represent communication lines. A node's status (busy, idle, sending and receiving) is indicated by its colour. To help the user determine if the network's physical connectivity is well exploited by the program's communication, message arcs corresponding to physical links are coloured differently from message arcs along virtual links[1]. Besides the hypercube topology, other layouts are supported as well, such as ring, mesh etc.

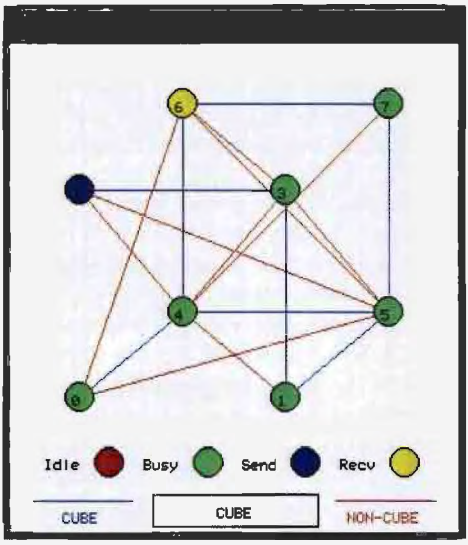

FIGURE 4. Example of a communication view in ParaGraph

A number of observations were made when we analysed ParaGraph in detail. Firstly, the tool is well-suited for a detailed performance analysis given the multitude of views. However, the views are relatively inflexible:

- They focus on providing feedback at the hardware level. For example, in the hypercube view it is clear when processor 0 communicates with processor 5, but depending on the complexity of the interaction pattern, it can require considerable effort

1. In ParaGraph terminology, a *virtual link* is shown in a ParaGraph view as a direct connection between two processors, despite the fact that this direct communication path physically does not exist: messages between the two processors always have to travel via other processors. For example, virtual links can appear in the hypercube view when the processors are in fact interconnected as a grid.

from the user to relate this information to the program's source code (which statement in the program text causes this communication). This makes the tool somewhat unsuitable for debugging purposes.

- They have a rigid spatial ordering of information. For instance, the ordering of processor numbers along an axis is either natural, fixed, or sometimes by Gray code[Scha96]. Another example is the hypercube view, where only fixed topologies are supported[1] and the representation of a processor is fixed, that is, non-scalable in size. One of the consequences is that the hypercube view quickly becomes unsuitable for topologies of more than 16 processors, a serious limitation!
- The views do not retain their information when the window is manipulated (moved, overlaid by another window etc.). This is a serious inconvenience. Performance analysis sometimes requires multiple views to be available simultaneously in order to be effective. Due to the physical limitations of the screen and the requirement of non-overlapping, the desire to have multiple views on screen is at the cost of detail in the graphs.

ParaGraph provides views at the *processor* level; explicit modelling of multitasking (multiple *processes* or *threads* etc.) is not supported. This is a serious limitation, considering the fact that it is not unusual to execute multiple sequential computations concurrently (in a time sliced fashion) on a single processor. In the same way, one could choose to run two Mona Lisa modules on the same processor. A Mona Lisa module is in fact itself implemented as a set of two processes running on the same processor (the reason for the existence of a second process will be explained in chapter 4).

ParaGraph gives the user feedback that is neither paradigm nor program text related. Instead, the information tends to be more about the use of processors and their interconnecting links on a distributed memory machine. This characteristic appears to be the result of a focus on performance analysis and the desire to general applicability. However, in the context of debugging, providing information that relates too much to the lower machine level defeats the purpose of an abstract programming paradigm. A comparison can be made with sequential programming: debuggers do not force a programmer to investigate memory dumps, but allow the inspection of variable values. Also with parallel programming it should be possible for the programmer to perform code writing and debugging activities at the same level of abstraction. As with programming languages, a programmer chooses a paradigm depending on the application domain. There is a need therefore for visualisation tools that are customisable at the paradigm level.

Also, ParaGraph's trace processing capabilities are rather restricted: only post-mortem analysis is supported (due to the required preprocessing of the trace file), and the tool imposes high requirements on trace message ordering and time stamps. As we will see in the next chapter, these requirements prohibit a direct interface between ParaGraph and the current Mona Lisa implementation.

Overall, we conclude that ParaGraph's qualities lie in the area of performance analysis at the hardware level, but the tool lacks the functionality and flexibility for program behaviour analysis at the paradigm level and debugging. We have designed the Viper visualisation tool to complement ParaGraph, as summarised in Table 2. The ratings on

1. Later versions of ParaGraph support user-definable topologies.

the desired functionality aspects vary between ++ (very positive), + (positive), 0 (neutral) and - (negative).

TABLE 2. Viper versus ParaGraph

Functionality aspect	ParaGraph	Viper
Support for real-time visualisation	-	++
Ability to provide views at paradigm level	-	++
Ability to provide views at machine level	+	-
Dealing with unreliable time stamps / trace message order	-	+
Adaptability of views	0	+
Multitude of views	++	-

We envisage a situation where we have two tools with complementary strengths: Viper provides information on the paradigm level (if desirable on a real-time basis), and has the functionality to construct a trace file (based on the trace data it receives) for ParaGraph that can be used off-line to do detailed performance analysis at the machine level. The development of a parallel program can then be a 3-stage iteration process where we write code, test it, improve its performance, and go back to coding. Streamlined support for such a process would rely on Viper generating the ParaGraph trace file, rather than having the parallel program generate two different trace files, one for Viper and one for ParaGraph.

In summary, the design of the Viper tool had to combine the good qualities of existing tools and address the needs mentioned earlier:

- The feedback must support the abstraction level of the paradigm (read: Mona Lisa), that is, directly relate to the modules and the primitive calls they use to interact. The programmer's code writing and debugging activities can then use the same level of abstraction.
- Visualisation of program execution is based on an event history that has to be generated by instrumentation of the Mona Lisa run-time library. The trace message generation should pursue a small trace message size etc. to keep the probe effect small.
- Viper must be adaptable to changes, including changes in the paradigm and its implementation. Both Viper and Mona Lisa are being developed in an area of research where no stable market nor product can be identified. The paradigm dependent part of Viper should be self-contained and as flexible as possible to allow Viper to be used for other paradigms as well.
- In relation to the previous point, maintainability is a priority. This has been translated into documentation requirements: not only a user manual, but also a technical maintenance manual needs to be provided to support future work on Viper (e.g. modifications and extensions).
- Viper must support both real-time visualisation (for monitoring purposes) and post-mortem visualisation (for detailed analysis).
- Viper must be able to construct consistent views with very few requirements on the order in which trace messages are produced and collected. In establishing this, we allow ourselves to assume error-free delivery of trace messages. This is to avoid an unnecessary complication of our task in terms of Viper's robustness.

- Viper has to provide an interface to ParaGraph, in the form of a ParaGraph trace file generation facility. Using a single Viper trace feed from the parallel program, this will enable exploitation of the strenghts of both tools in a co-operative way. It also allows us to benefit from the continuous improvements to ParaGraph as much as possible.

The list of Viper user requirements concludes chapter 2. In this chapter we have discussed the role of visualisation tools in the parallel program development process and how they support a programmer in writing correct and high performance parallel programs by increasing insight in program behaviour. We discussed the problem domain of Viper: parallel computing requirements in HEP, and the programming paradigm Mona Lisa. In the next chapter we will discuss the design strategy adopted for Viper, and we will draw up the first Viper specification.

Chapter 3

Design approach and first specification

In this chapter we start by describing how we designed Viper. Once the design approach is laid out, we continue with the construction of a first specification of Viper. The aim we have when we write the specification, is to adhere to the set of user requirements stated at the end of the previous chapter. This means for instance support for Mona Lisa program development, without a strong Mona Lisa dependency of the tool itself.

3.1 Design approach

Viper's overall design process can be described as an iterative one. Iterative, because we have not developed Viper in one go. Given the relative instability of Viper's working environment on which some of Viper's requirements are based (e.g. Mona Lisa and its underlying Chorus operating system, both research products under development), and the inexperience we have in building fit-for-purpose visualisation tools, the risk exists that any initial set of requirements may not be complete, accurate enough or stable over time.

In an iterative design process, the first product implementation does not necessarily have to meet all requirements. It is used as a first step in the right direction, and as a vehicle to get remaining requirements clearer. Through a process of extension and refinement, the ultimate project result after several design cycles has then converged to a product that meets the complete set of user requirements, without there being a necessity to have all these requirements in place at the start of the project.

Recognising uncertainty and taking future design adjustments into account from the very beginning allows an overall reduction in design effort, less rework and increased product quality when dealing with complex and dynamic environments. Recognition of this has led to the development of risk driven software development models, such as Boehm's spiral model [Boe88]. Viper has been developed in three cycles that map reasonably well to Boehm's 'spiral rounds'. In each cycle, Viper was significantly enhanced, though each time with a different design focus:

- the first cycle focused on developing a working tool with a satisfactory graphical user interface (GUI). The development of this GUI was deemed to be a major risk in terms of effort required.
- the second cycle focused on improving Viper's trace processing capabilities, e.g. consistent run reconstruction, trace message format and real-time processing.
- the third cycle focused on enhancing Viper's paradigm independence and demonstrating this by applying it in a different environment.

For the development of the GUI we used a *fast prototyping* approach. This meant we went through a small series of improved user interface versions, using members of the GP-MIMD project team as 'user evaluators' for each successive version. The decoupling of user interface and the rest of the application allowed us to iterate on improving the user interface whilst keeping overall development effort low. Fast prototyping is particularly supported by 4^{th} generation programming languages (4GL's). Unfortunately we did not find a suitable 4GL based graphics package and had to use a 3GL instead, so we had a little more coding effort, but the overall GUI prototyping effort was still restricted to about one to two man months development time.

Apart from the GUI, each iteration in Viper's overall development has concerned the translation of a relatively well-defined specification into a working product implementation. The management of this process has been done according to what is called "the waterfall model" in the software industry. The waterfall model owes its name to its linear structure: the project phases of specification, design, development (coding, testing) and installation follow each other in strict order. The phasing of the waterfall model is therefore relatively straightforward, and often the preferred option when the end result is well defined. In such situations, the waterfall model allows project management to focus on an efficient software development process rather than managing product risk. However, in a situation such as we were in, in March 1993, where the ultimate project result was not well definable within the given timespan, the different approach of prototyping and evolutionary development was preferable to reduce the risk of rework, budget overruns and missed deadlines.

To finalise this introductory section, we mention some general design principles that can be applied in almost any software engineering task:

- generalisation through abstraction: more robust design solutions appear from looking not only at the specific problem instance at hand, but also beyond.
- separation of concerns: keeping clear what issues need to be dealt with at what stage in the design process improves manageability.
- reduction of complexity by a "divide and conquer" strategy: for example, imposing a modular structure on the design such that each module provides a part of the overall functionality whilst aiming for little dependency between the modules.

Practical examples of these design rules shall be demonstrated when we discuss Viper's design in detail in the next sections.

3.2 Viper's first prototype: the specification

In the first phase in the development of the first prototype we have to distil a well-defined specification from the informal user requirements as they were set out at the end of the previous chapter. Some of these requirements can be classified as user requirements, others have been imposed by ourselves as design objectives. One of the main points mentioned was that Viper is to support the development of Mona Lisa programs through visualisation. The way we set out to do this follows below, although the reader should bear in mind that a number of aspects of the discussion are not Mona Lisa specific; they apply equally to other paradigms such as PVM or Linda.

Viper visualises the execution of a Mona Lisa program as a sequence of state changes, very much like a cyclic sequential debugger stepping through the states of a sequential

program. The state of a parallel program is a function of the states of the individual modules. For a Mona Lisa module we can distinguish the following mutually exclusive states:

- *executing*: the module is executing the application code.
- *interacting*: the module is executing a Mona Lisa primitive and is not blocked.
- *blocked*: the module is (temporarily) held up in the execution of a Mona Lisa primitive and there is also no progress in the communication subsystem:

  A module is blocked in an exposure when it wants to expose a variable but it cannot because that variable is already exposed; it has to wait for another module to access the variable, implicitly causing the variable's state to change to hidden.

  A module is blocked in an inglb or rdglb when the partner module has received the request for the global variable's value, but it is not yet in a position to send this value because the variable is not (yet) exposed.

  A module is blocked in an inrglb when all modules involved have received the request for the global variable's value, but none of them are yet in a position to send this value because the variable is not (yet) exposed.

  A module is blocked in a wrglb when the partner module has received the new value for its global variable, but it is not yet in a position to update the variable because it is not (yet) exposed.

  A module is never blocked in a hide.
- *inactive*: the module has not started yet, or has finished executing.

In the process of fine tuning of a Mona Lisa program the programmer may wish to minimise a module's time spent in some interacting and blocked periods, to achieve a reduction of the overall program execution time. In the interacting state, a module is delayed by the communication overhead that is caused by the paradigm. The blocked state represents idle time of a module that is not inflicted by communication delays but by a mere lack of synchronisation of the computation - e.g. one module wants something that the other one cannot deliver yet. The difference between these two situations is significant enough to model them as separate states.

The state information about a Mona Lisa module is further refined by including the states of its global variables, and the primitive it is executing (the latter is only applicable in the interacting and blocked states).

Until now we have focused on the state of a Mona Lisa module, and little has been said about the states of the processor on which a module runs. We will assume that each module runs on its own processor, or, in the case of multiple modules on one processor, at least the presence of some fair scheduling mechanism that avoids us having to deal with the danger of individual module starvation. In the current Mona Lisa implementation, modules can indeed share a processor, on a time slicing basis. A processor can be in one of three states:

- *communicating*: the processor is sending or receiving data.
- *executing*: the processor is executing the application code, or the processor is executing the code of the Mona Lisa run-time system. If the processor hardware has the capability of doing some communication in parallel with code execution, the processor is also considered to be executing at that stage.

- *idle*: the processor is neither communicating nor executing (this implies that the module(s) running on that processor is/are blocked).

The relationship between the module states and the processor states will be explored further in Section 3.2.2.

In addition to the above, the overall *parallel program state* can be characterised as:

- *inactive* when no module has started yet.
- *terminated* when all modules have finished executing.
- in *deadlock* when there is a cycle in the transitive closure of the blocking relationship between modules, where this relation is defined for Mona Lisa as: module A is blocked on module B if and only if module A is executing a Mona Lisa primitive that wants to access a hidden variable of module B. In a deadlock situation, the modules that belong to the cycle can make no further progress.
- *abnormally terminated* when a module is blocked on a Mona Lisa primitive that relates to another module that has finished executing, or when the only modules that have not finished executing are blocked on an expose primitive call.
- *running* in all other cases.

Now that we have done the pre-work, we can start to specify the first Viper prototype in terms of desired input and output. In the description of the desired output we will cheat a little, by showing actual pictures of the final tool to support the text. In reality these pictures were not to hand in this phase as Viper was still to be developed, and we drew up artificial pictures instead, to support the textual specifications.

3.2.1 Viper's output

A *view* is a graphical representation of (part of) the Mona Lisa program state information, and highlights a particular aspect of the parallel program state. Viper has to provide a number of such views. We will describe the views using an example Mona Lisa program called *Farm*. This program consists of five modules: one master and one slave module that has been replicated four times. The program execution can be described as a repetitive task that is performed by the master for a number of times, after which the program finishes. For each task, the master distributes four jobs over the four slaves by assigning a job to global variable JOB and exposing it. The slaves continuously try to get a job from the master with an inglb primitive. A slave that has completed its job makes the result available in its global variable RESULT and asks for a new job. After all four results are read by the master using the inrglb primitive, it finishes the task and starts the next one.

Animation view. This view gives a snapshot description of the parallel program state. Modules are represented by coloured circles. The colour of each circle is determined by the module's state. We choose to make use of intuitive traffic light colours: green for executing, yellow for interacting and red for blocked. The inactive state is represented by grey.

In the states interacting and blocked, a module can be in the process of reading or writing external global variables. In that case, an arrow is used to indicate the direction of data flow. Arrow colours indicate data request (red) or data transfer (black). The exposure of a global variable is indicated with an additional circle.

In the example shown in Figure 5 we are in the middle of the job distribution by the master. A job, stored in global variable JOB, is being communicated to Slave(3). This module is therefore *interacting*. Slave(1) and Slave(4) are *blocked* in their inglb primitive call, that is, waiting for a job to become available. Slave(2) was the first slave to receive a job and is now *executing* it. The master has started the exposure of the next job, but is now *blocked* because JOB is already exposed. Upon the completion of the inglb call of module Slave(3), the state of JOB will change to hidden, and the exposeglb call of the master can be completed.

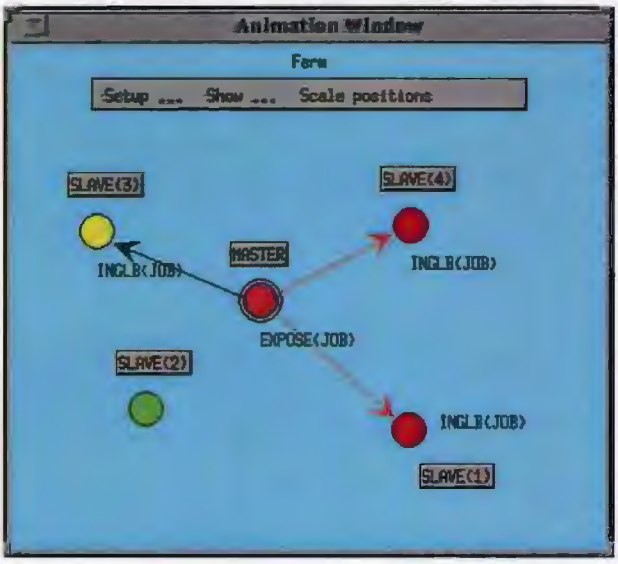

FIGURE 5. Example of a Mona Lisa animation view

Space Time view. In the space time view the state transitions of the individual modules are shown over time. Figure 6 shows a part of the execution of our master-slave program over time. Between T=500 and T=700 each slave reads a job from the master. The results are read by the master between T=1000 and T=2500. In the diagram we see that a primitive call can have both a blocked part and/or an interacting part, and may not terminate directly after the data has been received (see result collection by the master in the figure). In Section 3.2.2 we will discuss these issues in more detail.

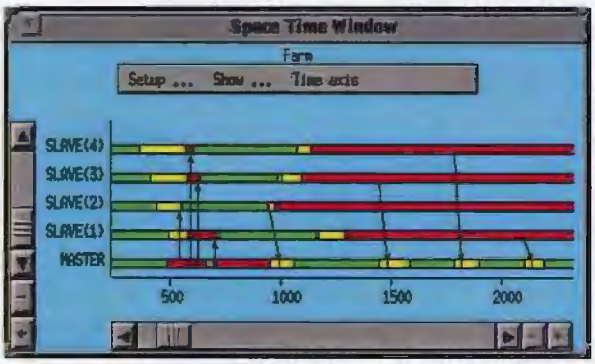

FIGURE 6. Example of a Mona Lisa space time view

Variable views. A variable view shows relevant information for Mona Lisa variables: name, type and state. There is one separate view for every module. Together with the animation view it allows the user to analyse, for example, deadlock situations.

FIGURE 7. Example of a Mona Lisa variable view

In our example, if the animation view would show that all slave modules have started executing an *inglb* primitive on JOB (requesting a new job), and that the master module is executing an inrglb primitive on RESULT (requesting a result), and the variable views show that the requested global variables JOB and RESULT are *hidden*, then we are faced with a deadlock situation; no further progress in the computation is possible.

Parallel program state view. This view shows the overall parallel program state as introduced at the beginning of this section. Particularly the two states *in deadlock* and *abnormally terminated* are relevant because they are generally not intended to occur. The Mona Lisa paradigm has the built-in functionality to detect these states and report them via trace messages. The trace messages are generated by each module separately, and collected by a central entity in the Mona Lisa program, the program manager, that runs as a separate process. The program manager analyses the trace messages it receives to detect deadlock situations or abnormal program termination.

FIGURE 8. Example of the program state view.

These states are represented in Viper by giving the square in the view an appropriate colour (e.g. yellow or red), accompanied by a textual description. The user can subsequently use the animation view and the variable views to analyse the program state in more detail, and possibly use the space-time view to explore the scenario that led to this state. Figure 8 shows the Farm program running, i.e. no abnormal program state has been detected (yet). In that case, the square is green.

From the description of the views it is clear that one of the dominant components of a visualisation tool such as Viper is the graphical user interface. The choice of an appropriate development tool for the graphical user interface is therefore an important design decision.

3.2.2 Viper's input

Viper's visualisation process is based on the interpretation of trace messages: whenever a state change occurs within the parallel program, it generates a trace message to register this state change; the trace message subsequently drives Viper's replay of the state change. To put it differently, a trace message corresponds to the occurrence of an event during the execution of the parallel program that is of interest to the tool user, usually because it relates to the parallelism being studied. Which events are of interest in the case of Mona Lisa? To answer that question, we will look at an example. Let us imagine we run our Mona Lisa program *Farm*, consisting of one Master and four Slave modules, on the Transputer machine, a distributed memory machine. We analyse the execution of the primitive call Master.inglb(Job,MyJob) by a module Slave(2), assuming both modules run on a separate processor.

Naively, the communication protocol would be a simple request-reply sequence of two messages: Slave(2) sends a request for the value of variable Job to the Master module and waits for an answer. When variable Job is exposed, the Master immediately sends back the value of Job; if not, it has to wait to send until variable Job becomes exposed. After sending the value, the Master module hides variable Job. When Slave(2) receives the message from module Master, it updates the value of its variable MyJob with the value of Job, and then continues execution.

In the protocol description above the Master module has to do two tasks simultaneously: execute the application code, and serve the request from the Slave module. For this reason each Mona Lisa module is in effect implemented as a set of two processes. The *application process* is executing the code of the parallel program, whereas the other process, called the *supervisor* (SPV), serves read/write requests of other modules. The supervisor and the application process share access to the global variable space.

The supervisor process guarantees the atomic execution of Mona Lisa primitives: reading a global variable and the transition to the hidden state of this variable (as happens with an inglb primitive) is carried out as one (atomic) operation.

It is important to realize that we consider the two processes per module to constitute an implementation issue from which we consciously want to shield the Viper user. This is why we have not raised this topic until now. However, the impact on the trace message structure is such that we cannot ignore it.

3.2.2.1 Module states versus process interaction

With the presence of supervisors, we will start with an analysis of the mapping between the module states as defined in Section 3.2 and the communication protocol between supervisor and application process. The mapping for the inglb primitive call from our example is depicted in Figure 9.

The left part of Figure 9 shows the communication protocol. Slave(2)'s application process sends its request for the value of variable Job to the Master's supervisor, and puts itself in a wait state. Upon the exposure of variable Job, the Master's supervisor sends its value to Slave(2)'s supervisor. This supervisor updates variable MyJob with the value of Job, and wakes up the application process.

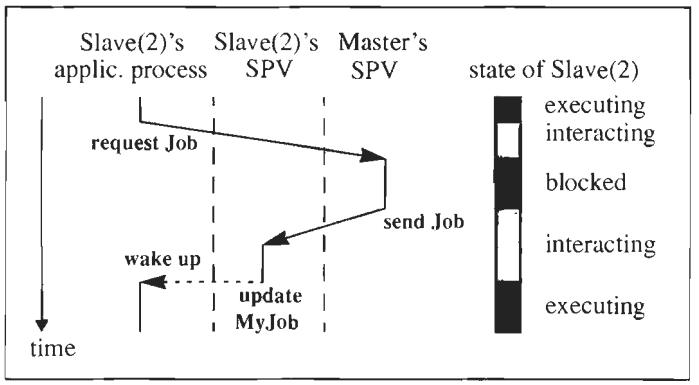

FIGURE 9. Analysis of the primitive call Master.inglb(Job,MyJob) by module Slave(2).

Module Slave(2) is defined to be *interacting* while messages are underway, or while its supervisor does internal housekeeping. In between the two periods of interaction there is a period where both the Master's supervisor and module Slave(2) are waiting for variable Job to become exposed. In this blocking state module Slave(2) is *blocked* and cannot make any progress, purely because the data it needs is not yet available. The reader can verify that this state assignment corresponds to our module state definition in Section 3.2.

The assignment of states as in Figure 9 allows us to attribute execution delays (performance loss) to distinct causes. Measuring the duration of interacting periods corresponds to measuring the overhead that is incurred by Mona Lisa and the underlying communication system. The blocking periods are more related to the way the computational workload is distributed, or the way a sequential algorithm is partitioned into modules.

The reader may notice that not all information is captured with this state definition. In particular:

- The delay / slowdown of the Master module's application process in our example is not captured in the state information of the Master module.
- The communication overhead at processor level is not captured explicitly, since all communications have been modelled as instant events whereas in reality they have a specific start and end.

Indeed, the state definition that has been adopted in this project may not be all-embracing, but is intended to focus on the identification of overall communication delays (marked by long interaction state durations) and algorithmic mismatch in data availability (marked by inappropriately long blocked state durations).

3.2.2.2 Events

The above analysis illustrates that, in the case of the inglb call, the interesting events (i.e. that mark a state change) are the start of a primitive call, the termination of a primitive call, and the sending or reception of a message in the Mona Lisa communication subsystem. The latter are called respectively *send events* and *receive events*. The events that do not use the communication subsystem are called *internal*. Note that, because

supervisor and application processes are assumed to be on the same processor and communicating via shared memory and semaphores rather than messaging, we model the wake up of the application process as a single, *internal* event. Send events and receive events explicitly refer to inter-module communication.

If we were to analyse the mapping between module states and communication protocol for all primitive types, and we attempted to partition events according to their state changing behaviour, we would find that there are 33 different categories of events in a Mona Lisa program execution (see table below). This includes event categories relating to module start/end, and program start/end. The other event categories each relate to a specific step of the communication protocol within a primitive type. Hence the event category implicitly indicates the state change involved. Each category is identified by a unique (*event_type, sub_type*) combination. The adopted nomenclature, in which *event_type* roughly corresponds to the Mona Lisa primitive involved and a further *sub_type* specification completes the event classification, is historically based and although intuitive somewhat arbitrary.

type of state change involved	**event_type**	**sub_type**
module start-up	S_RUNNING	INSTANT
module termination	S_DORMANT	INSTANT
exposeglb execution (with blocking period), start	M_EXPOSE	START
exposeglb execution, termination	M_EXPOSE	END
exposeglb execution (no blocking period), instant	M_EXPOSE	INSTANT
hideglb execution (with blocking period), start	M_HIDE	START
hideglb execution, termination	M_HIDE	END
hideglb execution (no blocking period), instant	M_HIDE	INSTANT
inglb execution, start	S_INRQ	START
inglb execution, data request received	S_INRQ	RECEIVED
inglb execution, data reply sent	S_INRP	SEND
inglb execution, data reply received	S_INRP	RECEIVED
inglb execution, termination	S_INRQ	END
rdglb execution	S_RDRQ	START
...	S_RDRQ	RECEIVED
	S_RDRP	SEND
	S_RDRP	RECEIVED
	S_RDRQ	END
inrglb execution	S_INRRQ	START
....	S_INRRQ	RECEIVED
	S_INRRP	SEND
	S_INRRP	RECEIVED
	S_ACK	SEND
	S_INRRQ	END
wrglb execution	S_WRRQ	START
....	S_WRRQ	RECEIVED
	S_WRRP	SEND
	S_WRRP	RECEIVED
	S_WRRQ	END
deadlock detected by program manager	S_DLOCK	ABORT
abnormal program termination	S_ABORT	ABORT
normal program termination	S_EXIT	ABORT

TABLE 3. Overview of Mona Lisa events

Each event that entails a state change has to be registered by means of a trace message. The time the event happened is recorded by time stamping the trace message. The time

stamp is obtained by reading the processor clock. The trace messages are generated by each module separately, and collected by the program manager. In addition, the program manager can generate trace messages related to the detection of deadlock situations or abnormal program termination (as discussed in Section 2.3).

3.2.2.3 Visualisation through event observation

The quality of Viper's visualisation depends on the quality of the trace message generation. Two issues are particularly important here:

- the accuracy of the time stamps. Incorrect time stamps have a direct effect on issues related to the space-time view. For example, if we want to investigate the load balancing in our example program Farm, we rely on the time stamps to correctly indicate the duration of slave jobs.
- the order in which trace messages are generated, which does not necessarily match the order in which they are fed into Viper. The animation view should preferably replay all state changes in the correct chronological order!

An example of Mona Lisa's implementation on the Transputer machine makes use of the Chorus distributed operating system [Cho94]. The current implementation based on Chorus v1.0 has the following characteristics:

- No preservation of message order is guaranteed between any two modules. As a result, the program manager collects trace messages in a potentially arbitrary order.
- No global timing exists; only a local clock is provided per processor. Clocks are synchronized within several ms. The clock frequency of a local clock fluctuates slightly around its mean value. The exact cause of this is unknown; it is probably largely due to scheduling by the Chorus kernel, causing unpredictable delays between the occurrence of an event and the time stamping of the corresponding trace message.

Given these characteristics, it is highly unlikely that the program manager's ordering of the trace messages corresponds to the chronological order of events. We cannot reconstruct the chronological order of events by reordering trace messages: uncorrelated events that occur in parallel in different modules, cannot be ordered correctly due to the inaccuracy of the time stamps. An example of uncorrelated events is a program with two modules, where the first event in both modules is the start of an exposeglb primitive call. The only way of knowing which exposeglb started first, is by comparing the time stamps of the corresponding trace messages.

Fortunately, the relative order of uncorrelated events is not so important as there is no fundamental difference for the computation. However, when a receive event is put before the corresponding send event we obtain an inconsistent observation of the computation: messages arrive at their destination before they are even generated! Therefore, correlated events *have* to be ordered correctly.

The correct ordering of correlated events (or the messages that record their occurrence) is called *consistent run construction*. A consistent run is an ordered sequence of Mona Lisa events, the ordering of which respects the causal dependencies between events. There is a direct causal dependency $\rightarrow$ between two events in the following two cases:

- events e1 and e2 are generated by the same module and e2 occurs directly after e1. By directly we mean: there is no event e3 such that e3 occurs after e1 and before e2. We write: $e_1 \rightarrow e_2$
- The receive event describing the reception of a Mona Lisa message is causally dependent on the event describing the sending of that message: $e_{\text{send m}} \rightarrow e_{\text{rec m}}$. We denote these two events by the term *send-receive event pair.*

The causal dependency relation is the transitive closure of the direct causal dependency. Note that not every two events need to be causally related, as we described in the example earlier on. The construction of a consistent run, which is an integral part of Viper's specification, has to be based on this causal dependency relation.

In literature, a consistent run construction technique is described that makes use of logical clocks [Mul93]. The technique assumes a logical clock in every module[1]. A logical clock "ticks" on each internal event and send event that occurs in the module. In the case of a receive event, the module inspects the logical clock value (of the sending module) that was sent with the message. It then increments its own clock with the number of ticks required to exceed this value. We will illustrate this with an example.

Figure 10 shows a part of the interaction between the Master and Slave(2) modules from our example program Farm. For the sake of discussion, we model a module as a single process, even though it is actually implemented as a pair of processes. Also, we do not include the interaction of the Master with the other Slaves. In the diagram, Slave(2) starts with the call Master.inglb(Job). This primitive call generates events e1 and e2 when sending the request, and e4 and e7 when receiving the data. The execution of Slave.inrglb(Result) by the Master generates events e5, e6, e9 and e10. The remaining internal events e3 and e8 correspond to the exposeglb calls.

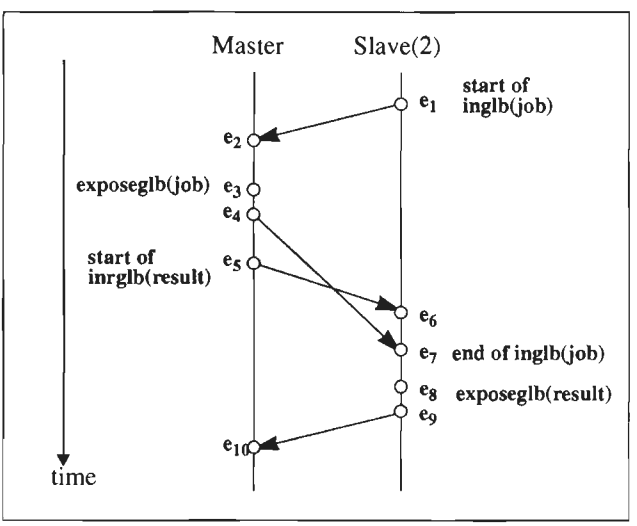

FIGURE 10. Sample of the event pattern in the Farm program

1. The logical clock of a module bears no relationship with the processor clock on which a module runs. For example, two modules, each with their own logical clock, can run on the same physical processor which has only one clock.

As the diagram shows, it is only after the application process of the Master sends off the inrglb request (e4) that the supervisor process of the Master completes processing the request sent by Slave(2) and returns the reply e5. These messages are communicated to Slave(2) through separate communication channels and it is not unlikely that they arrive in reverse order, as demonstrated by the arrows crossing in the diagram. Note that this can also occur when messages go through the same channel; we have made no assumption on FIFO behaviour of communication channels.

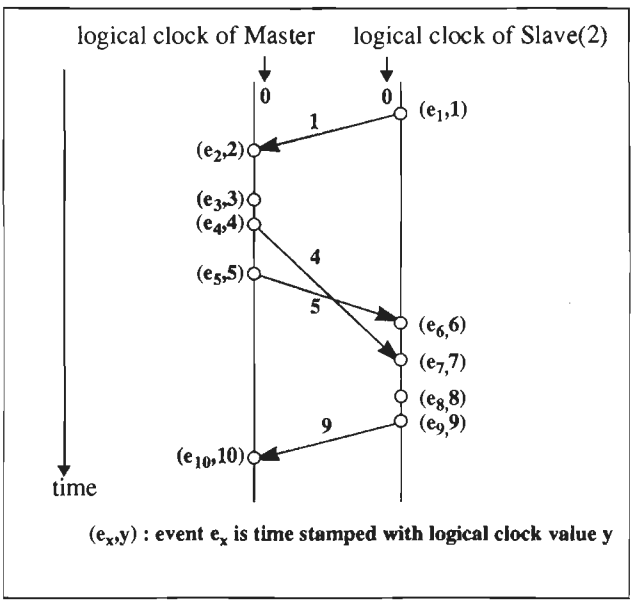

FIGURE 11. Logical clock time stamping of events for two interacting modules

In Figure 11 we have indicated the evolution of the logical clock values, starting at the initial value of 0. It also shows the time stamping of the events (e.g. event e_6 is time stamped with logical clock value 6), and the clock values that are sent across. In our example the events are numbered according to their occurrence in time. So the chronological order of events is e_1, e_2, ..., e_{10}. As we can see from the diagram, sorting the events on their logical clock time stamp results in the same order. Figure 11 illustrates two issues, associated with the use of logical clocks:

- Each communication arrow in the diagram is labelled with a logical clock value. Indeed, a logical clock value has to be sent with every message communicated between modules to update the clocks correctly. Hence the technique of logical clocks implies a structural additional overhead, resulting in a performance penalty.
- An observation of a computation such as Figure 11 is recreated from collected trace messages. In a situation were the communication pattern is a priori unknown to an observer that reconstructs this image as trace messages are received, this observer could not be sure whether indeed all trace messages had been collected. For instance, in Figure 12 there may be still some events missing between e5 and e10, generated by exposeglb calls. This uncertainty arises as the logical clock values within a module need not always be consecutive as illustrated in the example.

The latter issue can be addressed in various ways, based on the characteristics of the communication environment. In the situation where there is a realistic upper limit D on the delay in the sending of a trace message to the observing process, the following approach can be taken:

- The first step involves the implementation of a FIFO structure on all communication channels involved, if this is not the case already. This can easily be achieved by inserting a counter in each message at channel entry and re-ordering of messages at channel exit.
- The observer only accepts trace messages that have a time stamp T with T<=Now+D, where D is the delay upper limit.

The following algorithm illustrates how the observer would collect trace messages and construct a consistent observation (communication channels are assumed to be FIFO):

```
Process observer
VAR
        q : Queue; {message queue, sorted on time stamp T}
        r : Run; {message queue, sorted on logical clock value LC}
        m : message; p : process;
INIT
BEGIN
        WHILE True DO
                IF Now+D>T(q.head())
                        r.sorted_insert_from_right(q.head());
                        q.remove_head();
                IF receive(m,p)
                        IF Now+D>T(m)
                                r.sorted_insert_from_right(m);
                        ELSE
                                q.append(m);
                        FI
                FI
        OD
END
```

Summarising, if the communication delays are bounded to a certain maximum, we can accept incoming trace messages such that the program state observation lags the actual one with this maximum period (as implemented by the algorithm above). If this approach is not feasible as there is no guaranteed maximum delay in message delivery, another solution exists:

- Use the rule to adopt only messages with a logical clock value that is smaller or equal to the minimum of the observed logical clock values of the individual modules. This will guarantee a consistent observation, as demonstrated in [Mul93].

However, here one problem is substituted for another, namely progress: the rule requires all modules to steadily generate trace messages if the observed state is to progress in equal pace. In the case of a Mona Lisa Farm program, this may not always be the case; see Chapter 4 for an example Farm program where the master is only generating trace messages at distinct time intervals. Even with only one module temporarily inactive in terms of trace message generation, the run reconstruction process comes to a grinding halt.

Another approach to solving the problem uses an extension of the single logical clock, called the vector clock (also described in [Mul93]). This clock works according to the same principle, but uses N integer values for a program with N modules. This immediately shows its major drawback: not only do we still have the overhead penalty that is

associated with logical clocks, but the overhead also increases proportionally with the number of modules in the program.

We conclude that for the first prototype we will adopt the logical clock method assuming an upper limit message delay, although it may not be an optimal solution given the fact that at this stage we have no hard evidence of the existence of such an upperbound. The consequences for the Viper specification are twofold. For maximum flexibility and platform independence, the maximum message delay D will be implemented as a user-configurable parameter in Viper that can be adjusted at will. On the input side, we have to add a field to the trace message structure for the logical clock time stamp.

3.2.2.4 Module Setup file

In addition to trace messages, Viper makes use of a Module Setup file. This file contains a description of the Mona Lisa structure of the parallel program (number of modules, module names, global variable names, mappings of their names to numbers etc.). Viper uses this file to interpret the encoded contents of the trace messages. The programmer does not have to create the Module Setup file by hand: the file is generated by the Mona Lisa pre-processor when the parallel program is compiled.

The use of a Module Setup file follows from two requirements:

- Viper cannot pre-process a trace message file as ParaGraph does to determine the module identifiers etc., because that would prohibit real-time visualisation.
- Viper has to provide feedback at the paradigm level, so a certain amount of information (module names, variable names etc.), not present in the trace messages, needs to be supplied by a separate source.

At the start of the project, the Mona Lisa pre-processor did not provide a setup file, so an additional project task was defined to extend this tool's functionality. The example below shows the information stored in this file (derived from the *Farm* program):

```
MODULES:
5
GROUPS:
1
RANGE FOR GROUPS:
1 4
DEFINITION TABLE:
file name               module / group name        id
farm.MASTER             module MASTER              0
farm.SLAVE1             module SLAVE(1)            1
farm.SLAVE2             module SLAVE(2)            2
farm.SLAVE3             module SLAVE(3)            3
farm.SLAVE4             module SLAVE(4)            4
-                       # all_modules #            5
-                       replication group Slave    6

#ATTRIBUTES for module MASTER#
attribute name          id                         type
JOB                     0                          INTEGER(400)
#ATTRIBUTES for module MASTER#
attribute name          id                         type
RESULT                  0                          INTEGER(400)
```

FIGURE 12. Example of a Module Setup file.

Amongst other things, the file specifies for each module:

- The file name that contains the module executable. This information is not strictly necessary for Viper; the presence of this information is a by-product of the close integration of Viper with the Mona Lisa development environment.
- The name of the module (as declared in the source code).
- The numerical identifier assigned to the module.
- A list of attributes. In the case of Mona Lisa, attributes correspond to global variables. For each global variable the file lists its name, its numerical identifier and its type.

The formal definition of the Module Setup file structure can be found in appendix B. The LL-1 grammar rules on which the syntax is based are such that generation of the corresponding parser code is straightforward.

3.2.3 Operational specification

As mentioned before, Viper has to support several modes of trace message processing:

- Off-line, for post-mortem analysis. Trace messages that are generated by the parallel program are stored in a file, and interpreted by Viper at a later time, after the program execution has finished. The user can choose to have Viper process trace messages either continuously, or on a step-by-step basis allowing him/her to follow the state changes in detail.
- On-line, for monitoring purposes. During parallel program execution, trace messages are immediately communicated to Viper and the program state is updated.

When Viper processes trace messages continuously, the user controls the rate at which the screen is updated by specifying a time interval between successive updates. Specifying a zero length interval results in an update after every trace message. There are two reasons for wanting to control the screen update rate in the on-line mode:

1. When trace messages arrive at a high rate, a screen update after every trace message can slow down the tool so much that it cannot keep up with the rate at which trace messages are generated.
2. When successive screen updates follow each other too quickly, the user cannot follow the changes in a view. The resulting "screen flickering" is very unpleasant.

A status window allows the user to monitor Viper's trace message processing. It provides two kinds of information:

- progress information. Not every trace message input necessarily causes a view change. The decoupling of screen updates from the reading of trace messages makes it important to give the user explicit feedback on trace message input. The status view shows: when a trace message is read; when a trace message updates the parallel program state; and what the current time on the parallel program execution clock is. This time is associated with the animation view that represents the program state at a particular point in time during parallel program execution. In Figure 13 we see that a trace message generated by module MASTER with time stamp 2674 has been read, whereas the message with time stamp 2521 is the last one to update the parallel program state.

- special conditions: when conditions such as start of trace file, end of trace file, bad input (error in trace message format) occur, this is mentioned in the status window.

FIGURE 13. Example of the status window.

The operational control of Viper takes place via the control panel, depicted in Figure 14. With the first menu choice, *settings*, the user sets two parameters related to resource usage on the workstation: the screen update interval (affecting the amount of graphics processing required), and for what time interval all state information is saved. At each state update, existing state information that is time stamped older than the current execution clock time minus the history time interval, is discarded. This controls the memory consumption on the workstation. The next two menu choices control the creation and removal of the windows discussed earlier. The menu choice *status* invokes or removes the status window. With the menu choice *views* the user can select the views he/she wants to have displayed on the screen.
The rest of the menu is related to trace message processing. *Go/stop* (de)activates continuous processing, whereas *step* invokes one state change at a time. The menu option *breakpoint* allows a "fast forward" of the visualisation, stopping at the clock value that is specified as breakpoint. *Reset* brings Viper back to its initial state: the program state information is set back to default values ("start of program execution"), internal buffers are cleared, and in the case of off-line processing, the trace file input pointer is reset to the start of the file. Finally, the visualisation can be terminated by choosing *quit*.
Two parameters are specified at start-up: whether visualisation is on-line or off-line, and the name of the parallel program. Viper uses the latter to determine the name of the Module Setup file and the trace file to be read.

FIGURE 14. Viper's control panel.

With the operational specification details we conclude this chapter. Chapter 3 started with a section on the developing approach, explaining the iterative process adopted to deal with an environment of evolving requirements. Following on, we subsequently specified Viper's functionality in terms of input, output and operational behaviour. This resulted in a complete specification for the first Viper prototype. In the next chapter we will see how we can come up with an overall design (architecture) for Viper that meets the specification.

Chapter 4

Building the first Viper prototype

Viper's architecture and design will be described in a two-phased approach. First, we develop a system model that captures the essential requirements embedded in the specification, called the *essential model*. This rather abstract model describes which activities must be performed, and what data is needed to perform these activities. Such a model may be satisfied by multiple, different implementations. As we are taking these implementation decisions, we map this model to a second system model, which is less abstract and lends itself to straightforward coding without any further crucial design decisions.

4.1 The essential model

Numerous methods for systems modelling have been developed over recent decades, each with their set of concepts and notations. In most methods, multiple views on the same system are used to separate structural issues (what are the system's components) from dynamic issues (the system's behaviour) and functional issues (what does the system accomplish). StateMate [RA90] and OMT [Rum91] are examples of methods that use each of these views. This separation of concerns allows the modelling of systems with a high degree of complexity.

For our abstract model of Viper's architecture we adopt StateMate. The StateMate method has a significant user base in industry and academia (as demonstrated by case tool support, available in the market as commercial software). Another important reason for choosing StateMate is its comprehensive set of modelling tools, that allow us to work at the pure specification level. This means that implementation decisions can be postponed. For example, StateMate neither prescribes nor excludes an object-oriented approach to software implementation. However, for the description of design decisions taken at the implementation stage we will use OMT (see Section 4.5).

StateMate uses *activity charts* to describe functional capabilities (i.e. relating to data processing), *state charts* to describe dynamic, behavioural aspects (i.e. relating to control), and *module charts* to describe the physical structure of a system.

4.1.1 Functional view of the essential model

The activity chart for Viper is shown in Figure 15. The single activity at system level, which can be described as 'visualising a Mona Lisa parallel program state', has been decomposed into a number of component activities which are represented by ovals in the diagram. Where necessary, these activities will be split recursively into component activities until sufficient detail is captured, as described in [Pre94]. The activities

absorb and produce information flows, indicated by solid arrows. Labels indicate what type of information is involved. Information can be stored in a permanent data structure called a *store*, which is represented by a square.

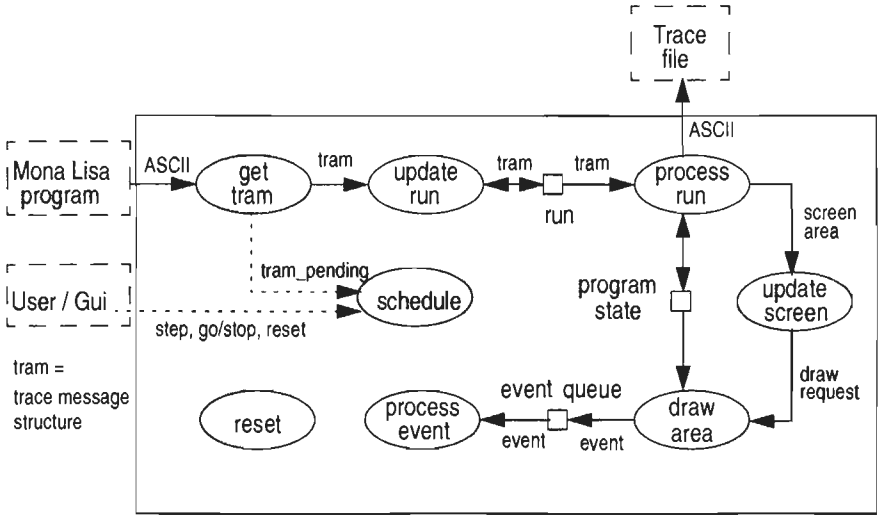

FIGURE 15. StateMate Activity chart for Viper

Description of activities and external actors:

- *Mona Lisa program:*
  This represents the parallel program under inspection.
- *get tram:*
  A string of ASCII characters (trace message), generated by the *Mona Lisa program*, is read into the system and formatted into a trace message structure (abbreviated as *tram*).
- *update run:*
  This activity performs the consistent run construction in *run*. The trace messages that are passed on to *update run* are buffered until they can be inserted into *run* such that consistency of the run is guaranteed, at that stage and in the future. The trace messages in *run* are sorted on their (adjusted) time stamp.
- *process run:*
  Of the trace messages stored in *run*, this activity takes the first one to update the state representation of the parallel program. Some parts of the Viper views need redrawing as a result of a state change. These screen areas are passed on to the activity *update screen*.
- *update screen:*
  This activity takes all the screen areas it has received so far, and issues redraw requests for them. Exactly one redraw request is issued for every part of the screen that needs to be updated, even though the same screen area may have been received multiple times since the last screen update. In this way, the redrawing effort can be reduced considerably by setting an appropriate screen update rate.

- *draw area:*
  This activity interprets the draw requests and generates appropriate graphics events (elementary drawing instructions). A request refers to an *abstract* screen area, and contains no geometrical screen representation data. For example, a request may entail something like 'redraw module X in the animation view'. The purpose of the activity *draw area* is therefore to provide a link between the otherwise strictly separated view states and actual screen representations.
- *process event:*
  A pending graphics event is delivered to and processed by the graphics event handler. Only by execution of this activity does an actual screen update take place.
- *schedule:*
  The controlling activity. It determines which activity is executed when. This is further specified by state diagrams in the following section.

Information flow characteristics (activity inputs/outputs) is one of the criteria that [Pre94] describes how to decompose activities into component activities. Other criteria include scheduling (which part of an activity is performed when) and functional complexity. A general aim is to minimise the complexity of the interfaces resulting from the decomposition process. These criteria have been applied and documented below.

The store *program state* on the right hand side of the diagram contains a representation of the Mona Lisa parallel program state that is constructed from the trace messages received. The upper part of the diagram involves all activities that contribute to the update of this state, whereas the lower part of the diagram shows the activities that are responsible for generating the graphical representation of what is stored in *program state*.

The introduction of the store *program state* into the architecture is a consequence of the design decision to separate program state representation and the graphical representation of that state. Alternatively, the application's structure could have been chosen such that each trace message is translated to a partial state update and immediately applied to the screen, without any record of the overall programme state. The store *program state* has been introduced for two reasons:

- The requirement that the screen update rate is user-controllable and therefore does not always correspond to the actual program state, is most elegantly met by introducing a separate program state representation; it allows the trace message processing to continue while the screen is frozen for example.
- The separation follows what is commonly referred to as the object-handler-viewer paradigm. In this paradigm, the object that needs a graphical representation automatically notifies a handler when it changes, and the handler subsequently triggers one or more graphical views on this object to redraw themselves. Most graphics packages support this paradigm nowadays, because it allows the different parts of an application to stay relatively independent so they can be modified, extended or replaced separately.

A similar decoupling has been introduced between the part of the application that constructs a consistent run of trace messages, and the part that processes these trace messages one by one to update *program state*. The activity *update run* buffers and reorders the trace messages it receives so that they are added to the store *run* in a consistent order. The activity *process run* independently takes these trace messages out of the

run one at a time to update the value of store *program state*. In addition, this activity can output ASCII formatted trace messages to a trace file when the corresponding trace message is processed. The written trace message need not be identical to the trace message being processed (i.e. it may represent a different type of event). This functionality allows for example the production of a ParaGraph trace file from a Mona Lisa trace file.

The activity *get tram* takes in strings of characters, generated by the external activity *Mona Lisa program* (external actors or activities are depicted by rectangles). These characters are formatted into an internal trace message structure for which we will use the abbreviation *tram*. The activity *get tram* has been modelled separately, to emphasise that it has a particular characteristic: the specification states that Viper must be able to read from a file, as well as directly communicate with the Mona Lisa program. Consequently, multiple implementations of *get tram* will be required to accommodate each of these situations.

The graphical representation of what is stored in *program state* is carried out by a set of activities that implement the object-handler-view protocol which we touched upon earlier. In this protocol, three parties are involved:

- the *object* that is being visualised (in this case the program state)
- the *views* on that object (for example, the animation part of a particular module)
- the *handler* that synchronises changes in the object's state with the object's views.

The *draw area* activity can be regarded as the view component: it is responsible for the graphics rendering. The *update screen* activity is essentially the handler's task and does the following: *process run* provides *update screen* with the screen areas that need to be updated as a result of the *program state* update. When it is time for a screen update, *update screen* merges all the screen areas it has received so far, and issues redraw requests to *draw area*. Exactly one redraw request is issued for every screen area that needs to be updated. Should two subsequent state changes have affected the same draw area since the last screen update (resulting in the same screen area being passed onto *update screen* twice), still only one redraw request is issued. With an appropriate screen update rate, this accomplishes very efficient graphics processing.

The implementation of *draw area* may or may not involve high-level primitives (such as 'draw text label'), but eventually this is translated down to operating system type of primitives such as 'draw pixel' or 'draw line'. In many graphical operating systems such as Xwindows, these primitives are modelled as 'graphics events' which are processed in the background by a separate O/S process. All other user interface related actions such as, keyboard input and mouse movements are in fact also modelled as events. Newly generated events are placed in an event queue. The O/S process responsible for processing them goes through an infinite 'event dispatching loop', taking events out of the queue one by one.

We have modelled the event processing explicitly in our activity chart, because we want to model the scheduling aspects involved in processing incoming trace messages and processing graphics events. Graphics processing is CPU intensive, and given its implementation as a separate process, potentially unpredictable in terms of CPU demand as a function of time. We would like to make sure however that processing of incoming trace messages gets priority, to reduce the probability that the trace message communication becomes a bottle-neck and starts to affect the parallel program execu-

tion more than necessary. It is important to realise though that we assume both *event queue* and *process event* to be integral parts of either the graphics package or the operating system environment.

Two activities remain to be described. *Reset*, although it updates the stores *run* and *program state*, has no relationship with other activities. It simply puts Viper back into its default state that is assumed at system start up time. Finally, the activity *schedule* controls all activities (what is executed when). The scheduling is based on the control information provided by the *user* (via the GUI) and *get tram*, indicated by the dashed arrows in the diagram. This is further described in the scheduler's state diagrams.

4.1.2 Dynamic view of the essential model

In the state diagrams, we use the following shorthand: instead of formal StateMate conventions, we link a state with the execution of an activity by simply using the same name for the state and the activity. For example, the state "updating run" is implicitly linked to the activity "update run". The link between a state and its definition is also established using the same name. This avoids unnecessary cluttering of the diagrams.

Figure 16 shows the main states for the scheduler. Initially the system is in the *idle* state (indicated with the curved arrow). A transition to the state *running* takes place when the *go/stop* event is generated; this event represents the user pressing the go/stop menu button, to start the visualisation. A transition from *idle* to *stepping* indicates the user wants to visualise the next state change in the parallel program. This is only possible if the run is not empty, or if there is a trace message pending. The return to state *idle* is either after a successful state update, indicated by the *tram_processed* event, or after reading in all pending trace messages without being successful in placing any trace message in the run due to run consistency constraints. A description of all relevant events and conditions is given below.

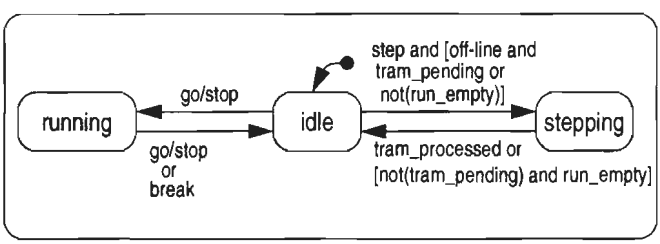

FIGURE 16. Top Level StateMate state diagram for Viper

Description of *events* and [conditions]:

- *step, go/stop, reset*: generated when the user selects the appropriate button in the menu.
- *break*: generated when a user specified breakpoint has been reached.
- [off-line], [on-line]: indicates the working mode of Viper.
- *display_timeout*: generated when Viper's real-time clock value minus time of last screen update exceeds period length of the screen update rate.
- [run_empty]: indicates that *run* is empty.

- [event_pending]: indicates that there is a graphics event available for the event handler in the event queue.
- *tram_pending*: indicates that the next trace message is available for Viper.
- *tram_processed*: indicates that the program state has been successfully updated.

Figures 17, 18 and 19 describe each of the main states in more detail.

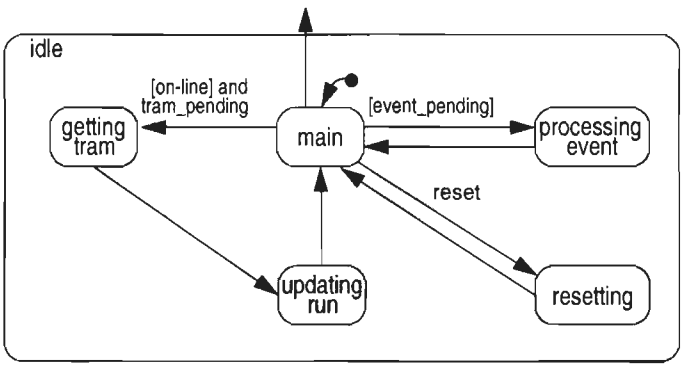

FIGURE 17. sub-states of state IDLE

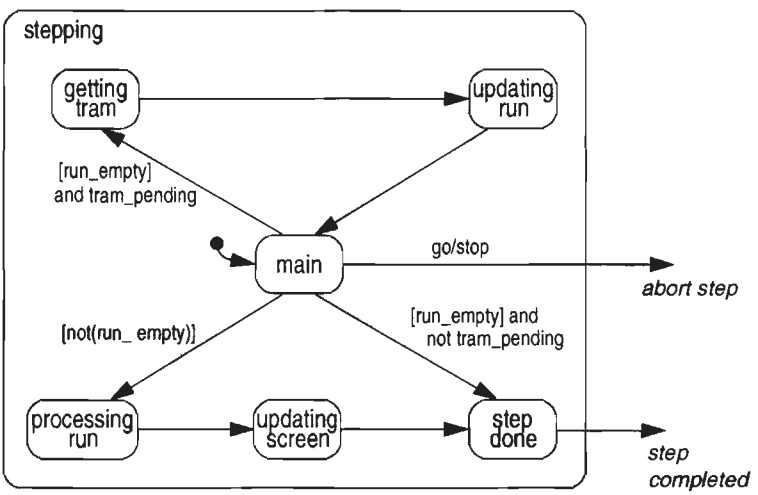

FIGURE 18. sub-states of state STEPPING

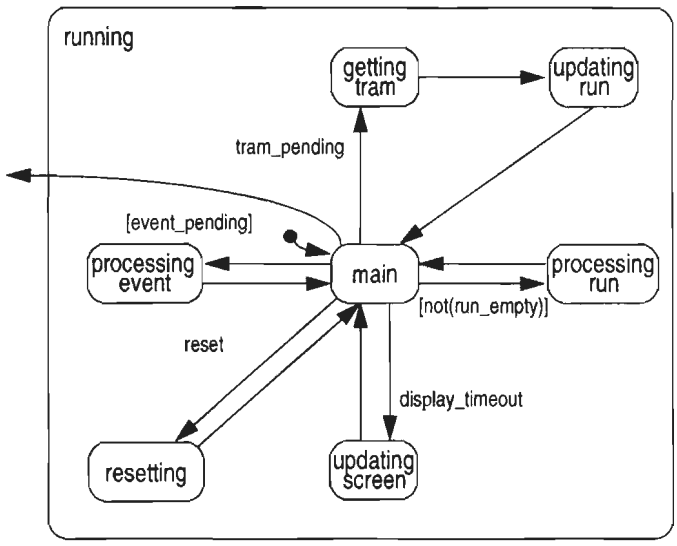

FIGURE 19. sub-states of state RUNNING

4.2 Using the existing trace message structure

At the start of the Viper project there was already a trace message system in place to allow deadlock detection and program termination. This system used the following trace message structure:.

field	type	description
time_stamp	integer	system clock value at time of trace message generation
event_type	enumerate	Mona Lisa event type to which the trace message relates, e.g. *exposure, inglb* etc.
sub_type	enumerate	further description of the event type, e.g. *start* (of inglb), *end* (of inglb), *instant* (exposure).
source	integer	id. of the module, involved in a send event or internal event
destination	integer	id. of the module involved in corresponding receive event
var	integer	id. of the global variable involved in the event
offset (optional)	string	vector specification of variable slice (for instance sub-array)
size (optional)	string	,,

TABLE 4. Description of trace message structure

Apart from the last two fields, all trace message fields were mandatory. The integer value -1 was used to indicate an undefined value. Figure 20 shows an example of such a trace message.

```
12100 M_EXPOSE INSTANT 2 -1 3 (1,1) (5,5)
instant exposure of variable 3, segment [1,5] by [1,5], in module 2, at time 12100
```

FIGURE 20. Example of a trace message

The program manager, after collecting and analysing the trace messages, simply discarded them, and (if applicable) wrote a message on the screen indicating the program deadlock or abnormal termination. For the first Viper prototype, we decided to build on the existing trace message structure, extending and modifying it where necessary. First, we streamline the communication:

- The program manager reports deadlock conditions and program termination by generating its own trace messages (with the same structure, module id = -1) instead of writing a message to the screen.
- The program manager stores all trace messages in a file or communicates them directly to Viper. In both cases, trace messages are carriage return separated, ASCII formatted strings with individual fields separated by a single space.

The next step is to make sure we can capture all Mona Lisa send-, receive- and internal events with an appropriate trace message. Each event category is directly mapped to its own trace message *type*, by incorporating the (*event_type, sub_type*) combination as fields in the trace message structure. In practice this means extending the list of trace message types, as not all of the 33 event categories had been identified originally.

The trace message format thus constructed has enough expressive power to meet our requirements. However, we have disregarded any efficiency issues. The use of text strings for the trace message type for example makes the trace message longer than strictly necessary. This has performance implications in terms of trace file sizes and trace message processing. We will work on optimisation of the trace message structure in Section 5.4 .

4.3 Choosing a graphical user interface development tool

A major component of Viper is the GUI. The development of GUI's is supported by graphics packages. These packages provide plug-and-play building blocks commonly referred to as *widgets* (for example drop-down menus) that can be customised and integrated with the rest of the application. Several packages are available, we mention here GKS, PHIGS, Tk/Tcl, X11 and InterViews.

The added value of a graphics package compared to a simple set of drawing routines lies in the built-in protocol that the application can use for screen updates, commonly referred to as the object-handler-viewer paradigm. In this paradigm, the object that needs a graphical representation automatically notifies a handler when it changes, and the handler subsequently triggers one or more graphical views on this object to redraw themselves. Most packages support this paradigm at the time of writing, because it allows the different parts of the application to stay relatively independent so they can be modified, extended or replaced separately.

We decided to develop Viper's interface with the InterViews graphics package [Lin92]. InterViews has been developed by Stanford University and Silicon Graphics. We chose InterViews for:

- The power of expression of the graphics primitives. Tk/Tcl for instance supports high-level constructs that allows a whole menu to be generated with a few commands, but InterViews allows in addition the use of low-level rendering routines for detailed control over the screen.
- The portability / flexibility of the package. Although not necessarily restricted to X11, InterViews is an object-oriented extension of the X11 Toolkit, and can therefore profit from the wide availability of X11 across many hardware platforms. The language interface is C++, the de facto standard at time of development.
- The user interface standards it supports. Independent of the graphics primitives used, the look-and-feel can be chosen at run-time to be either Motif compliant, OpenLook alike or a native crossing between the two.
- The complexity of the package. On the criteria mentioned above, the X11 Toolkit also seems a favourable choice. However, the level of complexity is much higher compared to InterViews.
- The performance of the package. The demonstrations that were included with the package gave a positive impression on speed, although it is still difficult to compare the performance of graphics packages, especially if they use different mechanisms/ primitives to implement a similar benchmark.
- The cost. InterViews is public domain software, and as a result there are no license requirements.

InterViews, like many other graphics packages, uses a model of the application environment, where all user interface actions (elementary drawing actions, keyboard input, mouse movements) are modelled as *events*. These events are processed in an event dispatching loop: at set times, an event handler takes events one by one out of a FIFO queue and executes/processes the action specified.

The screen update mechanism also makes use of this loop, in the following way: when a screen area needs to be redrawn, the application notifies InterViews that a part of the drawing canvas is *damaged*. Interviews registers the damage, but decides itself when to trigger the appropriate graphics components to redraw themselves on the canvas. Part of this decision process is InterView's attempt to perform some optimisation on the drawing process.

When a graphics component is triggered by InterViews to draw itself on the canvas, it sends drawing commands such as "draw a line from (x0,y0) to (x1,y1)" to the event dispatching loop, where the drawing commands are stored as *draw events*. Only when the event handler processes the event, will the actual line be drawn.

This screen update procedure may seem rather elaborate. However, a simple application needs to interface to InterViews only via the canvas damaging routine and the appropriate graphics components; the event processing loop stays under the control of the graphics package 'engine' and is not of interest to the application.

What may seem a modest amount of trace messages, may in fact represent a significant amount of graphics processing, depending on the complexity of the Mona Lisa events involved (e.g. draw a new circle with colour change and arrow). Assuming an execu-

tion model for Viper of sequential processing, this means that the processing of trace messages for state updates is sometimes significantly interrupted by graphics processing. This is an issue for on-line processing in the case of trace buffering constraints: if the delay in trace message processing causes a buffer overflow, trace messages are either lost (worst case), or the parallel programme execution has to be delayed to allow for Viper to 'catch up'. The latter requires appropriate support of the parallel processing environment.

However, we can take full advantage of the flexibility of InterViews' screen update mechanism. By unfolding the event processing loop we can give Viper explicit control over what InterViews does (when it checks the event queue, when the event handler processes the next event, etc.). In this way we can use one scheduler to deal with real-time constraints for both trace message processing (during on-line visualisation) and graphics processing in a uniform way.

4.4 The on-line interface between Mona Lisa and Viper

The Transputer Machine in the GP-MIMD project is controlled from a SUN workstation. The communication between the two machines takes place via a proprietary protocol that is Transputer specific. However, from a portability point of view it is unattractive to base the on-line interface between Viper and a Mona Lisa program on this protocol.

Mona Lisa is not implemented directly on Transputers, but makes use of the Chorus operating system as an intermediate layer. The Chorus/Mix extension of this system provides UNIX compliant functionality, including sockets. This is a standard mechanism for communication in a UNIX network. The SUN workstation also uses the UNIX operating system, so it was decided to design Viper in such a way that it reads trace messages from file or from a socket. To complete the interface with Mona Lisa, the Mona Lisa program manager needed to be changed to incorporate the possibility of writing trace messages to a socket instead of a file.

4.5 Viper's software architecture

Numerous methods for software development have been developed over recent decades. Object oriented software development is now considered to be best practice for complex mid-sized applications such as Viper. In this thesis we will use the Object Modelling Technique (OMT) method to describe Viper's software architecture, advocated by J. Rumbaugh et al. OMT is one of the leading methods in modelling object-oriented systems, together with the Booch method. At the time of writing, a merge of the two methods had been initiated towards one Unified Method, and a draft of the notational standard had been published. This was the basis for what is now widely known as UML (Unified Modelling language). We adhere to these conventions whenever possible.

In the Unified method, class/object diagrams are used to describe the static structure of the objects in an application and their relationships. Object message diagrams are sometimes useful to show the sequence of messages between objects that implement an operation. State diagrams document how objects exhibit different interactions with other objects depending on what state they are in. Other diagrams, not used in our dis-

cussions, include use case diagrams, module diagrams and platform diagrams. Finally, operational specifications using pre- and post conditions are used to specify object operations in detail.

Before we present the class diagram for Viper's architecture, we give a short introduction to the components of the class diagrams (objects, classes), their structure, and their relationships. For a more thorough treatment, the reader is referred to [Boo94].

4.5.1 About objects and classes

The execution of an object-oriented program can be described as an interaction process between a collection of *objects*. Objects represent a physical entity or a logical process in the real world, usually related to the problem domain the program addresses. Objects carry information in the form of *attributes*, and provide *operations* to read and manipulate these data. A *class* defines the exact behaviour of operations and the type of the object's attributes. From one class, several objects can be instantiated: the class definition acts as a template.

We will demonstrate the above with a small example. Figure 21 shows the pictorial description of a class Stack. Objects from this class can be used to represent a stack of books on a desk. One might envisage two stacks of books: one stack of unread books, and another of read books. In that case one would create two objects from the same class Stack (as indicated by the instantiation arrows from the objects to the class). The fact that one can have multiple stacks of books on a desk, is shown by the named line interconnecting the two class pictograms. The filled dot signifies "multiple".

In the class pictogram, attributes come directly under the class name, and a horizontal line separates the attributes from the operations. The only attribute of Stack that is accessible by other classes is top. This attribute represents the book on top of the stack. The type of top is Book, which is in itself a class with operations and attributes (necessary for reading the book!). The attribute list is used to implement the class; it is not accessible by other objects (indicated by the (-)).

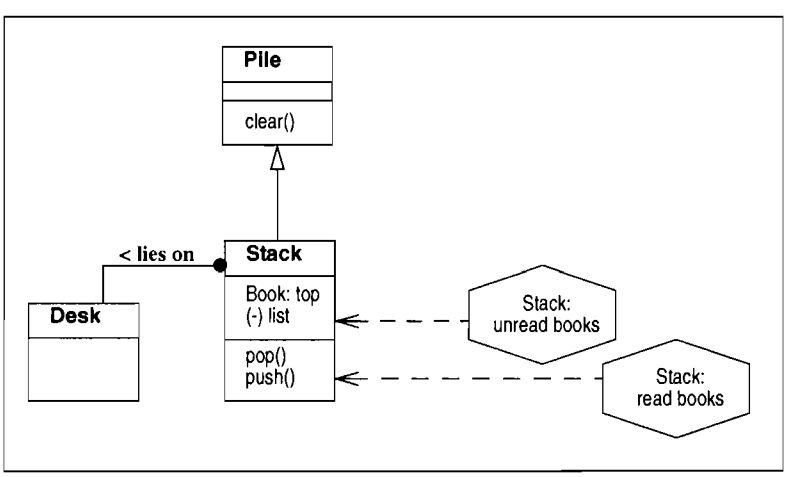

FIGURE 21. An example of a class description: the class Stack represents a stack of books, and provides useful operations to manipulate the stack

Taking a book out of the middle of the huge pile is a hazardous adventure, so the only operation that takes a book off the pile is pop, which removes the top book. Similarly, the only safe way to add books to a pile is to put them on top, so the operation push takes a book and puts it on top. The implementation of both pop and push (that manipulate the private attribute list) has to be done in such a way that the value of attribute top is adjusted accordingly, to preserve the integrity of the representation.

A stack is really a special kind of pile (namely, an ordered pile). The class Stack is therefore modelled as a sub class from Pile (indicated with an arrow), and it is said to *inherit* the operations and attributes from Pile. So, although the operation clear() is not mentioned within the Stack pictogram, it is part of the class, and accessible as such.

We can elaborate on many other object oriented concepts, but instead we will discuss them as they appear in the object models. In the text and diagrams we will use some conventions to improve clarity. All class names start with a capital. All InterViews classes (provided by the graphics package) have "iv" preceding their name. We will not show their implementation unless vital for the understanding of other classes. The general purpose classes List and Table are also supplied by InterViews. All other classes are specifically designed for Viper.

4.5.2 Viper's object model.

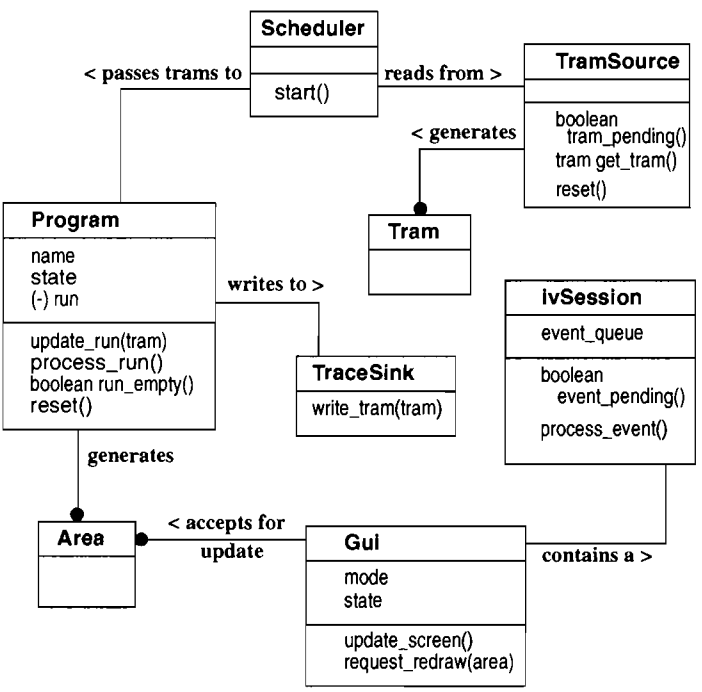

FIGURE 22. Viper's architecture as cooperating classes.

The abstract model is a good starting point for identifying classes and mapping activities (for which we have given informal operational specifications) to operations. Start-

ing with the activity *get tram*, with the associated external actor *Mona Lisa program* and the event *tram_pending*, how to create a suitable class structure that captures this functionality? An obvious choice is to create a class whose objects represent a Mona Lisa program. The activity *get tram* and the event *tram_pending* can then be implemented as operations of that class. To avoid a Mona Lisa dependent nomenclature, we prefer to use a more abstract class name, namely TramSource.

The class TramSource embodies the input device from which Viper expects to get its trace messages. The operation tram_pending returns a boolean value indicating whether or not the next trace message can be read. The operation get_tram implements the activity of the same name; it returns the next trace message under the precondition that tram_pending returns "true". Reset has as postcondition that the input medium (for instance, a trace file) is read from the start again. This operation is part of the implementation of the activity *reset*.

The class Program is the mirror image of the Mona Lisa program. Whereas the real program changes state and produces trace messages, this class does the inverse: Program absorbs trace messages and updates its state as a result. Hence, it incorporates both the *run* and the *program state* store from the StateMate activity chart. The operations *update_run* and *process_run* implement the activities of the same name. The run reconstruction is an implementation detail of the class, so run is modelled here as a private attribute. The program state is (for now) represented by the attribute state. The operation reset() clears run and resets state.

The class Gui represents the graphical user interface, and indirectly the user. The two attributes shown provide the scheduler with the information it needs according to the dynamic model; mode indicates whether Viper is in on-line mode or off-line mode, and state is set according to the events that the user generates (such as "press go/stop button in control panel"). These attributes are updated by so-called *callback* routines in the Gui class (not shown).

The Gui class is also responsible for the graphical representation of the program state, so it implements the activities *update_screen* and *process_event*. A screen update is set up by providing one or more Area objects to request_redraw(); these objects specify what part of the screen needs to be redrawn. Invoking update_screen() then generates graphics events in the event_queue, an attribute of the ivSession class. The operation event_pending returns "true" if the queue is not empty, whereas the operation process_event takes an event from the queue and passes it to the InterViews event handler for processing (causing the actual screen change).

Finally, Scheduler is shown as a simple class with one operation, start(). This operation only returns to quit the application. This makes the scheduler class the active thread of control. The other thread of control is the windowing system. The graphics event queue is in fact an integral part of this system, and normally its processing is done in a closed loop. As shown in our modelling, we decided to break this loop so the scheduler can decide at any particular moment which of the following three actions has the highest priority: reading in a tram (a priority in on-line mode to avoid buffering/blocking on the other side of the socket!), processing the run queue, or processing the event queue. This behaviour has already been modelled in the StateMate state diagrams, and the object state diagram for the scheduler class is a direct equivalent. We therefore decided not to draw a new object state diagram, and used the StateMate diagrams directly instead.

This concludes the top-level description of Viper's software architecture. This top-level can be expanded to refine our software design until we have reached a sufficient level of detail, beyond which an implementation in a programming language is straightforward. As starting points for refining we can use both the underlying StateMate activity model and the OMT object model. StateMate supports a hierarchical diagram structure, so the activity model can be expanded simply by describing an activity in more detail in a new (sub)diagram. Also, implementation of the classes of the object model provides a good starting point for introducing new classes and state diagrams.

As an example, we will discuss the implementation of the Program class. In Figure 23 we see that a Program object is an aggregate of a Run object and multiple Module objects representing the modules that together constitute the parallel program (aggregation is indicated by the diamond). Program also has a state object of class ProgState, representing state information that is not related to any particular module or that relates to more than one module (such as "program in deadlock"). The attribute PSclock (Program State clock) of class Program contains the time stamp of the Tram object that last updated a module's state or the program state.

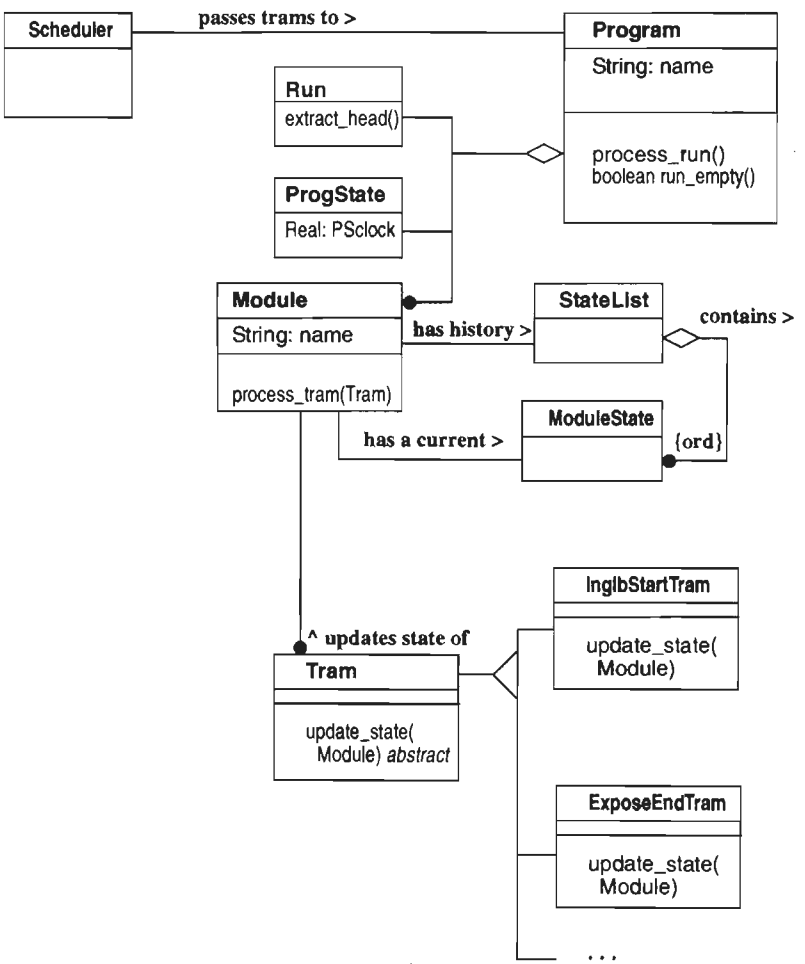

FIGURE 23. Object diagram for class Program, with a focus on run processing.

The class Module has a current module state associated with it, and a state history. This history is an ordered list of module states. The update of a module's state proceeds as follows. The Program takes the first Tram object out of the run with extract_head(), and passes it on to the module that is indicated by the association "updates state of" between Tram and Module, through the operation process_tram(). The Module then invokes the operation update_state() on the Tram object, passing a reference to itself as the argument. So it is the Tram object that actually updates the state of a module.

As each state change can be radically different depending on what Mona Lisa event a trace message represents, we have a subclass of Tram for every trace message that has a different event tag. The diagram shows two of them, one being used for trace messages that indicate the start of an inglb call, and one indicating the end of an exposeglb call. For each subclass we can implement the operation update_state() differently. In fact, the operation is explicitly left undefined within the Tram class (indicated by the word *abstract*). When update_state() is called on an object of class Tram, the correct operation implementation of the appropriate subclass is automatically selected. This concept is called *polymorphism* in object oriented languages.

The description of Viper's abstract and software architecture in this section has been rather high-level, and especially the class diagrams deserve more detail. Nevertheless, it is not the purpose of this document to provide a detailed technical design of Viper. Instead, we focus on the major design and implementation issues, for which we have made a good starting point. The process of translating diagrams such as the ones shown here into actual, compilable code is a subject that is described in an excellent way in [Str92].

4.6 Evaluation of the first prototype

The development of the first prototype was a time restricted project with a strict deadline of December 1993. This was due to the nature of the project, it being the final stage in the two-year postmasters programme of Software Technology. As such, it was essential to deliver a fully functioning prototype.

On the whole, the results of 9 months work were satisfactory. The primary objective of providing an effective graphical presentation and dealing with large amounts of data whilst providing the user with easily interpretable diagrams was achieved succesfully. This was validated with artificial example programs.

The development of the first prototype involved an intensive learning process in several areas:

- Understanding of Mona Lisa from concept to implementation. During the development, some unforeseen improvements and additions had to be made to the Mona Lisa implementation. In addition, the functionality of the Mona Lisa front-end Fortran parser had to be extended. This was partly the result of being the first serious user of the paradigm.
- Usage of the InterViews graphics package. A considerable drawback was the lack of documentation and direct support, especially since there was no prior experience with other graphics packages similar to InterViews in the project group. This caused the learning process to be quite long.

- Global timing in distributed systems. The characteristics of Chorus in this area were such that more time had to be spent on the consistent run construction then initially foreseen. Even with the logical clock implementation as described in this chapter, the result was not entirely satisfactory (the assumption of a maximum message delay being quite restrictive and inflexible).

The design decision to use InterViews had two major impacts:

- Due to the learning curve with InterViews, the full delivery of the Viper gui was very much skewed towards the end of the project. The time investment that was required also had an impact on the functional scope of Viper, which was kept to the essentials.
- On the other hand, the functionality of the package, its object oriented design, and the C++ compatibility with Viper ultimately benefited the quality of the application design.

Our design decision to separate run construction and run processing as activities in their own right (see Figure 15) has proven to be an important one. Given our concerns about the logical clock implementation as described above, we will most probably have to overhaul this part of the design in the next prototype. The modularity of the design allows us to make changes here without impacting other parts of the design.

This chapter has introduced our vehicle for describing Viper's first prototype, StateMate and the Unified method. In the next chapter the enhancement of the first prototype will be addressed. When we describe the second prototype we will expand on the models that have been presented here.

Chapter 5

Viper's subsequent prototypes

The delivery of the first prototype after 9 months concluded the final project from the two-year post-masters programme in technological design. Although Viper proved usable on example programs, there were some key points that needed further addressing, most notably the consistent run construction. In the development of the second prototype we therefore focused on an improvement of this part of the design. This will be the topic of discussion in Section 5.1 through to Section 5.3. In a third prototype we finalised the trace file format, the on-line interface to the parallel program and the interface hooks to ParaGraph, which will be discussed in Section 5.4.

5.1 Moving away from logical clocks

In Figure 11 we already demonstrated the two disadvantages that are associated with the use of logical clocks:

- Each communication between modules needs to be accompanied by a logical clock value, an undesirable overhead. The logical clock value also has to be incorporated into the trace message structure.
- It is difficult to construct a consistent run without any assumption on latest arrival time of trace messages, because the logical clock values within a module are not consecutive (i.e. there is no so-called gap detection possibility).
  An attempt had been made during implementation of the first prototype to implement a partial gap detection mechanism using logical clocks in combination with primitive counters, see [Schi93/1] for details. This still did not solve the problem of the delay between trace message receiving and processing, and it further increased the trace message size. Therefore, the search for a better technique continued.

The strong point of the logical / vector clock theory is that it provides a widely applicable solution because there are no assumptions about the interaction patterns between modules. However, by exploiting the specific characteristics of our programs, we can take a different approach that does not suffer from the disadvantages mentioned, which we will demonstrate in the remainder of this section.

The run construction as we have proposed it for Viper is a two-phase process. We start by partitioning all observed events into *sequence lists*. A sequence list is a chronologically ordered list of all events generated in one particular module. At the same time, we partition all observed events into *primitive lists*. A primitive list is a *consistent* and *complete* list of all events generated by one particular primitive call (which may contain events belonging to multiple sequence lists). These primitive lists are subsequently concatenated into *module lists*. A module list is a chronologically ordered list of all primitive lists that have originated in one particular module. The second phase of the

run construction process consists of merging the sequence lists into a *run* respecting the ordering imposed by module lists. Thus, a run is one serial observation that obeys the ordering of both module lists and sequence lists.

We will show later how we can extend Viper's logic with a model of the Mona Lisa primitives that will help Viper to place an (initially anonymous) Mona Lisa event into a primitive list, thus enabling Viper to build module lists by concatenating primitive lists.

Figure 24 demonstrates the concepts of primitive list and module list, using the same realistic example from Figure 10. Although we have kept all the events in the diagram, numbering them according to their occurrence in time, the reader will notice that some events (such as e_3) are not relevant for the discussion. For the moment we will assume that the corresponding trace messages are observed by Viper in the same sequence. Part A shows the ordering of events in the primitive list for the slave's inglb call. The sequence of events that make up the primitive list is indicated by a bold line that joins them together (e_1, the start of the inglb call; e_2; e_4 and e_7 that completes the inglb).

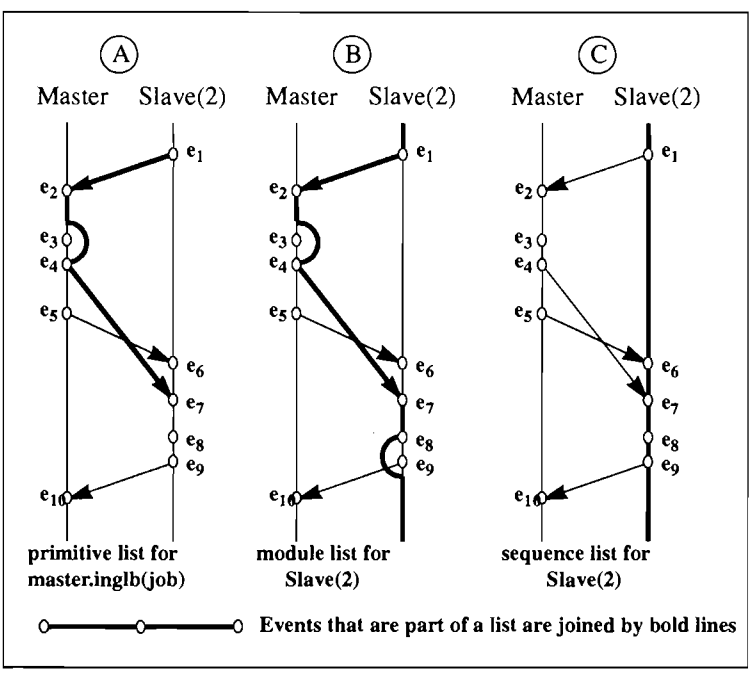

FIGURE 24. Primitive list, Module list and Sequence list demonstrated in the Farm program

Event e_8 forms another primitive list in its own right, generated by the exposeglb primitive call. The bold line in Part B of Figure 24 shows how the two primitive lists belonging to the inglb and the exposeglb list are incorporated in the module list for the slave module. Part B also shows how different module lists can 'cut through' each other: the events e_5, e_6, e_9, e_{10} are part of the master's module list. Following Mona Lisa primitive list properties can easily be understood using the example of Figure 24.

property 1.
Every send-receive pair in Mona Lisa belongs as a whole to one primitive list. This is demonstrated by the inglb primitive list in Figure 24.

property 2.
Every Mona Lisa primitive list starts with an event that is generated within the module generating the primitive call.

We will use these properties to help us with the construction of primitive lists in Viper. Note however, that these properties are not valid for just any paradigm as shown by the following example: property 2 would not hold for a paradigm where a *receive*() primitive exists; this primitive would have primitive lists associated with it that start with the send event, which is generated in a different module.

As mentioned before, primitive lists are combined into *module lists*. A module list, associated with a certain module, concatenates all primitive lists that are generated by that module. The primitive lists in a module list are ordered in sequence of execution: the first list corresponds to the first call in the module execution etc.

Property 3.
Module lists are consistent. This follows directly from the fact that a module list is a consistent concatenation of consistent primitive lists.

We can make the following observation: every send-receive event pair belongs as a whole to one primitive call and is thus part of a module list. This means that the module lists' ordering respects the direct causal dependency for send-receive pairs. The other direct causal dependency (i.e. between successive events in one module) is respected by the sequence lists: in every module's sequence list, events are ordered according to the sequence in which they are generated by the module. See Part C in Figure 24.

Viper constructs a consistent run by merging the sequence lists, using the module lists to ensure correct ordering:

- The module list ordering ensures we do not violate the causal dependency between a send and receive event.
- The sequence number ordering ensures we do not violate the causal dependency between two events in the same module.

Merging takes place by continuously transferring the 'heads' of the sequence lists to the run list, with the following rule: an event is only transferred when its predecessor in its module list has been transferred. In the algorithm below (shown in pseudo code) we leave out the construction of the module lists for the moment (note that in the algorithm we manipulate trace messages, not events):

```
Process observer
VAR
        m : Module;
        q[m: Module] : Array of Queue; {array of sequence lists implemented as message queues}
        r : Run; {message queue}
        t : Tram;
BEGIN
        WHILE True DO
                IF (    m,t : t in module list of module m : next_approved(m) = t) THEN r.append(t);
                FI
                IF receive(t,m) THEN q[m].insert(t);
                FI
        OD
END
```

The function *next_approved(m: Module)* in the above algorithm returns the next trace message for which the predecessor m_{pred} in the module list is already in run *r*, or nil if such a trace message does not (yet) exist.

The function *receive(t: Tram; m: Module)* returns true if a new trace message has arrived for module m (which is then returned in variable t), and false otherwise.

5.1.1 The construction of sequence and module lists

The construction of a sequence list is relatively straightforward. We tag each event, sent in a trace message to Viper, with a sequential number that defines the ordering. This sequential number is derived from an internal counter maintained within the module.

The construction of a module list as a concatenation of primitive lists is demonstrated by using our example program with five modules, Slave(1) through Slave(4) and Master. Let us assume that for module Slave(2) we have constructed the first n-1 primitive lists, and we now aim to reconstruct primitive list n.

Figure 25 shows the situation we aim to obtain. Holes in the sequence lists (indicating events that have not yet been observed) are marked white. For events that have been observed, the corresponding squares are marked red. As events are allocated to a primitive list we will mark the corresponding square yellow, joining subsequent events in different sequence lists with arrows.

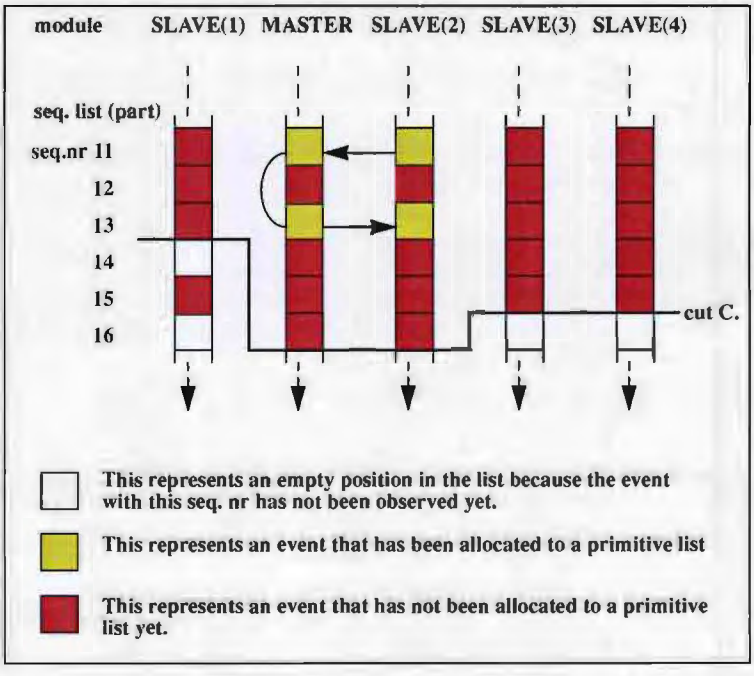

FIGURE 25. Situation after completion of primitive list n for Slave(2).

The line across the sequence lists marks up to which point the set of sequence lists presents a complete observation: the line passes via the first missing event in each

sequence list. In the primitive list construction process we can only take the observation up to cut C into account. For instance, event $e_{slave(1),15}$ cannot be incorporated in any primitive list until $e_{slave(1),14}$ is received to determine whether it is a better candidate. This means that the information below cut C cannot be used. This information is usable once cut C has 'descended' in the diagram.

The first step in constructing primitive list n of Slave(2) consists of identifying the event that starts the list. As property 2 indicates, this event must be found in the sequence list of Slave(2), among the red events above cut C. The trace message type marks for an event whether or not it is the start of a primitive call (refer to Table 3). Assuming in our example that events $e_{Slave(2),11}$ and $e_{Slave(2),14}$ are both the start of a primitive call, we must take $e_{Slave(2),11}$ as the first event in the list n.

The trace message type of $e_{Slave(2),11}$ immediately tells us:

- the number of events list n is composed of;
- which trace message type is expected for each position in the list;
- which sequence lists contain these events.

In our example $e_{Slave(2),11}$ is the start of a call Master.inglb(..). Therefore, list n contains events as listed in Table 3 which must be found in the sequence lists of Slave(2) and Master.

Once the first event of list n has been found, the process of list construction continues by identifying the second event for list n (of which the type and source sequence list is now known), then the third event, etc. For example, the second event with trace message type INRQ RECEIVED is searched for in the sequence list of Master. As with the first event, we look for the red events above cut C of the required type. Assuming that both $e_{Master,11}$ and $e_{Master,16}$ in list Master are of the required type, which one should we take?

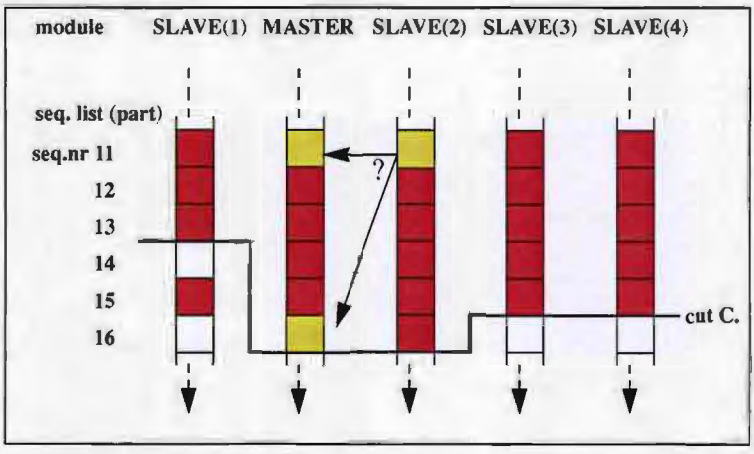

FIGURE 26. Finding the correct 'receive request' event as part of the inglb call. We will show that such choices cannot occur.

Figure 27 shows the general case of a parallel program where send-receive pairs can cross-over in time. Situation A can be distinguished from situation B by an external

observer with a technique such as logical clocks. However, we will show that such choices can never occur.

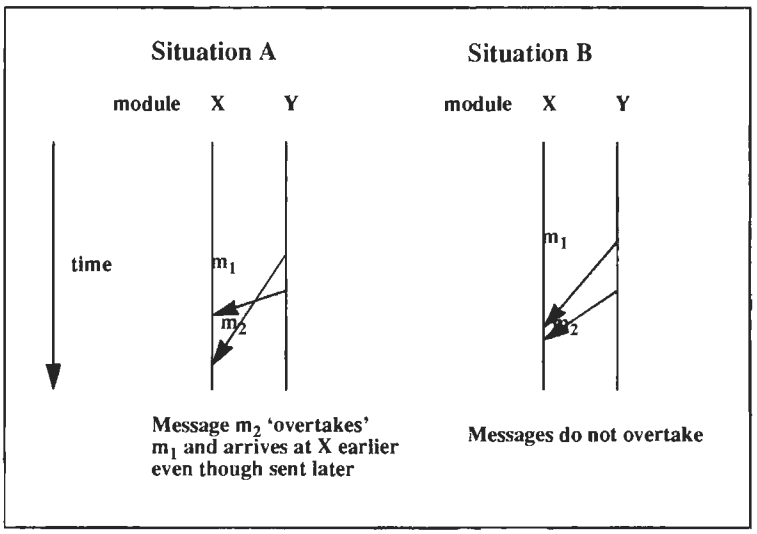

FIGURE 27. Generally, modules might interact with messages that 'overtake'.

Situation A can never occur in Mona Lisa programs because of the way primitives are structured: the 'request-reply' protocol imposes the first send to complete with a receive before the next reply (e.g. acknowledgement) or request is sent, as demonstrated in the next diagram.

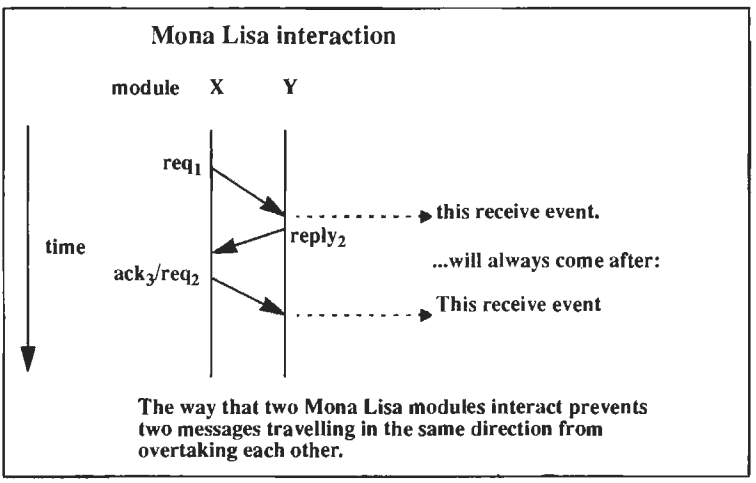

FIGURE 28. The request-reply protocol in Mona Lisa prevents 'overtaking'.

Therefore we know that we can always take the *first* event in the sequence list with the appropriate type and from a given source. In our example this is $e_{Master,11}$ (and event $e_{Master,16}$ must belong to a primitive list >n). After completion of the primitive list construction process as described above, we obtain the diagram of Figure 25. After con-

struction of all the master and slave(2) related primitive lists (an exercise we leave to the reader), we obtain the diagram of Figure 29 showing the partial module lists of the two modules.

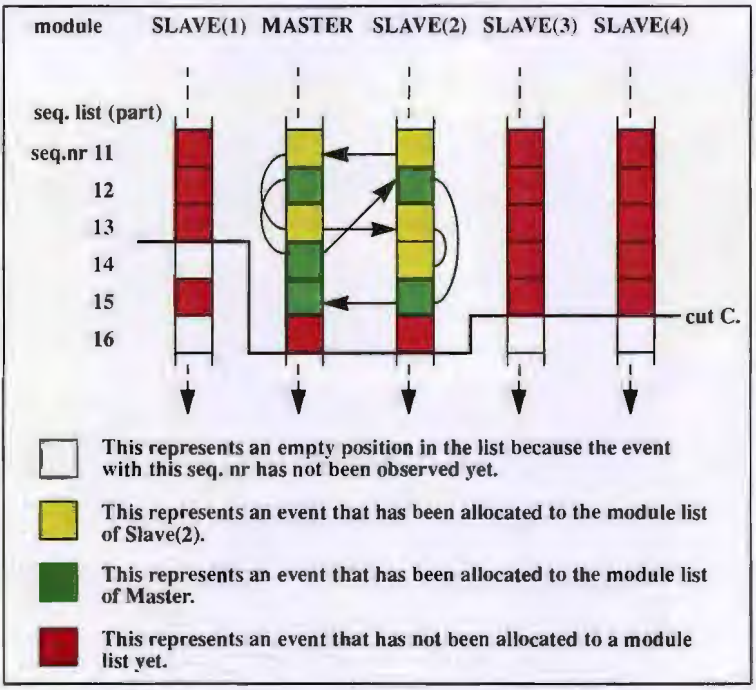

FIGURE 29. Situation after completion of multiple primitive lists.

In the next section we will look at a formalisation of the primitive list construction process.

5.1.2 The algorithm of primitive list construction

We have seen that trace messages can only be allocated to primitive lists when some conditions have been satisfied (e.g. all predecessors of relevant sequence lists have arrived; profile of the primitive has been identified). For this reason, trace messages go through various states:

- INITIAL - the trace message is in the sequence list but below the cut (so its predecessor has not been observed yet);
- QUEUED - all predecessors are also in the sequence list;
- INTEGRATED - the trace message is assigned a position in a primitive list;
- TRANSFERRED - the trace message has been merged into the run.

Tracking these trace message states in the run construction algorithm is straightforward as demonstrated below (again, in pseudo code):

```
Process observer
TYPE
        PrimList = record (
                list[i: Integer]: Array of Tram;
                type : PrimitiveType;
                size : Integer;
                );
VAR
        m: Module;
        t: Tram;
        q[m: Module] : Array of Queue; {array of sequence lists implemented as message queues}
        r : Run; {message queue}
        prim_list[m: Module] : Array of PrimList;
                {one PrimList per module that holds the primitive list under construction}
BEGIN
        WHILE True DO
                {1. promote tram statuses to INITIAL or QUEUED where possible}
                IF receive(t,m) THEN
                        q[m].insert(t);
                        t.status = INITIAL;
                        FOR ALL t in q[m]
                                IF head(t) or predecessor(t).status=QUEUED THEN t.status:=QUEUED;
                        ROF
                FI;
                {2. start new primitive list if possible}
                FOR ALL m
                        IF prim_list[m] is empty THEN
                                {find the first t in q[m] with status QUEUED" that starts a primitive call}
                                IF (∃ t: t in q[m]:t.status = QUEUED and is_primitive_start(tram_type(t))) THEN
                                        prim_list[m].list[1]:=t;
                                        t.status = INTEGRATED;
                                        prim_list[m].type = primitive_type(tram_type(t));
                                        prim_list[m].size = primitive_size(tram_type(t));
                                FI
                        FI
                ROF
                {3. promote tram statuses to INTEGRATED where possible}
                FOR ALL m
                        FOR ALL t in q[m]
                                IF accepted(prim_list[m],t) THEN
                                        prim_list[m].list.insert(t);
                                        t.status = INTEGRATED;
                                FI
                        ROF
                ROF;
                {4. promote tram statuses to TRANSFERRED where possible}
                IF (∃ m,t : t is present in q[m] and t.status=INTEGRATED
                        and (head(t) or predecessor(t).status=TRANSFERRED)) THEN
                                r.append(t);
                                t.status = TRANSFERRED;
                FI
        OD
END
```

The function *predecessor(t: Tram)* in the above pseudo algorithm returns the predecessor of *t* in the sequence list that *t* belongs to. The function returns an empty Tram value with the status field set to INVALID in those cases where a predecessor does not exist, i.e. when *t* is not part of a sequence list yet, or when *t* is the first element in the sequence list. In the particular case where a tram is the first element, the function *head(t: Tram)* returns true.

The following four functions reflect the fact that we can derive many properties from the type of a trace message as explained on page 65.

- The function *ttype(t: Tram)* returns the tram type of *t*.
- The function *is_primitive_start(tt: TramType)* tests whether the tram type passed corresponds to the start of a primitive call. For example, with *tt* equal to INRQ RECEIVED this function would return false, as this can never be the start of a primitive call.
- The function *primitive_type(tt: TramType)* maps tram types to primitive types. For example, only inglb primitive calls start with a trace message of type INRQ START. Hence, primitive_type(INRQ START) returns INGLB.
- Similarly, the function *primitive_size(tt: TramType)* returns the number of trace messages that are required to make up the complete primitive list using the fact that the primitive type can be derived from the tram type.

The function *accepted(p: PrimList, t: TraceMessage)* tests whether the trace message conforms to the conditions as outlined previously (i.e. of all messages with status QUEUED, appropriate type and appropriate sequence list, it must be the one with the lowest sequence number).

The algorithm does not show the clean up of finished primitive lists. The prim_list structure is meant to hold only the primitive list currently under construction for each module. Once this primitive list has been fully constructed, the algorithm proceeds with the construction of the next list, starting with an empty prim_list. The incorporation of this logic into the algorithm would be achieved by inserting the following pseudo-code segment after step 4:

```
.....
{5. clear primitive lists that have been incorporated into run}
FOR ALL m
        IF (    i : 0<i<=prim_list[m].size : prim_list[m].list[i].status = TRANSFERRED) THEN
        prim_list[m] := empty;
ROF
.....
```

The run construction algorithm as presented here does not suffer from the disadvantages associated with logical clocks:

- The additional overhead is small. Although sequence numbers need to be attached to trace messages just like logical clock values, there is a difference regarding the communication between modules. Whereas logical clock values need to be attached to messages between modules, the algorithm presented here does not interfere with the inter module communication at all; the sequence number generation is done locally (inside a module). There is also no extra cost associated with the determination of the trace message type: trace messages already contain type information to distinguish, for instance, send events from receive events.
- The approach has an advantage over logical clocks in that it allows the generation of sequence numbers at a later stage. Imagine for example an MIMD parallel processing architecture where trace messages are cached on the local processing node. At the point of parallel processing completion, trace messages are collected from all nodes through FIFO communication channels; finally, sequence numbers are attached to trace messages at the point of collection.

- The reconstruction algorithm adds a trace message to the run list as soon as the trace messages it causally depends on have been added to the run list as well. Therefore the delay between trace message reception and parallel program state update is minimal under the consistency constraint.

5.2 Wall clock time of the observer process versus 'global system time'

A parallel program state as displayed by Viper refers to the state of the program *at a certain point in time*. Viper attempts to give the user an indication of this point in time, by maintaining a virtual clock in the status window. This clock "ticks" on the basis of the time stamps, stored in the trace messages.

The Chorus operating system provides a clock on each node which is used to time stamp trace messages. The clock values have a precision of 1 ms. However, the clocks tend to drift slightly, both between themselves and also in relation to the wall clock time. Drifts in the order of 10-20 ms. have been noted. The Chorus operating system therefore does not provide a global system clock.

In the absence of a global system clock, we have to construct our own virtual global clock as an approximation. This clock still "ticks" on the basis of the time stamps, but when trace messages are collected by Viper, their (unreliable) time stamps may sometimes be in conflict with the ordering imposed by causal dependency. This is then corrected in the process of run construction as outlined below.

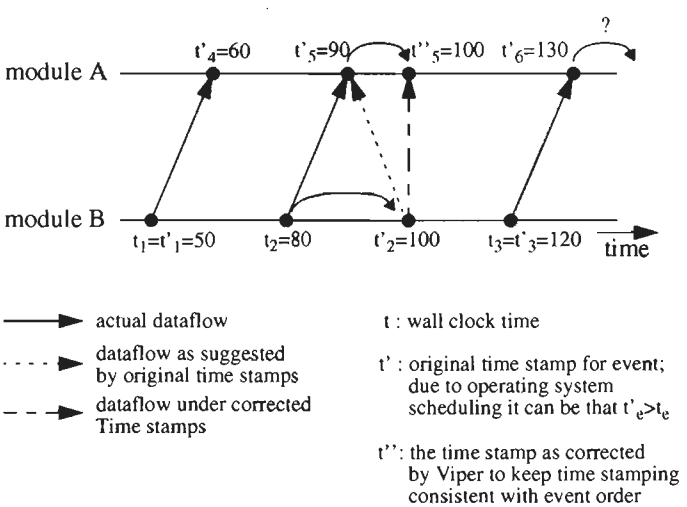

FIGURE 30. Time stamp adjustment to make time stamps consistent with event order.

Figure 30 shows the time line for two modules that exchange messages, and the time stamps of the associated send and receive events. In this scenario the operating system delays the read out of the system clock (by internal rescheduling) such that time stamp t'_2 does not correspond to the 'real' time t_2. When the time stamps are interpreted as wall clock time, this gives the incorrect observation that a message has been sent at

time t=100 and is received at time t=90. Viper tries to correct this by setting the **clock offset** for module A to 10, thereby shifting t'_5 to t''_5. Now at least messages do not travel back in time: the observation is changed to module A instantaniously receiving a message sent by module B.

The question arises however what should be done with later time stamps of module A. In this scenario, it is reasonable to set the clock offset back to 0, as the incorrect time stamp t'_2 was just an incident. But if in fact the clocks of module A and B are not aligned correctly, it is better to keep the offset at 10 for all future trace messages. To allow a smooth compromise between the two options, we introduce a **clock offset decay parameter**.

The clock offset decay parameter determines the evolution of the clock offset over time. A value of 0.5 for instance means that with every trace message the clock offset parameter is reduced by 50% (unless consistency constraints dictate otherwise). The values 0 and 1 correspond to the incidental time stamping error and the misalignment of clocks respectively.

The progression of system time can now progress in line with the construction of the consistent run: the global system time is set equal to the maximum time stamp value of events in the run constructed so far.

The implementation of the concept described above involves the following steps. With every module we associate a time offset that indicates how much the time stamps of trace messages generated by that module are "out". When trace messages are ready to be transferred to the run list, their time stamp is adjusted if necessary by comparing the current time stamp *plus* the time offset with the time stamps of the trace messages it *directly* causally depends on. The maximum of these values will be the new time stamp. If the gap between the old and new time stamp exceeds the current time offset, this offset is updated to reflect the need for a higher offset value for future trace messages. The offset value is subsequently multiplied by the clock offset decay paremeter. Finally, the new time stamp is used to update the global clock.

Summarising, the time stamp re-calculation ensures two things:

- within a send-receive event pair the send event will never have a time stamp greater than that of the receive event. The time stamps for successive events in the same module also remain ascending. Hence, the new time stamping respects causal dependency.
- if there are no send-receive event pairs to be taken into account, the time distance between successive events is not disturbed (at most event occurrences will be translated in time due to a non-zero module offset).

With a clock offset decay parameter value of 1, module offsets can only increase, possibly resulting in a stretched time axis in Viper's space time view. Setting the parameter to a value lower than 1 allows module offsets to reduce gradually by a fixed percentage each time a time stamp calculation is performed, provided the dependency constraints permit this. The reduction percentage is a user defined parameter.

The run construction method as discussed in this section has been presented at the EuroMicro conference on parallel and distributed processing [SS95], as it is one of Viper's unique features: at a small cost and with minimal delay Viper can reconstruct a consistent observation of the parallel computation despite potential disturbances that

may occur in the time stamping or the trace message collection. A potential weakness of the method is its strong dependency on reliable trace message delivery; if we want to allow incidental trace message scramble or loss, there are many more issues to be taken into account, and at present this is not supported by Viper.

5.3 Implications for Viper's design

The refinement of the run construction method has direct implications for the activity *update run* of the Viper activity diagram, since this activity performs the run construction. The revised description of the activity *update run* starts with Figures 31 (a StateMate activity diagram) and 32 (an object diagram). As outlined in the previous section, the run construction involves the construction of one sequence list and one module list per module. A module list is a concatenation of primitive calls performed by one module, so it can be constructed on a primitive list by primitive list basis.

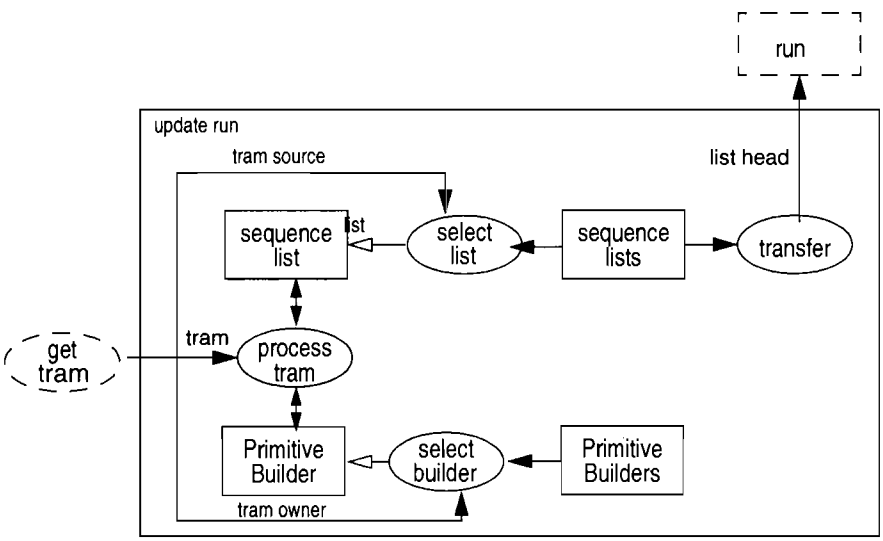

FIGURE 31. Activity diagram for the activity update run.

In the activity diagram of Figure 31 the activity *select list* selects a sequence list from all lists according to the source specification of the incoming tram. The activity *process tram* can thus insert the tram in the appropriate sequence list and update the relevant tram states in this sequence list to INITIAL and/or QUEUED as outlined in the algorithm presented earlier.

The list selection process makes sure that a sequence list receives the trams that are generated by the module it is associated with. In the object diagram we see this link is established through the association "generated by" between Tram and Module, and "has a" between Module and SequenceList.

A module delegates the task to construct a primitive list to a primitive builder. There is therefore one PrimBuilder object per Module. A tram is assigned to a builder according to the association "updates state of" between Tram and Module, which is represented in the object diagram of Figure 32. This association is represented in the activity diagrams by the tram attribute 'owner'.

In the activity diagram of Figure 31 we see that *select builder* selects the appropriate builder using the 'owner' attribute of the incoming tram. The builder subsequently absorbs the copies of the trams that are delivered by *process tram* to build the appropriate cross-dependencies between the trams (this dependency is marked in the object diagram by the association "cross depends on" for the Tram class).

Finally, the activity *transfer* moves the trams from the sequence lists to the run. The pre-requisite for moving a tram is that any tram it has a cross dependency with is already transferred. In other words, it moves a tram's status from INTEGRATED to TRANSFERRED. This activity therefore performs the last stage of the algorithm presented earlier.

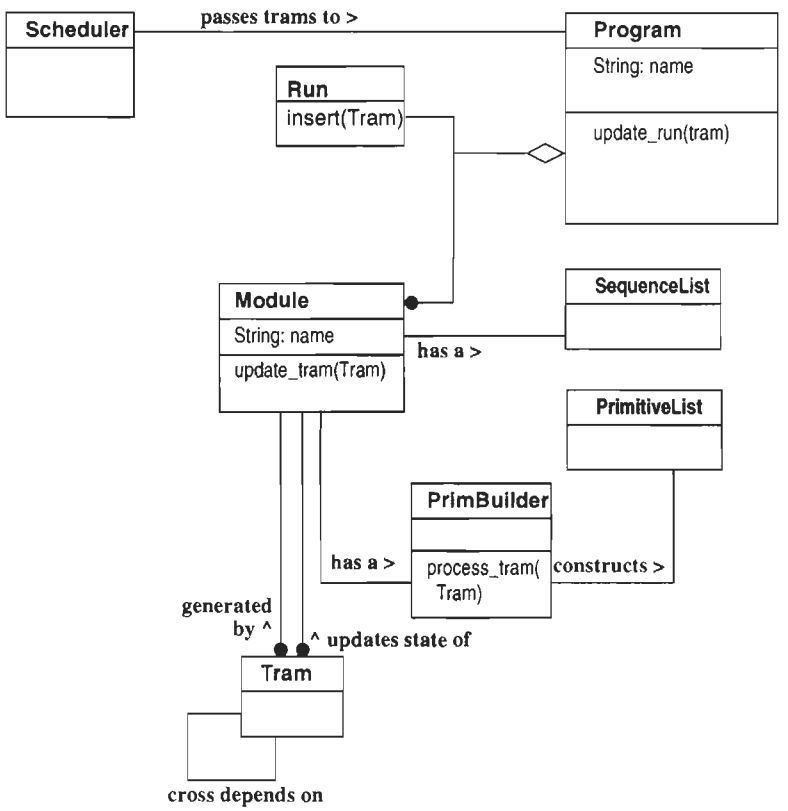

FIGURE 32. Object diagram for class Program, with a focus on run construction.

Summarising, trams are passed from the Program class to the Module whose sequence list needs updating, and to the Module whose primitive list needs updating. The Module that needs to update a primitive list passes the tram to the PrimBuilder. This implies that the activity *process tram* is implemented as part of the operation update_tram() of the Module class which passes the data to the operation process_tram() of the PrimBuilder class. The primitive list under construction is an internal data structure of that class, consisting of pointers to the tram data structures that are stored in the sequence lists. This will now be discussed in more detail.

In Figure 33 we show the activity diagram for the operation process_tram() of the class PrimBuilder. Figure 34 shows the associated state diagram.

When the primitive builder has to start a new primitive list, it waits for a tram to arrive. If the tram is of the correct source and type, the activity *select* creates a primitive object, based on the type of primitive call to which the tram belongs (for instance, inglb), and inserts this tram into the primitive. Subsequent trams are inserted as well, or, if they are not accepted by the primitive, put in a buffer. When a new primitive list is started, this buffer is *searched* and *scanned* first. The *search* activity tries to find a tram that is the start of a primitive call. The *scan* activity goes through the whole buffer and tries to find further tram candidates for the primitive.

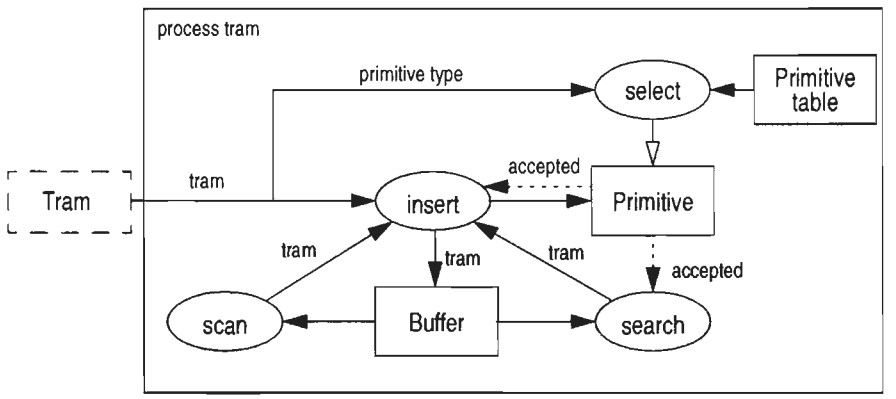

FIGURE 33. Activity diagram for process tram.

We are not interested in permanently storing the primitive list. We only need the list to establish the cross dependencies between trace messages of different sequence lists. Hence, the activity process tram (and hence class PrimBuilder) absorbs trams only to establish their cross dependencies. It does not produce any other output. Finally, Figure 34 shows the state diagram for the PrimBuilder class.

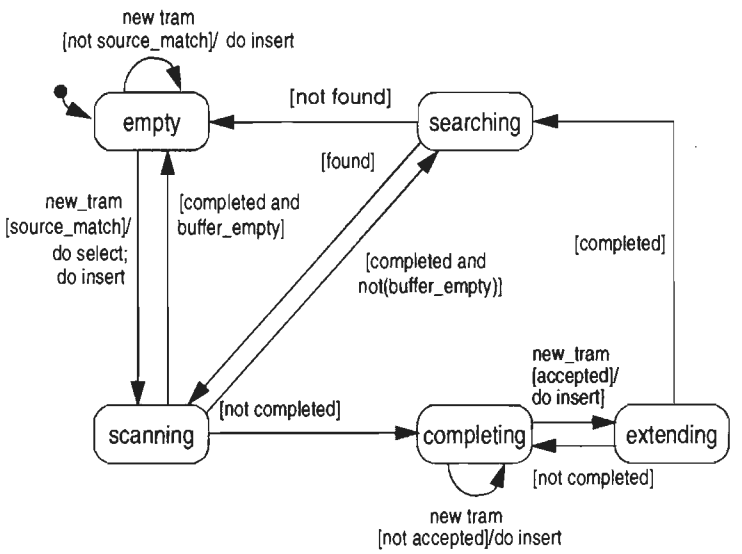

FIGURE 34. State diagram for the Primitive Builder.

5.4 Viper's third prototype

The conclusion of the optimisation of the run reconstruction process resulted in an intermediate release (the second prototype), to mark the rounding off of this work. However, three outstanding trace message issues still had to be addressed before Viper could be considered 'mature'. The implementation of these points resulted in the third prototype:

- The implementation of the real-time connection between the parallel program and Viper.
- Standardisation of trace message formatting and reduction of trace message size to reduce overhead.
- The implementation of a trace file export facility to allow tools such as Para-Graph to interpret some of the information contained in the Viper trace files.

5.4.1 Adoption of the PICL trace message format

For the first prototypes of Viper the attention was focused on the functionality of the application and minimising the requirements of the application towards e.g. the information captured in trace messages. The format of trace messages was chosen to be as expressive as possible (including textual fields in ASCII format) to facilitate debugging.

In adopting an improved trace message format for Viper, we pursued three goals:

- following existing standards;
- wide applicability across computing environments;
- minimising overhead (i.e. trace message size and format related processing).

At the time of writing, the little work that had been done and published on trace message formatting was essentially captured in the format that was adopted for ParaGraph. This visualisation tool was specifically designed to work with the Portable Instrumented Communication Library (PICL), a packaged set of communication primitives, also developed at MIT. The PICL trace format is expressive and general enough to allow any paradigm to use it, yet its numerical encoding scheme ensures relatively modest storage requirements. The use of the ASCII storage format ensures portability across platforms.

Table 5 lists the data field structure of a PICL formatted trace message.

field	type	description
record type	enumerate	indicates the type of info (start/end event, statistics etc.)
event type	enumerate	Mona Lisa event type to which the trace message relates
time stamp	floating point	processor clock value at time of trace message generation.
source	integer	id. nr. of the module that generated the trace message
# datafields	integer	number of data fields that follow (event type dependent)
data field format	enumerate	For Viper we use the same format for all data fields: integer
first data field : sequence nr	integer	each module enumerates all the trace messages it generates, starting at 1. This number is stored as the first data field

TABLE 5. Description of the PICL trace message structure as adopted for Viper.

field	type	description
other (optional) data fields:		
partner	integer	id. nr. of the partner module, involved in sending/receiving
variable	integer	id. nr. of the global variable to which the message relates
offset & size	integer vector	vector specification of variable slice (for instance sub-array)

TABLE 5. Description of the PICL trace message structure as adopted for Viper.

Figure 35 shows how the original trace message format for Viper maps to the final PICL format:

```
12100 M_EXPOSE INSTANT 2 -1 3 (1,1) (5,5)
```
instant exposure of variable 3, segment [1,5] by [1,5], in module 2, at time 12100

```
1 13 12100 2 6 22 5 3 1 1 5 5
```
in addition to original format: record type 1, event type 13, sequence number 5.

FIGURE 35. Example of a trace message in original format (upper) and PICL format (lower).

5.4.2 Implementation of the on-line connection

The hardware and operating system details of the interface between a parallel processing architecture and the controlling console or work station are usually proprietary and vary substantially across different systems. The challenge with building an on-line (i.e. real-time) connection between Viper and the parallel program therefore lies in making the implementation sufficiently general by abstracting from such details. Object-oriented design is particularly suited for this, as we will demonstrate.

In our design we aim to provide a UNIX socket connection, which is not only a standard mechanism in distributed computing, but also maps very well to the Chorus/Mix operating system which provides such a socket implementation. It should be noted however that the design is not reliant on the presence of a UNIX socket communication structure; for example, it applies equally well to a Windows socket environment.

Our starting point for the design discussion is the class TramSource from Figure 22, shown now in more detail in Figure 36. This class has an attribute EXclock (Execution clock) that contains the highest time stamp read so far. There is also a reference to the tram it last returned from get_tram(). As we can have two different types of input (file or socket), TramSource is specialized into two different subclasses, TramFile and TramSocket. A TramFile is created for a specific file name, and a TramSocket reads from a specific socket-id, so these are modelled as class attributes.

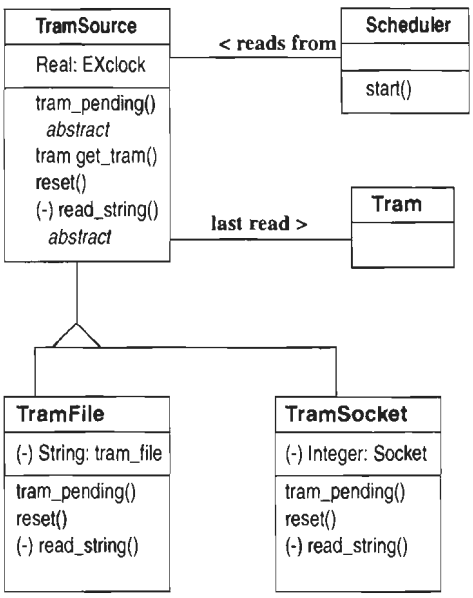

FIGURE 36. Object model for the TramSource class

The implementation of operation get_tram() is not really different for both cases, only the way ASCII characters are retrieved is different. Therefore, get_tram() is defined at the level of the TramSource class, using a private operation read_string(). This operation is not implemented at this level (abstract), but only at the subclass level (TramFile and TramSocket). The operation tram_pending() needs to be treated the same way, as the implementation is also input device dependent. The operation reset() has both device specific aspects and general aspects; this operation is therefore implemented at both the super and the subclass. This completes the modelling of the class TramSource and implementation is a straightforward programming exercise.

5.4.3 Implementation of Viper's trace file export facility

The aim of the trace file export facility is to allow the use of Viper as a tram pre-processor for other analysis tools such as ParaGraph, that might benefit from Viper's tram ordering capabilities. The trace file exported might be a simple re-ordering of the input trace messages, or a sub-set, or it might be a completely different set of trace messages.

The design of it is essentially based on an extension of the capabilities of the Tram class. An operation output() is called at the moment a Tram object is taken out of the run. This operation writes (to a trace file) zero, one or more trace messages, depending on the mapping that is defined between the input trace messages, and the output trace messages. This mapping is completely localised within the Tram class definition.

As an example, we describe below how we could produce a ParaGraph trace file from a Mona Lisa trace file. The transformation process of input trace messages to output trace messages consists of two main parts:

- Remove the details of the Mona Lisa paradigm. Translate Mona Lisa primitives to elementary send/receive operations.
- Add implementation details. Explicitly model a Mona Lisa module as a supervisor process and an application process. Correctly show the scheduling out of the application process as the supervisor process becomes active (both processes run on a single node and in the absence of time slicing, only one can be active at any point in time).

The transformation would then be done in two steps:

- Viper first processes a Mona Lisa trace file, it translates all trace information to PVM-like primitives, and doubles the number of nodes to introduce the supervisor processes. Remember, in ParaGraph 1 process corresponds to 1 processor/node.
- Viper then processes the newly produced trace file with the PVM view on the computation. This time, the scheduling is corrected (idle periods are introduced in the application process), and the PVM-style trace messages are translated to the ParaGraph format which is in effect a subset of the PICL standard.

The two-phased approach simplifies the translation logic that has to be implemented as part of the output() operation of each Tram subclass.

Aiming at flexibility of Viper's design to allow such close integration to other tools was part of the design effort. However, the implementation of the ParaGraph transformation proposed above was out of scope. Therefore, actual examples cannot be shown here.

This section concludes the description of the third prototype. Viper is now in a state where it can be applied and evaluated. This will be the topic of the next chapter.

Chapter 6

Application of Viper to high energy physics image reconstruction software

In the previous chapter we have discussed the development of Viper for Mona Lisa. In this chapter we will report on the first use of the Viper tool to support and evaluate the transformation of a sequential high energy physics application called CPREAD into a parallel Mona Lisa program.

In the CPREAD case study we have investigated the parallelisation of an *existing* high energy physics event image reconstruction software package in terms of costs (effort, computing resource requirements), benefits (performance increase) and the feasibility of a systematic parallelisation approach. The following sections start with the motivations for doing this work. The reader is then given an informal introduction to some relevant aspects of the CPLEAR experiment for which CPREAD has been developed, followed by an overview of the structure of CPREAD and of its potential for parallelism. Subsequently we discuss the required implementation effort, focusing on code decomposition and data organisation. These issues are illustrated by the implementation of parallel track fitting within CPREAD. Finally, we aim to extrapolate some guidelines facilitating parallel implementations for future software development in high energy physics. The work as presented in this chapter has been presented at the Real Time '95 Conference in Lansing, Michigan USA, and subsequently published [SF95].

6.1 The CPREAD case study background

One of the deliverables of the GP-MIMD project has been a document reporting on the investigation of the parallelisation of *existing* high energy physics analysis software. Various ways to exploit parallel platforms for this type of HEP applications have been looked at in this study. The investigation has covered the use of different paradigms: message passing, the Mona Lisa paradigm and a novel paradigm based on database transactions, see [Arg98]. Each paradigm has been used on a specific hardware platform. A detailed overview of the work carried out can be found in [Dob95].

The Viper project's contribution to the GP-MIMD deliverable has pursued parallelisation of CPREAD in combination with Transputers, Chorus and Mona Lisa, by *partitioning* the application into smaller modules to be executed in parallel. This means we exclude the traditional event farming, as this has already been covered in [War95].

Our interests in doing this study are twofold; firstly, to see if the promises that have been made about Mona Lisa and Viper within the context of parallelism in high energy

physics can be fulfilled. Secondly, we want to determine whether significant parallelisation using partitioning of CPREAD at reasonable cost (i.e. limited amount of code rewriting) is at all possible given the complexity of this type of software, and if so, what the limits in performance improvement are. It is to be expected that the effort constraint certainly limits the performance increase.

The significance of this study to the high energy physics community lies in the contribution that it provides to an ongoing discussion on a highly controversial issue, namely the extent to which sequential legacy software can benefit from parallel computing. Although event farming is now widely recognised as a highly effective technique for boosting performance, opinions on the effort/benefit ratio of other types of parallelism based on partitioning of the sequential application (as opposed to replication) are still divided. That there is cause for doubt is already partly demonstrated by the duration of this study; it has taken approx. 6 man months to produce the first parallel version! In addition we mention here that several previous abortive attempts by others (such as an attempt to transform the software to a parallel version using Fortran90 constructs) preceded this study.

6.2 The CPLEAR experiment

The CPREAD software package performs the image reconstruction for the CPLEAR experiment: it processes data from the detector, corresponding to a particle collision (an event), to reconstruct the trajectories of the involved particles and to derive other properties of the particle interaction. Figure 37 shows a cross-sectional schematic of the CPLEAR detector, with the graphical representation of a recorded CPLEAR event. We will now describe this event in more detail to show some of the relevant issues in CPLEAR's event image reconstruction.

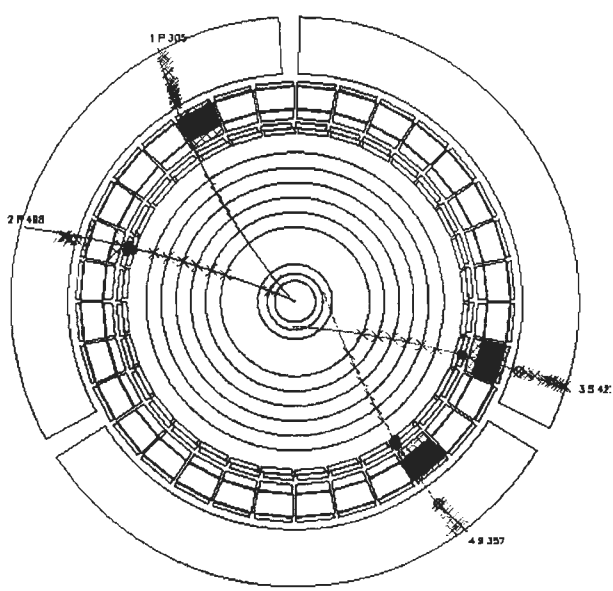

FIGURE 37. A typical 4 track event recording of the CPLEAR experiment.

In the centre of the detector, an anti-proton annihilates with a fixed proton target and produces three particles: a positively charged kaon K^+; a negatively charged pi^-; and a neutral kaon K^0. The first two particles' trajectories are depicted by the curves starting in the centre of the diagram. The v-shaped origin of the two particles is called a *vertex*. The lines that represent the particle trajectories are the result of a curve fitting process, applied to the hits that were registered by the detector (indicated as X's in the diagram). As we are dealing with particle tracks, this fitting is called *track fitting*.

The K^0 particle travels from the centre in the opposite direction, and quickly decays into a pi^+, pi^- particle pair, as indicated by the other two curves in the diagram. These curves extend beyond the particles' origin; crossing the two extrapolated lines gives an accurate indication of the pi^+, pi^- vertex position. The position of this second vertex is a function of the momentum and the lifetime of the kaon particle, which are not constant. Given the fact that the detector cannot trace the neutral kaon, the vertex position therefore has to be derived from extrapolating the charged particle curves. From the distance between the two vertices, the life time of the neutral kaon for this particular event can be determined.

The event type as described above is only one of several types that are relevant to the CPLEAR experiment. By comparing rate of event occurrence and life time of the neutral kaon and its anti-particle, the CPLEAR experiment can measure CP-, T- and CPT-violation phenomena, which is the overall aim of the experiment.

6.3 Structure of the CPREAD program

CPREAD is the off-line event reconstruction package of the CPLEAR experiment. Off-line means CPREAD batch processes tapes with recorded raw event data from the experiment. It is a typical representative of off-line software:

- CPREAD is large - in the order of 200K lines of Fortran.
- It is developed and maintained with the code management tool Patchy [KZ94].
- It uses standard HEP packages such as Zebra [Zeb95] for I/O and dynamic memory management, HBOOK [Hbo95] for histogramming purposes and FFREAD [Ffr95] for application steering.
- It performs typical tasks such as decoding, pattern recognition, track and vertex fitting and event display.

Figure 38 shows a schematic of the event reconstruction steps, the main data structures and the data flows within CPREAD. Two types of data may be distinguished, *event data* and *long term data* (e.g. calibration constants). The latter is constant throughout a sequence of events (called a *run*). The event data only exists within a single event analysis.

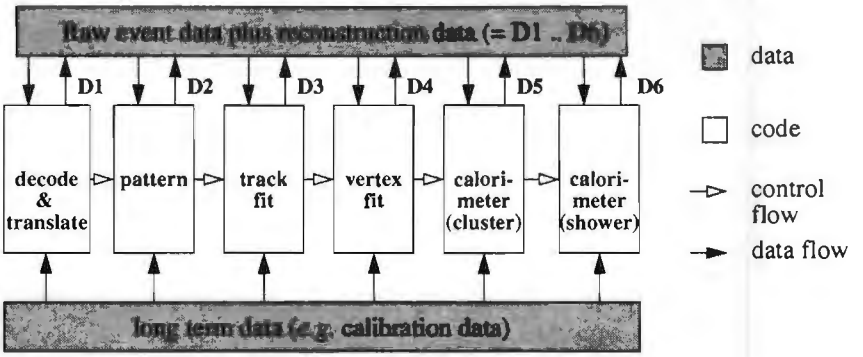

FIGURE 38. A schematic of the event reconstruction steps in CPREAD.

Each reconstruction step may access both the long term data and event data to produce new, additional event data, as depicted by the data flow and data dependencies in Figures 38 and 39. For instance, the data dependency between D3 and D2 indicates that the track fitting accesses D2 in order to produce D3. All data depicted in Figure 39 are event data.

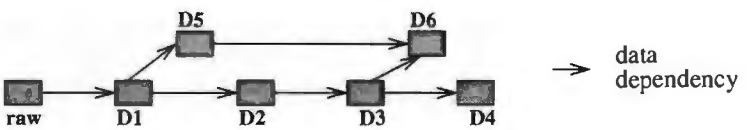

FIGURE 39. Data dependencies among event data banks in CPREAD

The first reconstruction step, *decoding*, involves the translation of detector read-out data into units with meaningful dimensions, such as energy (eV), vector co-ordinates etc. Then, *pattern recognition* is performed on the hit patterns, to find candidate track segments. These are then subjected to kinematic constraints etc. in the *track fitting* process. Once the vertices are determined, they are also subjected to constraints in the *vertex fitting* step. Once these steps have all been performed successfully, the outer sub-detector read-out (the calorimeter part) is analysed for energy showers in two steps: first *clustering* is performed, then *shower analysis*.

In each reconstruction step the event analysis may be aborted, for example because some event data appears to be missing, inaccurate or corrupted, or because the event is not one of the relevant types. In such cases, the event is said to be *rejected*. Of all event data that is generated in the CPLEAR experiment, approx. 80% is rejected.

The sequential order in which the event reconstruction steps are executed is based on data dependencies and computing optimization (i.e. aiming for minimal average event analysis time). The computing optimization is a function of the *rejection power* of each reconstruction step, defined as the rejection percentage per second of step analysis time. Computing optimization is achieved by executing reconstruction steps in decreasing order of rejection power where data dependencies permit. This is the reason why the calorimeter clustering step is at the back of the chain: this computationally intensive step can be done much earlier as indicated by the data dependencies, but it

does not have enough rejection power. Doing this step earlier would therefore result in wasting more computing on events that are rejected in later stages of the analysis.

6.4 Investigation of potential parallelism

The introduction of parallelism in event image reconstruction software can be pursued in two directions, as discussed in Section 2.2 : improving event throughput (i.e. the number of events processed per time unit) and improving event latency (i.e. the time to solution for a single event analysis).

The aim here is to introduce parallelism at low cost. This means aiming for a parallel version of CPREAD resembling the sequential code as much as possible, minimising the effort involved in code rewriting. Therefore the program structure (in terms of reconstruction steps and data structures) is to be left intact as much as possible, using it as a guideline for identifying tasks and partitioning the program into modules. With this strategy in mind, three (non-exclusive) ways of parallelisation can be considered:

- Reconstruction steps that have no data dependency between them are run in parallel for the same event. In CPREAD we can execute the analysis on charged tracks in parallel with the calorimeter analysis, as suggested by Figure 39. This is algorithmic parallelism within an event, and will be referred to as *task parallelism.*

  Task parallelism reduces the event latency and increases the event throughput compared to the sequential analysis. Also, additional rejection paths may be created and exploited if the step running in parallel has significant filtering capabilities, further improving latency and throughput. A negative effect is that running a step in parallel with the step normally succeeding it, implies that the latter now has to deal with more events due to less filtering, making the latency of the step an issue.

- Reconstruction steps that are data interdependent can still work in parallel on different events using event pipelining. For example, when pattern recognition has processed an event, the track fitting starts to work on it, but at the same time the pattern recognition can accept a new event. This is *pipelining*, an example of algorithmic parallelism working on *different* events.

  Pipeline parallelism does not reduce the latency of the event reconstruction, but does increase the event throughput compared to the sequential analysis.

- Parallelism is introduced within a reconstruction step by replicating an individual step and partitioning the data structure on which analysis has to be performed, so that the overall structure can be processed in parallel. The simplest example is loop unrolling. In this context, *data parallelism* will solely refer to parallelism within an event, excluding the well known parallelism applied at the event level, event farming.

  There is a high degree of inherent data parallelism in HEP analysis code, given the repetitive nature of the work in each reconstruction step. For instance, for every track in an event a track fit is performed and these track fits do not depend on each other. Data parallelism reduces latency and increases throughput. An additional positive aspect of this type of parallelism applied to a specific reconstruction step is its local impact on the program structure.

All three approaches reduce the memory requirements per processor. Both algorithmic event parallelism and data parallelism can affect the rejection power of a reconstruction step, so a re-ordering of independent steps after parallelisation may be beneficial. For

example, should performance improvements predominantly apply to the calorimeter clustering step (currently performed at the end of the reconstruction), then in theory the rejection power of this step could increase sufficiently to justify moving this step forward in the reconstruction process.

The choices concerning the amount and type of parallelism to introduce are based on the effort required and the benefits obtainable. The amount of effort is measurable by the amount of code changes, while the benefits are understood in terms of the improvements to event latency and throughput. The reduction in memory requirement is not considered to be a strategic benefit given the low cost of memory per Mbyte.

As a first step in quantifying the cost and benefit parameters, a profiling tool is used to measure the amount of computing time spent in each step. Table 6, column 2 shows the results for a representative CPLEAR batch processing job. For example, it can be seen that 19% of the execution time is spent on pattern recognition. In total, the reconstruction steps account for 77.4% of the execution time. The remaining time (22.6%) is mainly spent on i/o. Included in column 3 are the *cumulative* rejection percentages at the end of each step. Track fitting for instance rejects 3% of all events processed. Finally, column 4 shows the profiling for events that go through all steps; the percentages therefore indicate the relative contribution of each step to the so-called maximum latency of a single event analysis.

TABLE 6. Profiling of CPREAD using IBM 3270, event sample size ~1200.

step	% execution time	% rejection (cumulative)	% relative latency
decoding	11	0	8.0
pattern	19	34	13.9
track fit	15	37	15.9
vertex fit	27	70	30.3
calorimeter	5.4	79	17.7
total	**77.4**	**79**	**85.8**
others	**22.6**	**0**	**14.2**

The distinction between column 2 and 4 in Table 6 is important when defining a parallelisation strategy. If for example a choice would have to be made in halving the execution time of the decoding step or the calorimeter step, it would depend on which alternative would be more beneficial: an increase in event throughput can best be established by parallelising the decoding as in column 2 its contribution is twice the contribution of the calorimeter. However, a parallel calorimeter analysis is more effective for decreasing the *maximum* event analysis time as shown in column 4.

6.4.1 Applicability of algorithmic and data parallelism

At the beginning of Section 6.4 three parallelisation techniques have been introduced. These techniques are not all equally applicable with CPREAD. For instance, the strong data dependencies between the individual steps leave little scope for reducing the latency with algorithmic parallelism: the only option (at the granularity level of reconstruction steps) is running the calorimeter analysis in parallel with the rest.

The event throughput can be addressed by pipelining the steps. The prime advantage here is the cost effective introduction of additional processors while keeping the overall memory requirement of the program constant. But the creation of different code modules requires work, proportional to the number of modules, whereas the amount of work involving the implementation of data parallelism is independent of the number of module copies working together on the same data structure.

Within the processing of an event, data parallelism can only be identified *within* a reconstruction step. This is due to the fact that most rejection decisions are of a global nature and are taken at the start or end of a reconstruction step, subjecting the whole of the event data to various constraints. At the end of the track fitting for instance, an event may be rejected if only three out of four charged tracks have been fitted properly. As such, rejection criteria impose synchronisation points in a parallel implementation.

The cost/benefit estimates for data parallelism for a single event are summarised in Table 7. The numbers in column 2 are estimates of the speed-up that may be obtained by applying data parallelism and are based on the task being performed in terms of physics. For example, on average four track fits need to be performed per event; fitting the tracks in parallel should naively reduce the latency of this step by a factor four if we assume linear speed-up.

The percentual amount of code change in column 3 that is required to obtain the speed-up is a first order estimate. Further precision requires a detailed study of the application structure. The size of each code module in column 4 is information that was actually obtained at a later stage in the study, but it is presented here to allow translation of the percentual amount of code change to absolute numbers.

TABLE 7. Cost/benefit estimates for data parallelism per reconstruction step.

step	proj. speed-up (estimate)	% code change	module size in # of lines
decoding	6	< 5	20000
pattern	< 10	> 25	25000
track fit	4	< 1	17000
vertex fit	4	< 10	25000
calorimeter	< 10	< 15	20000

6.4.2 Projected performance improvements of CPREAD

Table 8 shows the calculated benefits for event processing latency and throughput for the different types of parallelisation strategies (applied separately and a combination of data parallelism and pipelining) at the application level. In the calculations of our projections we have excluded tasks that are not related to the event reconstruction such as i/o. This does not mean we assume this 22.6% of CPU time to be a fixed cost overhead, as this would have serious implications for the scalability of any parallel version according to Amdahl's law. However, addressing the improvement of i/o performance by a sequential optimisation, a parallel i/o implementation or even by making hardware modifications is not within the scope of this thesis.

The projections are based on the figures from Table 6 and 7, not taking into account communication overhead and imperfect load balancing effects. Both throughput and latency figures are expressed using numbers from Table 6, column 4 (i.e. time % of the total analysis time for a single event that is not rejected). Rather than expressing throughput as number of events processed per time unit, we indicate throughput rate by measuring the *cycle time*: the average time period it takes for the program to complete one event and already start on a new event (irrespective of the number of events being worked on in parallel). The word cycle time is borrowed from the manufacturing industry, where the concept is used to describe the througphput of a production line. This facilitates the explanation of the link with Table 6 and 7. The last column shows the number of processors that the implementation requires based on the assumption that each parallel program module runs on a separate processor.

TABLE 8. Calculated performance improvement. Throughput is expressed as 1/cycle time; latency is expressed as percentage of single event analysis time.

program version	throughput	average latency	maximum latency	#processors required
sequential	1/56.8	56.8	85.8	1
task parallelism	1/51.5	51.5	68.1	2
pipelining[a]	1/21.4	84.1	116	5
data parallelism	1/10.7	10.7	16.0	11
combined[a]	1/4.9	19.6	26.0	34

a. obtained through computer model simulation

The figures for the sequential version show a maximum latency of 85.8% of the total single event analysis time, equal to the total of column 4 in Table 6 whereas the average latency is less because an average event does not complete all analysis steps. The throughput of the sequential version is a direct function of this average latency, i.e. 1 event per cycle, where the cycle time equals 56.8% of the maximum single event analysis time. There is only a single processor required to run the sequential version.

The task parallelism strategy only has a moderate improvement over the sequential program. The maximum latency drops by 17.7%, the time required for the calorimeter analysis (see column 4, Table 6) because this analysis is now done in parallel with the other steps. This has a minimal positive effect on both average latency and throughput, especially compared to the amount of extra processing power required. This example shows that task parallelism can improve both latency and throughput, but the benefit is highly dependent on the application. There are two processors required for the task parallelism strategy, with one processor dedicated to the calorimeter analysis.

A pure pipelining strategy of the five steps improves throughput, but the differences in step latencies (pipeline imbalance) has a negative effect on the processor utilisation and the event latency, because consecutive events in the pipeline can block each other. This effect is partly compensated for by the dropping out of events (rejection). The figures in Table 8 are obtained via model simulation. Pipelining requires five processors, one for each step.

The data parallelism strategy addresses the latency of the individual reconstruction steps and thereby also the total event throughput. The figures in Table 8 are based on the speed-up figures from Table 7 applied to each step, showing a relatively high performance improvement. Full data parallelism requires an extra 10 processors in a farm

configuration, in order to speed up the pattern recognition and calorimeter analysis by a factor close to 10.

The combined strategy of data parallelism and pipelining gives maximum throughput, although the latency improvement is not as good as in the case of pure data parallelism. This is because of the blocking that is introduced by the unbalanced pipeline; the pace with which events go through the pipeline is determined by the slowest step. It should be noted that the application of data parallelism to the stages in the pipeline influences the unbalanced nature. It is in fact a well known technique for reducing the idle time that is introduced by an unbalanced pipeline: by improving the latency of the pipeline bottleneck with data parallelism, the work load is spread more evenly over the pipeline, resulting in a more efficient use of processors.

Table 9 contains the same data as Table 8, but presents it in a normalised form. Both throughput and latency improvements are expressed in speed-up factors, thereby allowing a direct comparison between performance improvement and computing power (number of processors) required. In addition, Table 9 shows the average processor utilisation, calculated as:

utilisation % = (throughput / sequential throughput)*(1 / #processors required)*100 %.

This calculation is a simple re-distribution of a fixed workload of N events taking time T to be processed by the sequential version: on a parallel version the same workload is done in a different time frame (the first factor in the calculation), and distributed over multiple processors (the second factor in the calculation).

TABLE 9. Calculated performance improvement, normalised figures.

version	throughput	average latency	maximum latency	#processors required	processor utilisation (%)
sequential	1.0	1.0	1.0	1	100
task parallelism	1.1	1.1	1.3	2	55
pipelining[a]	2.7	0.68	0.74	5	53
data parallelism	5.3	5.3	5.4	11	48
combined[a]	11.6	2.9	3.3	34	34

a. obtained through simulation

The first conclusion that we can draw from the table above is that the scalability of our parallelisation approach is limited. This is of course a direct consequence of our choice to restrict the amount of code rewriting, and to refrain ourselves from redesigning the algorithms at all. In the following sections we investigate whether the modest targets of Table 9 are indeed feasible with moderate effort.

6.5 Investigation of implementation

In the previous section the scope for parallelism in CPREAD has been investigated. In this section, we discuss the approach that should be taken to implement parallelism, and what obstacles can be encountered in doing so.

6.5.1 Development approach

Current HEP software development and maintenance makes use of code management tools such as Patchy [KZ94] and its successor CMZ [Cod94]. The code modifications necessary for a parallel version need to be implemented using the same tools, allowing the sequential and the parallel versions to co-exist. This corresponds to the real-life situation in which different members (i.e. institutes) of an international HEP collaboration may require a sequential or parallel version, depending on the hardware available to them. It also facilitates the necessary integration of ongoing code developments, thus ensuring that any parallel version stays as up to date as the sequential version. HEP code is continuously evolving!

Patchy and CMZ provide version management for a large library of Fortran routines, referred to as a *code pool*. A program is constructed from this code pool by executing a user-written extraction script which selects the appropriate routines and combines them into a program. Code management is facilitated by the decomposition of the code pool into a hierarchical structure. For this study, CMZ was adopted.

The Mona Lisa parallel programming paradigm is used as an additional structuring tool, as the development of a parallel application requires additional structuring to that provided by a sequential programming language such as Fortran77. As explained in Section 2.3 , Mona Lisa is shaped as a language extension, offering a set of primitives which allow the exchange of data among modules that run in parallel and together make up the parallel program. Mona Lisa provides development support in several ways:

- In a single file, declaration statements indicate the start and end of the individual code modules that together make up the parallel program. A front-end translator processes this file and automatically produces a (separately compilable) Fortran file for each module, thus reducing user file management.
- Data exchange between modules occurs through user designated variables. The Mona Lisa front-end translator, while processing the single file, performs cross-module type checking on the communication of the values of these variables, trapping potential programming errors before compile time.
- The Viper visualisation tool, an integral part of the Mona Lisa development environment, can be used in the process of analysing, improving and debugging a parallel program at the level of the interacting modules. It can assist for instance in investigating the program behaviour as a function of the number of processors that we allocate.
- Every Mona Lisa program has a built-in component, the program manager. The parallel program is executed by starting up the program manager. It then takes care of loading the modules on processors, starting the application, and unloading the modules once the application has finished. It also has the capability of run-time deadlock detection.

To summarise, CMZ is used for static and syntactic code management, while Mona Lisa provides dynamic and semantic development support. Automatic executable code generation from a single source file (via code extraction scripts and the Mona Lisa front-end translator) and the existence of a ported sequential version (see below) ensure systematic code and result comparison.

6.5.2 Porting CPREAD

In general, the parallel platform will involve different hardware with different processing characteristics (e.g. processor speed, floating point arithmetic precision). The sequential version of the application should first be ported to this platform, thus decoupling the issues related to application porting from the parallelisation issues. The existence of a sequential version on the target platform allows a better comparison of performance improvement and is needed to verify that the parallel version produces the same results as the sequential version.

The porting of CPREAD to the target platform has been combined with a study of the existing code modularity. To this end the code has been ported incrementally to the new platform. The first ported version only performs program initialisation and termination. Each successive version includes an additional reconstruction step. An additional advantage of this approach is that it avoids putting effort into porting code that is not used in this case study, for example the code needed for event displaying.

The porting of the code has involved not only the foreseen problems (different i/o handling, floating point arithmetic precision etc.) due to a new hardware platform, which in this case study was a Transputer network. Some additional code changes had to be made caused by compiler differences and limited memory availability, forcing us to work on eliminating unneeded code for both sequential and parallel versions. We will not elaborate on these issues, as they do not influence the essence of the problem. It is important to mention however that this particular phase can generally be very time consuming and is usually underestimated.

6.5.3 Code decomposition

For any of the proposed parallelisation approaches to be effective and efficient in terms of code size per processor, the code needs to be decomposable into the modules that correspond to the reconstruction steps. A coarse decomposition of CPREAD already exists, based on the existing code pool hierarchy and associated code extraction scripts. However, a finer grain extraction procedure is required. The following two points prohibit a systematic change of the existing extraction scripts to obtain the required level of extraction and necessitate new scripts:

- The existing hierarchical code structure only roughly coincides with the decomposition into reconstruction steps. The routines that make up one reconstruction step are not grouped together, but exist at different places within the hierarchy.
- The existing scripts, for reasons of simplicity, extract much code regardless of whether it is used at run-time or not, in a coarse grain extraction process.

Based on the structure of the code at the highest level in the program, an attempt has been made to extract in isolation the program initialisation, termination and reconstruction steps. These modules are all identifiable by an appropriate top-level routine which is called in the main program loop. By recursively selecting the routines that are called (a procedure which was largely automated), the set of routines constituting a reconstruction step can be constructed. However, this strategy of creating code modules fails due to the fact that at a lower level the identified modules have too many inter dependencies, although there appears to be a clean separation between individual reconstruction steps at a higher level. Trying to construct the decode module with this approach

we end up with a code segment that is almost half the size of the complete program, whereas this module covers at most 10% of the total functionality of the program!

TABLE 10. CPREAD code decomposition.

code module	module size in # of lines
init & end	15000
main event loop	20000
decoding	20000
pattern	25000
track fit	17000
vertex fit	25000
calorimeter	20000
other (display etc.)	88000
total	**230000**

The adopted procedure to create code modules is a combination of the procedure mentioned above and the elimination of calls to routines that are deemed not to be relevant to the code module being extracted. This requires an understanding of the purpose of the individual routines. The resulting code decomposition is summarized above. The combined process of code porting and decomposition has taken around 2 man months to complete. A significant part of this period had to be spent on rewriting the extraction scripts for our needs. This is a CPREAD specific process, and therefore a significant learning curve for this process does not exist (i.e., for an application other than CPREAD but similar in size and complexity, a similar investment would have to be made). Our main conclusion therefore is that it is more than worthwhile to take into account the code decomposition needed for a parallel implementation when setting up the code pool hierarchy structure, to avoid a "hierarchy redesign" at a later stage.

6.5.4 Data organisation

The main data structures depicted in Figure 39 are all implemented as globally accessible so-called *Zebra* data structures, upon which the physics analysis code operates. A Zebra data structure is a type of dynamic data structure based on variable sized arrays called *banks*, that are organised in tree configurations. Although an individual bank is contiguously stored in memory, the overall tree structure may be dispersed. The Zebra package that provides the functionality to create and manage Zebra structures was developed within CERN as an extension to the poor data structuring capabilities of Fortran77 and its predecessors.

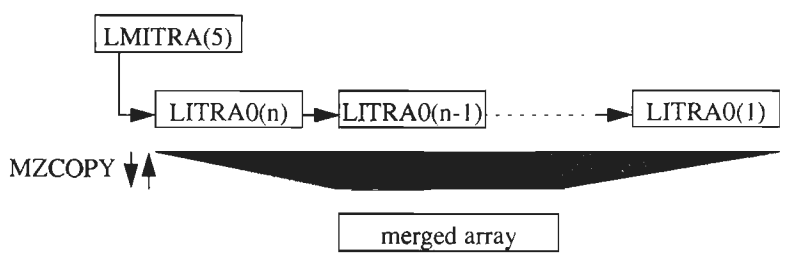

FIGURE 40. An example of a Zebra structure in CPREAD: the organisation of track data. The routine MZCOPY can be used to reformat the data into a contiguous array or vice-versa.

The figure above demonstrates the principle for CPREAD. The track data are stored in a linked list of *n* Zebra banks (where for example *n* is the number of tracks in the event). Both direct and indirect data access are used in CPREAD. The static array LITRA0 provides direct links to the individual banks. However, because the number of banks used can vary, it is better to access the so-called mother bank first via the link LMITRA(5). This bank contains several flags and parameters that relate to the track data as a whole, including the number of banks currently defined. With the Zebra routines LUP(), LDOWN() and LNEXT() it is then possible to 'walk' from the mother bank down to bank LITRA0(1).

The standard mechanism of data communication on MIMD parallel computers is via message passing. The message passing library supports the communication of blocks of contiguous data, thus supporting the communication of individual Zebra banks. For the communication of large data structures, consisting of several banks, the communication overhead becomes excessive as multiple communications are required. Zebra provides a set of calls (MZCOPY calls, see figure) that implement the selective copying of (partial) Zebra structures to an array (and vice versa), thereby establishing a contiguous data block that can be communicated in one message passing call. However, the additional overhead introduced by reformatting the data must be weighed against that of multiple communications.

6.6 Parallel track fitting

In the previous sections scope for parallelism in CPREAD has been discussed, followed by a discussion on implementation issues. In this section we apply this to an example of data parallelism: parallelisation of the track fitting. Due to its minimal impact on the algorithmic structure, this example allows us to focus on the development approach, code partitioning and data flow implementation with minimal application specific code rewriting.

The data sets representing the individual tracks are self-contained (mutually independent), and a fit is performed to each data set within a loop construct. Data parallelism is therefore introduced by simply unrolling this loop.

The first step towards a parallel version is to decide how many modules to implement and what tasks they must perform. Subsequently, the data flows between the modules are identified, and finally, using the task assignment and the data flows, the individual modules are constructed.

Two different modules are developed: one which performs the full CPLEAR event reconstruction except for the track fitting (called the *master*) and a second module which performs a single track fit (called a *slave*). A representative CPLEAR event sample contains events with 2 to 6 tracks, approximately 80% being 4 tracks events. Therefore we implement a Mona Lisa program consisting of 1 master and 4 slaves.

The principal data flow consists of *event data*: track banks, communicated from master to slave, and the resulting fit banks, communicated from the slave to the master. Both the track banks and fit banks are around 1 Kbytes in size and stored in a linked list structure in the master. In addition, track fitting uses calibration data (introduced in Section 6.3 as *long term data*). This data is independent of the master or slave and as its name suggests does not have to be communicated per event. For these reasons it is replicated over all modules. Updates of calibration data are triggered by a change in the

run number in the master. This run number is therefore also communicated from the master to the slaves.

The construction of the master and slave code modules is relatively straightforward: the modules can be composed from the individual components obtained during code decomposition. The master code is identical to the sequential code without the track fitting step. The slave code consists of program initialisation/termination, the main event loop, the track fitting step and the routines necessary for updating the calibration data. Some additional code in both the master and slave is added to implement the data flows.

6.6.1 Running the parallel track fitting program

The first implementation of the parallel track fitting version has taken place on a T805 Transputer network, using the Mona Lisa paradigm to implement the parallel features. The Chorus distributed operating system supports the current (prototype) Mona Lisa implementation and provides (amongst other things) point-to-point message passing functionality across T805 Transputer nodes.

Initially we were unable to run the program for more than a few events, after which the program would 'hang'. Taking one of the observed 'deadlock' scenarios visualised with Viper as our starting point for an investigation, we discovered the locking up was due to a bug in the Mona Lisa communication library.

With the corrected Mona Lisa library, we have run the parallel program for benchmarking on a set of 10 4-track events; the measured speed-up that we have observed was a factor 1.5 instead of the naive estimate of 4 (see Section 6.4.1). Using Viper to analyse this discrepancy, we arrive at Figure 41 where we focus on a particular event.

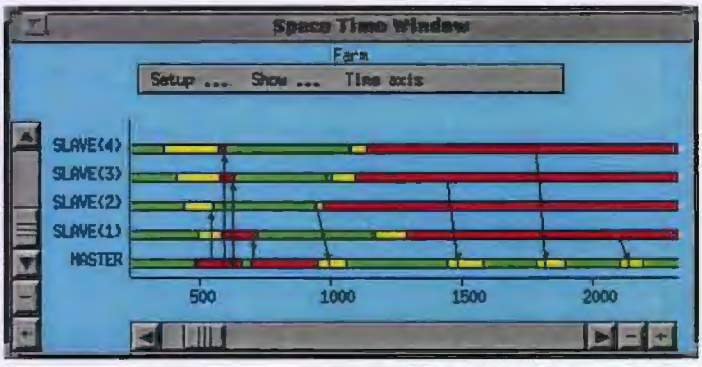

FIGURE 41. Viper's space time view on the track fitting of one event

Although we can use the figure for a qualitative analysis, it is good to bear in mind that the introduction of additional tracing has a disturbing effect on the computation; the speed-up factor for instance for the traced parallel version is further reduced, to just below 1 - in effect we then have a parallel program that is slower than the original sequential version!

From the figure we observe the following:

- The program is, as far as module interaction is concerned, behaving correctly: each slave receives track data (the arrow going upwards), processes the data, and the result is communicated back to the master (the arrow going downwards).
- There is fairly good load balancing, judging by the similar size of the green part of each slave's time bar.
- The track fits are performed largely simultaneously, and not staggered. Staggering as a result of the communication pattern between the master and the slaves during job distribution would have been a possible explanation for the lower speed-up factor, see Figure 42.

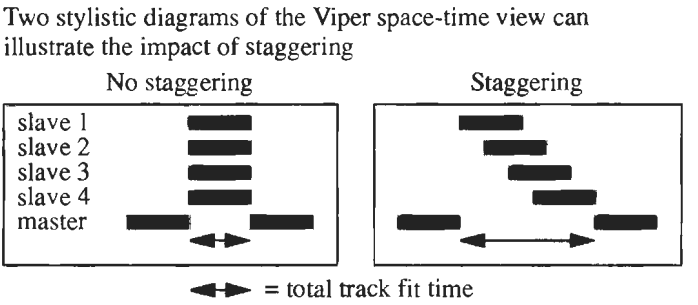

FIGURE 42. The effect of staggered track fitting on total fit time and hence speed-up.

The most important observation however is the clear cause for the disappointing speed-up: the result collection by the master. Comparing the job distribution overhead with the result collection, we see a striking difference: only long after slave 1 has completed his job, is his fit data collected by the master. Why is this?

To answer this, we have to do some communication analysis. We recall Figure 9 from Section 3.2.2, where the implementation of an inglb primitive has been explained for a distributed memory machine (as is the Transputer network that we use) in terms of required point-to-point communications.

If we count the number of point-to-point send's and receives that are required for each Mona Lisa primitive call, we get:

- no send/receive's when using exposeglb or hideglb.
- 2 send's and 2 receive's when using rdglb, inglb or wrglb.

The situation for an inrglb primitive is more complicated, and is illustrated below in Figure 43. We take as example the call Slave.inrglb(Result,MyResult,SlaveId) that a Master module uses to retrieve data from the first Slave that has finished his job. In this particular example, it is Slave(2) that will actually deliver the data. The actions of Slave(3) represent what may take place for each of the other Slave(i) modules.

The protocol starts with the Master sending a request to all Slave modules involved. A number of Slave modules may have a result ready and exposed, and reply by sending it. The master accepts the first result that it receives, and indicates this to the Slave

module involved (i.e. Slave(2)) by sending it an Acknowledge. Slave(2) can now update the state of the Result variable and set it to hidden.

Immediately after sending the acknowledge, the Master sends Cancel messages to each of the other Slave modules to try and prevent them from sending a reply that it has to ignore anyway. After updating the value of MyResult, the Master's Supervisor thread is finished and the application process wakes up.

Some of the Slave modules may have sent a reply before they had time to receive the Cancel message (such as Slave(3) as indicated in the diagram). These replies are received by the Master Supervisor, and a Cancel message is sent in reply.

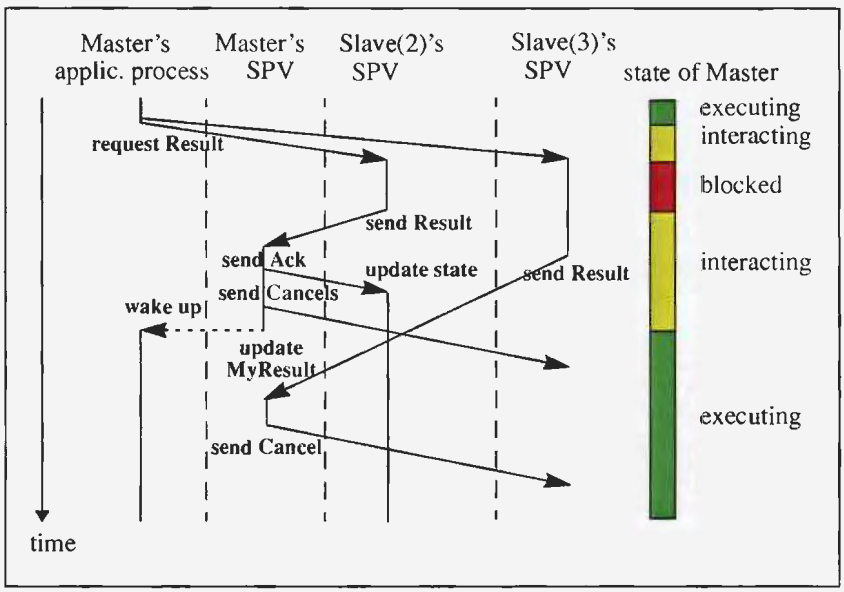

FIGURE 43. Analysis of the call Slave.inrglb(Result,MyResult,SlaveId) by module Master.

In the current implementation of Mona Lisa, the Cancel message is sent to all Slave modules through a broadcast mechanism. Performance benchmarking has shown that broadcasting to N modules under Chorus v.1 has similar performance characteristics as sending to N modules consecutively. Therefore, if we count the communication effort for a general case of N Slave modules, assuming that N/2 modules will try to reply, we obtain a total of 3*N send's and 3*N receive's. When tracing is active, this increases to 9*N send's and 3*N receive's.

Applying this knowledge to our parallel program, we conclude that the result collection phase involves 4 inrglb's and the overlapping start-up of 4 inglb calls from the Slaves for new jobs, totalling to 52 send's and 52 receives (without tracing). The majority of this communication takes place at the Master side, effectively causing a major bottle-neck. On Chorus v.1, the kernel processing time of sending and receiving a single byte amounts roughly to 1 ms. These processing times increase to 2-3 ms. for the cases where 4 Kbytes of track and fit data is communicated. Processing times may increase further due to increased context switching, the result of the Master becoming an obvious communications bottle-neck. Summarising we can state that the Chorus run-time

kernel overhead can easily account for the extended period of time of the result collection phase.

Now that we have analysed and explained why the parallel version underperforms, we can work on a solution. We start by replacing the Mona Lisa primitive calls by Chorus point-to-point message passing calls. The result is a speed-up factor of 2.3. This is already quite an improvement, but apparently Chorus is still a heavy burden for the T805 platform.

The Chorus version of the parallel program has a communication structure that allows it to be mapped easily to the T9000 Transputer network using the INMOS toolset, the native message passing library. The improved communications in terms of latency and bandwidth, supported by the native message passing library, result in a speed-up factor 3.0: instead of 137 msec. for sequential track fitting, only 45 msec. are needed in the parallel case.

To investigate the discrepancy between the measured speed-up factor 3 and the theoretical maximum 4[1] we perform a detailed timing analysis by instrumenting the code with timing statements. The results are presented below in Table 11. The maximum track fitting time within an event is measured as 39.97 msec, which is 15% above the average track fitting time. This imperfect load balancing causes the theoretical maximum speed-up factor to drop to 3.5. The Zebra data reformatting operations on both the Master and the Slave side account for another 3.67 msec. The remaining time is spent in communication.

Please note that the communication time does not only relate to the data associated with the longest track fit. As this fit can be any of the 4 tracks, it can take 1, 2, or 3 preceding bank communications before the longest track fit data is sent out by the Master. Therefore, the 2 ms. that is mentioned in the table corresponds to an average of 0.25*(1+2+3+4)=2.25 sends by the Master, 1 receive by the Slave, 1 send by the Slave and 1 receive by the Master.

TABLE 11. T9000 wall clock time spenditure on parallel fit, 1000 evt. sample.

aspect	attributable time in msec.	time in %
algorithm (fitting)	39.97	87.6
communication	2.01	4.4
Zebra manipulations	3.67	8.0
total	**45.65**	**100.0**

An important observation is that the Zebra operations that move the data structures between the Zebra dynamic storage area and the static communication arrays are relatively expensive: 8.0 percent of the total time, compared to 4.4 percent for the communication. This emphasises our point made earlier on the fine balance that exists between data reformatting with a single communication or choosing multiple communications without data reformatting.

1. The term theoretical maximum may be misleading; it is not the maximum speed-up achievable at all cost, but the maximum speed-up obtainable given the decision to keep the track fitting code for an individual track fit intact.

With this section we conclude our implementation example of parallelism in CPREAD. In the following sections we summarise our findings and topics for further research. We start with aspects relevant for high energy physics and subsequently discuss the learning points for Mona Lisa and Viper.

6.7 Discussion on high energy physics aspects

The tasks performed by typical HEP applications have substantial potential for parallel execution. To achieve reduced event latency, finer grain parallelism compared to event farming has to be introduced into the application. Parallelisation can reduce the task size and complexity per processor, so resources such as (cache)memory are utilized better and consequently a more cost effective solution is obtained. The disadvantages compared to farming lie in the area of scalability, processor utilisation (load balancing) and difficulty of implementation. We will shortly address each of these issues.

Scalability is not an issue for event farming: the number of events to be processed is large enough, and the ratio of i/o to computation is low enough, to ensure good load balancing, even with a large set of slaves. Scalability need not be an issue for finer grain parallelism either; principles such as locality behaviour of particles that are used in computations on fluid dynamics could equally be applied to high energy physics, thus achieving scalable solutions. However, it does become an issue when we want to impose this parallelism to an *existing* sequential application. The requirement to keep the sequential and parallel version identical as much as possible puts a limit on our scalability.

In this case study we have looked at one example of CPREAD's tasks in particular, track fitting. We have shown that with minimal code changes a speed up factor close to the theoretical maximum - that is, given the decision to keep the track fitting code for an individual track fit intact - can be achieved. The maximum theoretical speed-up is not obtainable due to load imbalance and communication overheads introduced by data parallelism. Indeed, track parallelism is more sensitive to load imbalance than event farming, where it is not an issue. Data parallelism can be equally applied to the other tasks performed in HEP reconstruction. Scope for large scale parallelism of this type exists in future experiments on LHC, where for instance the number of tracks per event increases considerably.

We have seen that communication overheads introduced in the parallelisation process can be reasonably minimised using the data reformatting functionality offered by Zebra. This functionality - the grouping and structuring of data - allows efficient data communication to be implemented for current and future parallel Fortran applications. However, the Zebra data structures should be defined carefully. As banks can be used as the unit of data distribution, independent data sets that can be processed in parallel should not be stored in a single bank whereas interdependent data should. These two points aid in achieving minimal communication and Zebra operation overheads. The track fitting example showed that, although no data reformatting needs to be performed for a single bank, the copying of individual track banks to communication arrays is already costly.

This case study also looked at the code modularity of CPREAD. It has proved very difficult to decompose the application into its component reconstruction steps although it has been developed and maintained by a code management tool. Code management

tools such as Patchy and CMZ can and should be used more efficiently to achieve code modularity, a basic software engineering requirement.

Modularity should also be addressed at the level of the algorithms. Logically independent algorithms should be self contained at the physical code level. The parallel execution of these algorithms is easier if code optimization issues are decoupled from the algorithms. This could lead to a substantial reduction in the amount of code rewriting.

In this case study various tools have been found to be necessary for a structured parallelisation strategy. A profiler is needed to asses the potential for different types of parallelism. During development, a code management tool (CMZ) is needed for classical code management. Finally, a parallel programming paradigm (Mona Lisa) provides the tools for additional structuring, tuning and debugging.

In the future, good programming techniques will be used to allow efficient/effective parallelism. Indeed, one could say that developing parallel software requires careful software development and similar techniques applied to sequential software will improve its quality as well. Parallel programming techniques could be regarded as a tool for quality software development in general.

The use of fine grain parallelism allows a reduction of the (cache)memory requirements per processor. This has been demonstrated in the track fitting example, where the code of one slave amounts to less than 20% of the sequential program code size. The memory savings should be weighed against the disadvantage of suboptimal use of processors that is introduced by fine grain parallelism.

Table 12 summarises our findings regarding the strong and weak points of the various parallelisation strategies, with ratings varying between ++ (very positive), 0 (neutral) and - (negative).

TABLE 12. Parallelisation strategy characteristics

	data parallelism		algorithmic parallelism	
aspect	**event farm**	**track farm**	**pipeline**	**task parallelism**
throughput	++	+	+	-
latency	0	++	-	+
scalability	++	+	-	-
memory req.	-	+	+	+
effort	++	-	+	+

Going through the table, we see that event farming is regarded as the most suitable for increasing event throughput. This is hardly a surprise given the large amounts of events to be analysed, the usually extremely small communication to computation ratio, and last but not least the ease with which it can be implemented. The latter point causes the minus in the category of memory requirement, as the full replication of the application over the computing nodes is quite wasteful in terms of memory.

The track farm variant and the pipeline strategy also address throughput, although to a much lesser extent. The task parallelism alternative is the least favourite for throughput increase: due to changing event rejection patterns this type of parallelisation is likely to incur extra computing on events that would normally have been discarded earlier.

Data parallelism scales up to higher speed-up factors than algorithmic parallelism. The projections in Section 6.4.1 may need some adjustment in the light of the results for the parallel track fitting, but they still show a clear case for data parallelism, especially if the reduction of latency is an issue. The relatively positive characteristics of track farm parallelism compared to algorithmic parallelism do not come for free: it is the most labour intensive parallelisation strategy, noted by the minus in the category effort.

Concluding we can say that no single parallelisation strategy appears to address all five issues of latency, throughput, implementation effort, memory requirement and scalability. The best approach would appear to be a combination of the various parallelisation strategies. Scalability of throughput is best assured by farming out events to workers. Latency, not addressed by event farming, can be addressed by the addition of finer grain strategies. For example, the replacement of a worker by a team of subworkers, implementing a pipeline with data parallelism at each stage.

6.8 Discussion on Viper and Mona Lisa aspects

In the CPREAD case study we have looked at potential speed-up, obtainable under the restriction that the main code rewriting should be restricted to the way the code is organised, and not replacement of fundamental algorithms such as pattern recognition.

To be able to do the type of analysis as presented in this chapter succesfully, the following tools, skill sets and environments can be summarised to be essential:

- knowledge regarding application structure (which modules exist and how are they constructed)
- knowledge regarding application domain (for example, the speed-up factor 4 for track fitting is based on the underlying characteristics of the physics involved)
- a profiler to determine the processing time distribution at module level, per unit of work (in the case of the CPREAD study a unit of work is equal to a single event).
- a software configuration management tool to help with the creation of the modules (in this case we used CMZ).

Last but not least, the use of Viper was instrumental in our analysis of the behaviour of the parallel version of CPREAD. Starting with a visual confirmation that the actual track fitting was indeed properly distributed over the four slaves, the space time window allowed us to identify the nature of the problem of the lack of program speed-up under Mona Lisa. Viper proved easy to use, with all the appropriate navigational aids available to focus on the area of interest in the parallel computation (e.g. the use of go/stop, fast forward until a certain time stamp is reached, scroll bars in the time windows).

Viper could not be used in the later stages of the analysis because the Chorus libraries and T9 toolset are not equipped with PICL trace file generation capabilities. This clearly demonstrates the biggest dependency of successful use of any visualisation tool: access to program execution data via generic and standardised trace message structures. Therefore, this points to a weakness that needs to be addressed across the board in commercial parallel computing, where the lack of such standard infrastructure prohibits us to reap the benefits of advances made in parallel computing science.

The use of Mona Lisa for designing a parallel CPREAD has been satisfactory. The process of parallelising the CPREAD program has been intuitive and straightforward:

- Partition the sequential code in modules
- Variable accesses over module boundaries indicate which variable declarations should be preceded with the word GLOBAL (so that they become Mona Lisa global variables)
- Any references to global variables belonging to other modules are replaced by Mona Lisa read primitives (rdglb, inglb or inrglb).
- When a result value is stored in a global variable that is subsequently needed in another module, an exposeglb primitive call is inserted to make the result available.

However, both the Mona Lisa paradigm and its development environment are still at a level of immaturity. During our study, the Mona Lisa front-end parser had to be extended to parse the CPREAD program, as some parts of it are older than Fortran77. Also, a limit had to be defined on the support of automatic translation of global variable declarations using parameterised dimensions. For instance, a declaration of

```
PARAMETER P=100
GLOBAL INTEGER X(P)
```

is allowed, because the front-end also parses the PARAMETER statements and can therefore substitute P with 100 in the global variable declaration, but the declaration

```
COMMON A(10)
DATA A(1)=100
GLOBAL INTEGER X(A(1))
```

is not recognised by the front-end and results in a parse error.

Also the existing functionality of the Mona Lisa primitives is not to our full satisfaction. Currently, the primitives only allow the reading of fixed-size variable slices: in the call Slave.inglb(Result,MyResult,Offset,Size) we must indicate precisely what subarray (slice) of Result we want to read into MyResult. CPREAD's extensive use of Zebra banks for data structuring, including the track and fit data, forced us to make a choice between the following alternatives:

- Communicate a fixed sized array that is always big enough for the banks in question (thus sometimes communicating more data than necessary). This means we can replace the argument Size with a constant.
- have a preceding communication indicate the size of the bank(s) first. This is then used for the Size argument in the following inglb call.

Despite the communication of excess data, the first alternative is the better of the two; separate communication of the bank size has a higher performance penalty. However, what would have been more suitable for instance is the possibility to have a zero value for Size in the inglb call indicate that we want to read whatever part of the array is exposed. Upon return, the value of Size can indicate the size that has been communicated. An additional primitive in the shape of Module.is_exposed(var) that checks if a global variable is exposed would also be essential for elegant programming solutions in some cases.

Aside from the functionality of the paradigm, there is the performance issue on a parallel machine with a distributed memory architecture. Inherent to the paradigm, a slight overhead has to be paid for to the use of a supervisor process. Much more severe however is the multitude of messages that are required to implement a basic primitive such as the inrglb. With a worst case of 4*N send's and receive's per primitive call (where N is the number of replicated instances involved), this primitive clearly exhibits non-scalable behaviour. One could of course try to avoid using the inrglb, for instance by doing result collection with wrglb calls, but this does not solve the problem...

The performance issue is aggravated by the Chorus operating system. With the used implementation (Chorus v.1) on T805 Transputers, Chorus message passing operations incur more than a ms. kernel cpu processing overhead, even with very small message sizes. The Chorus functionality that Mona Lisa uses could easily be replaced by native Transputer Toolset operations. Doing this for the T9000 platform would make Mona Lisa affordable, at least for medium sized applications with not too large networks: the T9000 has a communication latency from T9 to T9 under 10 microsec., with link speeds exceeding 100Mbit/s.

Another performance related issue is trace message generation. When we analysed the Mona Lisa version of parallel track fitting, we experienced a significant performance degradation of the program execution when tracing was enabled: the track fitting time increased from 149 ms. per track without tracing to 177 msec. with tracing! This is attributable to the interference of the Chorus microkernels, that together have to process another 88 trace messages per event that are immediately communicated to the Mona Lisa program manager after generation.

To reduce the perturbation of parallel program execution resulting from trace message communications, we have implemented two new Mona Lisa primitives that allow us to temporarily buffer the trace messages on the node where they have been generated. The generation of the trace messages hardly takes up any CPU time, and provided there is enough memory to store the trace messages, we can wait with downloading the trace information to file (via the program manager) until the parallel program execution has finished.

The new primitive **trace_buf**(*int size*) is used at the start of a module's execution to allocate an internal trace message buffer of the requested *size*. With the primitive call **tracing**(**true**) trace message generation is then activated, whereas the call **tracing**(**false**) suppresses further trace message generation. This enables us to trace an interesting part of a long program execution without having the need for an extremely large buffer to hold all trace messages generated.

In this case study, the Viper tool has assisted us in analysing and improving the execution of a parallel program with a small number of parallel modules. Now that we have covered the issues that emerged from this (such as the trace buffering), we can focus on increasing the problem size. In the next chapter we discuss the use of Viper in a project where it is used to visualise the interaction of a 1000 network nodes.

Chapter 7

Application of Viper in a study of large switching networks

In the ESPRIT Macramé project P8603 [OMI95][Mar98] studies have been carried out on the behaviour of traffic patterns going through large switching networks. Of particular interest is the appearance of communication bottlenecks (so-called *hot spots*) in different topologies such as mesh, tree and hypercube, and the behaviour of these hot spots over time - how they spread out, how long the effect lasts etc. CERN has an interest in participating in this project because of the applicability of study results in the building of data acquisition systems. HEP experiments such as the ATLAS experiment for LHC are foreseen to make extensive use of switching network technology in their data acquisition systems. In this thesis we will describe the situation of the Macramé project as it was in January 1996, to sketch the context in which Viper made a contribution to the project.

7.1 The Macramé project background

The switching network that was built in the Macramé project consists of more than a thousand traffic sources that send messages to each other through a network of 64 switching devices. The traffic sources and switching devices are interconnected using point-to-point links based on Transputer technology. This network is used as a test bed to study different traffic patterns. Different topologies can be studied by using reconfigurable hardware. In addition, the software that boots and controls the network and drives the data communication patterns can be set up so that only a sub-network, or sub-topology, is used.

Some issues of complexity must be dealt with:

1. Over a thousand devices have to be interconnected, partially by hand-wired cabling, to implement a desired topology. The mapping of a logical topology to a physical network setup is complex due to cabling restrictions, and is error prone. Therefore, a systematic verification process is desirable.
2. A network of this nature is likely to show at least some transient problems such as link failure or device failure. Making sure that all devices have booted without error states and that the network is fully operational before starting to use it for experimental runs is a necessity.
3. The amount of data produced by the network to monitor the traffic patterns is substantial, and necessitates some form of data filtering and further processing in order to facilitate data analysis.

4. There are three separate network descriptions that each use different device identification schemes: the physical network cabling, the sub-network definition in the control software and the logical topology being implemented. These descriptions need to be mapped onto each other in a coherent way.

The picture below of the Macramé network that was produced at the end of the project demonstrates the size and complexity of some of these issues:

FIGURE 44. The Macramé network of 1024 nodes and 64 switches.

Visualisation techniques can help to address the issues mentioned:

- The first two issues, which can be classified as hardware debugging related, can benefit from a graphical representation of the system state where the sound operation of devices and links is represented via the use of colours. We can take note here of the similarity with Viper's animation window where each module of the overall parallel program has a state definition associated with it, which is represented graphically using colours.
- Visualisation techniques facilitate the analysis of large data volumes considerably, provided a suitable graphical mapping can be defined. In the case of Macramé, the data volumes relate particularly to data describing the development of hot spots over time. Viper's time view similarly shows inter-module communication and module state patterns over time.
- The handling of different network representations such as logical topology is a key support feature of many visualisation tools. Viper's capabilities in this area were still quite rudimentary after three prototypes, but could easily be extended.

The decision was therefore made to investigate the suitability of Viper's visualisation capabilities to address some of the issues mentioned. From the project description given so far we can distil some preliminary new requirements for Viper on which its usability for Macramé depends:

- The ability to support different module types (e.g. nodes and switches) that may have different state behaviour.
- The ability to graphically represent different modules in different ways, so that for example a node and a switch can have a different appearance on the screen.
- The ability of Viper to display a network of a thousand nodes in the animation view, and to provide multiple views that the user can switch between at any time.
- The ability of Viper to reach a sustainable update rate of this view.

In the following section we discuss the impact that these requirements have on the core design. Next, we describe the Macramé network architecture in more detail to refine the requirements for Viper. We then describe the specification for Viper that supports the aforementioned requirements. After looking at the design and implementation issues that arise from the extended Viper specification, we report on the results.

7.2 The concept of customisation

As we saw with Mona Lisa, Viper uses the term *module* for the individual components that interact with each other. From the previous section it is clear that the different devices that display state information have different state definitions and requirements. Therefore, we need to be able to define multiple *module types* and associated state representations. Viper as it was developed for Mona Lisa does not support multiple module types: all modules have identical state behaviour and graphical representation.

The implementation of such capabilities relies on the adaptation of fundamental software classes in the Viper design, such as the ModuleState class. We introduce the concept of *customisation*: adapting Viper to a particular application through the definition of so-called customised sub-classes. We can illustrate this quite easily with the ModuleState class: by defining a class CustomModuleState which is a sub-class from ModuleState (see Figure 45) we achieve the following:

- A module can now have a customised module state definition by adaptation of the CustomModuleState class.
- All base functionality that is required from the state class can be incorporated in the base class ModuleState. Therefore, even an empty definition of the CustomModuleState class would be sufficient for Viper to function properly.
- The definition of the CustomModuleState class can be kept in a separate code package (in our case, the files custom.h and custom.c).

The principles of using subclasses to override default implementations with the aim of introducing application flexibility are described in an excellent way in [Gam95]. This book describes many useful object-oriented design patterns. A design pattern can be loosely defined as 'a description of the core of a solution to a recurring design problem, such that it can be used generically and repeatedly, albeit not necessarily with the same result each time'.

The patterns that are relevant here are the Factory method and Abstract factory. They describe in a generic way what we want to achieve with the Viper design: the core of the application does not care about the specifics of a customised object (e.g. CustomModuleState), as it only handles it through the interface of the base class (e.g. ModuleState), which is in any case inherited by the sub-class (CustomModuleState).

The result is extremely powerful: in the same way that a Node object visualises the state of a ModuleState object in the animation window, a CustomNode object visualises the CustomModuleState object. However, since these objects can have problem domain specific class definitions, their graphical representations can also be problem domain specific!

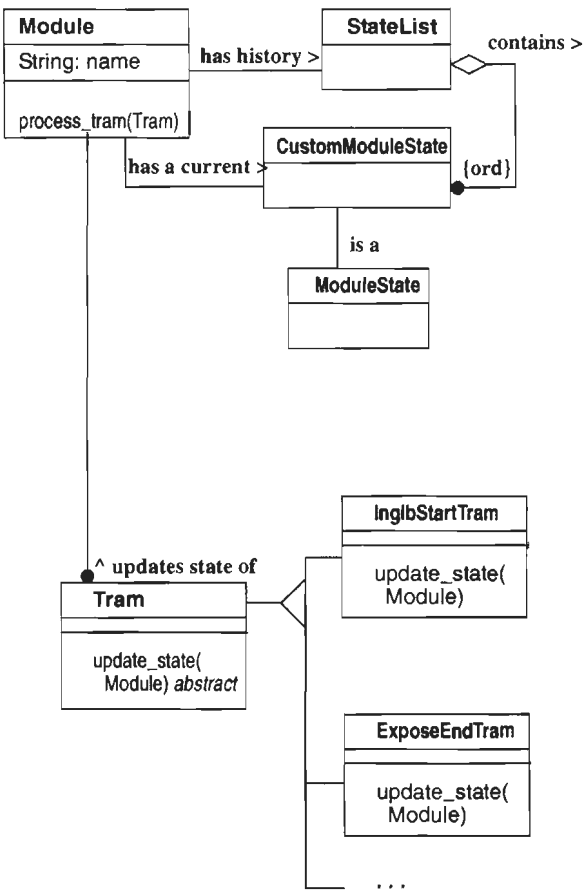

FIGURE 45. Customising the State definition of a Module

The use of factory (and other) design patterns greatly enhances the flexibility of an application from a code maintenance point of view. However, some up front investment in a more solid class structure is required (for example, which abstract factories are required to instantiate problem domain specific objects). It is therefore desirable to know in advance the dimensions along which an application has to be generic, so that the class structure design can take this into account from the start.

7.3 The Macramé network

As we mentioned in the introduction of this chapter, the Macramé network consists of traffic sources, called *nodes*, that send messages to each other through a network of switches. Both the nodes and the switches belong to the family of Transputer technology devices. The C104 switching device that is used is a 32 port asynchronous packet cross-bar switch from SGS-Thompson. The nodes that are responsible for generating messages are implemented with C101 chips.

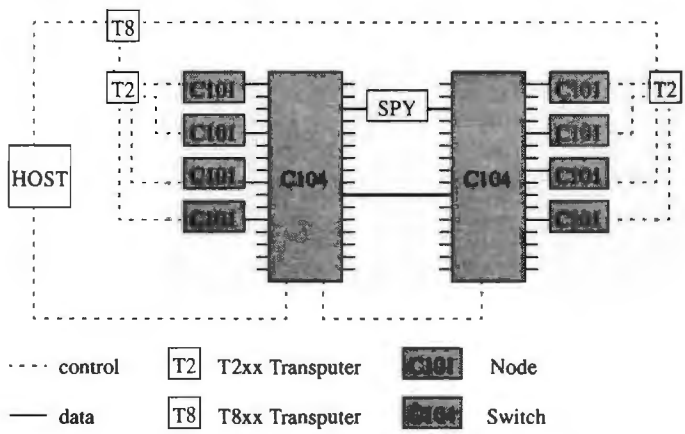

FIGURE 46. Example of a small network, showing the key components and interconnections.

Figure 46 shows some of the relevant aspects of the network technology. The example topology here is two switches interconnecting 8 nodes, indicated by the gray shading of the relevant parts. Both switches are heavily underutilised in this example, as they are each capable of fully interconnecting 2 groups of 16 nodes. The unused C104 links are drawn explicitly as small lines hanging off the C104 to illustrate this. In the full hardware configuration the 64 switches are fully utilised to interconnect over a 1000 nodes.

The C104's in Figure 46 are interconnected by two links, with one link passing through a so-called *spy* node. This node has been specially equipped to measure the properties of the individual messages that go through the link (e.g. travel time), as well as monitoring properties of the link itself (throughput, occupancy rate etc.). The spy node is also called an *intelligent* node as opposed to the C101 nodes, which are *simple* nodes.

Apart from the devices that make up the actual topology, there are also a number of devices required for the *control network*, mostly Transputers of the T2xx and T8xx series. The supporting interconnections are drawn as dotted lines in the diagram, to distinguish them from the topology interconnections over which the actual data patterns are transmitted during a run. The control network is responsible for:

- Downloading device configuration files, executable programs (such as routing software on the T8's) and data (e.g. the data patterns for the simple nodes) from the host to the relevant devices in the network. This occurs during network booting.
- Collecting measurement data from the topology. This needs to be done during the run.

- Collecting status information from the network devices (e.g. active, or in error). Such actions are initiated from the host after booting but before the run, using special software tools such as *ispy* and *T9spy*.

The switches and the nodes are part of different control networks. This is due to the fact that they make use of different link technologies: the nodes use OS link technology, whereas the switches use DS link technology. As a result, the C104's control structure is a daisy chain that is connected to the host separately.

An important device in the node control structure is the T2: this chip 'manages' a group of 4 simple nodes, collects all relevant data from these nodes and communicates it back to the host. T8's are used to take care of the routing of these messages. For a 1000 node network the control structure becomes quite a large, tree shaped network, with the host at the root, the T2's as leaves and T8's as branches. However, our example is quite small, so we only have one T8.

To allow for easily reconfigurable network topologies, standard board modules have been designed that can be interconnected on a per link basis using link cables. The link cables are part of the data network; the control network is interconnected differently. The 3 most important board modules are depicted schematically in the diagram below.

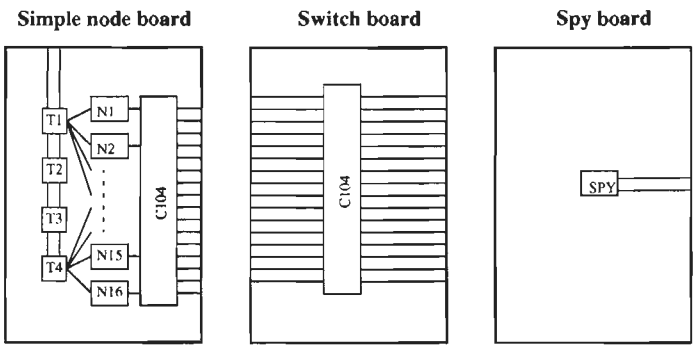

FIGURE 47. The Macramé network is built using standard modules.

The simple node board connects 16 simple nodes (numbered N1 to N16) to a switch. They are controlled via 4 T2's, numbered T1 to T4. The remaining 16 links of the switch can be connected to other boards through cabling. The switch board is simply a switch with all links available for connections to other boards. Finally, the spy node board has an intelligent node mounted on it with ingoing and outgoing links, allowing monitoring of a data stream passing through the node.

The boards are organised in *crates* where they occupy a *slot position*. The crate provides the power supply, but more importantly, also the control network interconnections that run over the crate backplane. The data network connections between boards are all established with link cables. Depending on the logical topology that has to be implemented, the cabling patterns can become quite complex, because of two restrictions:

- There is a limited number of crates (~30), and a crate has a limited number of slots.
- A link cable has a limited length.

The result of the two above limitations is that the symmetry that exists in the topology (e.g. mesh) cannot be translated completely to a corresponding symmetry in the wiring between crates.

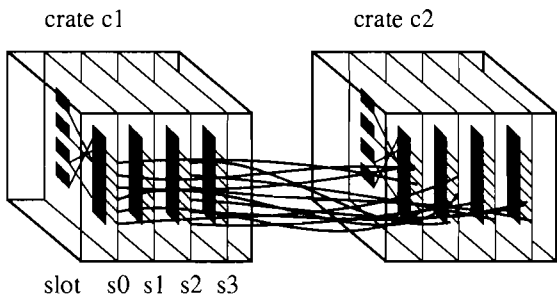

FIGURE 48. Connecting up crates can give complex and asymmetrical wiring patterns.

In the next section we discuss how Viper can visualise the state information of simple nodes and switches.

7.4 The Viper specification for Macramé

Viper visualises the execution of a Macramé test run as a sequence of state changes, very much like it visualises a Mona Lisa program execution. The state of a Mona Lisa program is a function of the state of the individual modules. Here, the state of the test bed is a function of the states of the individual devices. With Mona Lisa we were dealing with uniform modules whereas in this case we are dealing with several types of devices (nodes and switches) that each have different state behaviour. The support of multiple module types is therefore a new requirement for Viper.

The state information for the network devices can be split into two parts:

- setup information related to the network boot (setup) phase of the run.
- state information related to the execution phase of the run.

The setup information is essentially describing whether a device is included in the network configuration software (called *NDL file*) and whether it has come alive successfully after booting. If this is the case, we can detect its presence with the software tools *ispy* or *T9spy*. This setup information applies to both nodes and switches.

During the run, a node can send and receive data. We therefore define a node to consist of a transmitter and receiver part. The transmitter sends data according to the data pattern that has been loaded into the node as part of the setup. This data pattern specifies a list of messages to be sent, and per message the size and the time that it must be sent, relative to the start of the run. Subsequently, a transmitter can be in any of the following states:

- **idle/ok**: there is currently no data to be sent according to the data pattern loaded.
- **communicating**: the transmitter is sending a message as defined by the data pattern.

- **congested**: the transmitter is sending a message, but at a time that is later than defined by the data pattern due to temporary unavailability of outgoing links (i.e. blocking because of other messages being transmitted). If sufficient idle time is 'planned' between successive sends, this temporary backlog can be corrected and the transmitter goes back to **communicating** or **idle/ok**.
- **overflow**: the transmitter is sending a message, but at a time that is later than defined by the data pattern, similar to the congested case. In addition, the time difference exceeds a threshold that puts the transmitter in an overflow state to indicate the severity of the 'time overflow'. The only state transition allowed from this state is to **error**.
- **error**: a hardware failure has occurred.

The receiver part of a Node has similar states **idle/ok**, **communicating** and **error**, but not **congested** and **overflow** because it does not send data. In addition, both transmitter and receiver have a data rate associated with them, indicating the current average communication bandwidth.

If we define boolean A to represent "device is alive" and boolean B to represent "device is included in the sub-topology according to NDL file", the following state domain definitions serve as basic building blocks of state information:

- Setup_state = {not A & not B, not A & B, A & not B, A & B}
- Dynamic_state = {idle/ok, communicating, congested, overflow, error, undefined}
- Data_rate = integer value ranging from 0 to MAXRATE.

The state of a transmitter can now be defined as a compound state (Dynamic state: d, Data_rate: r) with initial state (undefined, 0). The same definition applies to the receiver. The state definition for the node and switch then become:

- Switch state: (Setup_state: s, Dynamic_state d).
  Initial state: (not A & not B, undefined}
- Node state: (Setup_state: s, Transmitter_state: t, Receiver_state r).
  Initial state: (not A & not B, initial_transmitter_state, initial_receiver_state).

7.4.1 Viper's output

Applying the concepts we introduced in chapter 3 to Macramé, we define a *view* to be a graphical representation of (part of) the Macramé test bed state information, that highlights a particular aspect of the test bed state. The intuitive traffic light colours that we used for Mona Lisa can be re-used here, for example to mark a node as communicating (green), congested (yellow) or in overflow (red). Before describing Viper's views, we need to introduce the graphical representation of nodes and switches in these views.

7.4.1.1 Graphical representation of a Node

A node with state (s,t,r) is represented by is a rectangle consisting of a transmitter and a receiver part and a base whose colour is defined by the colour mapping defined for setup state *s*.

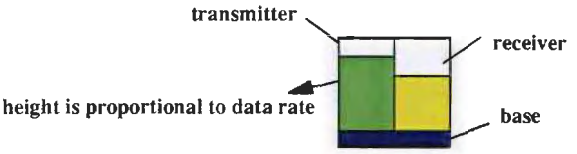

FIGURE 49. Graphical representation of a Node.

The transmitter and receiver parts are each split into an upper and lower region, suggesting a meter. The relative size of the lower region is proportional to the value of the data rate, and is coloured green/yellow/red according to the value of the dynamic state. The upper region is transparent. When the node is in error, a white cross is drawn across the rectangle.

7.4.1.2 Graphical representation of a Switch

The representation of a Switch with state (s,d) consists of a rectangle and a static text label. The rectangle's colour is normally defined by the colour mapping for setup state *s*, but overridden by the colour mapping for dynamic_state *d* in case of error.

FIGURE 50. Graphical representation of a Switch.

7.4.1.3 Viper views

We will describe the views using an example Macramé setup. This setup consists of 9 switches arranged in a 3x3 mesh topology and 9x16 nodes. This means that the nodes gain access to the communication network through a dedicated switch in groups of 16. The device boards are mounted in 3 crates with 3 slots each.

Animation view. The adaptable animation view can be re-used without significant modifications. A setup file specifies the subset of devices shown, their positions and their graphical representation. Specific device error messages can be displayed selectively, using the functionality that was created for displaying Mona Lisa primitive calls.

Figure 51 shows our example setup of 9 switches and 144 nodes. The static text label of the switches is used to indicate the vertex in the mesh that a switch and its corresponding 16 nodes represent.

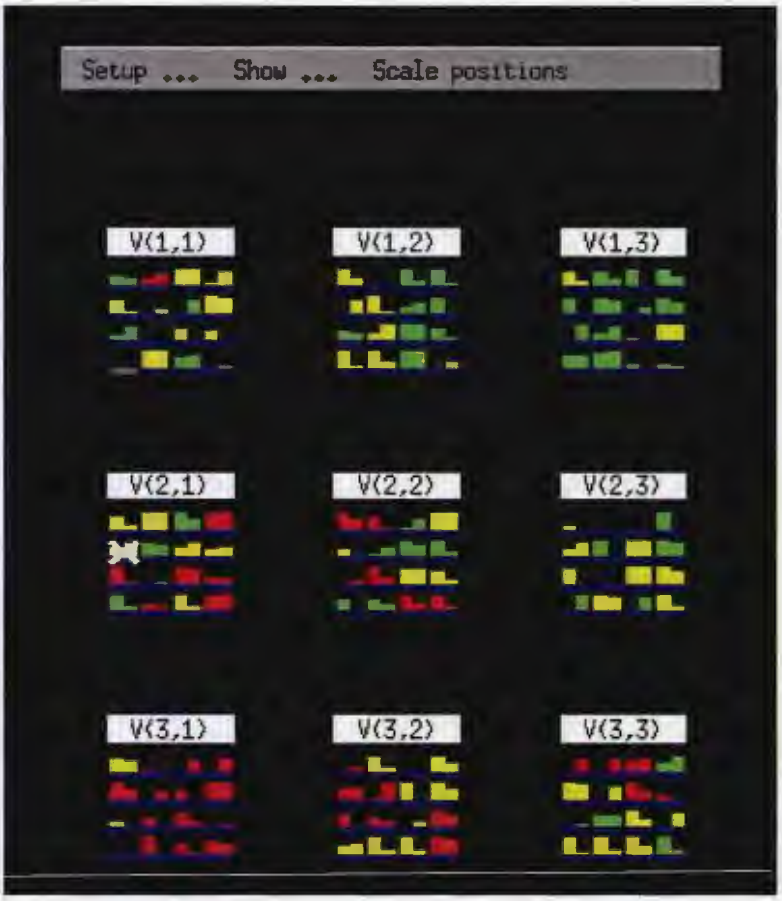

FIGURE 51. Example of a Macramé Animation View, topology view.

For the Macramé testbed we define two relevant animation views: one view that shows the setup of the hardware architecture (crates, slots, boards etc.), and another view that shows the topology setup (vertices arranged as a mesh, tree etc.). Figure 52 shows what the hardware view looks like for our example. This view is useful for locating devices with problems; for instance, faulty devices can be highlighted with a different colour. The topology view in Figure 51 enables the global analysis of on-line monitoring data, including the observation of hot spots. Both views assist in the setup process where the mapping between topology and cabling is verified.

We therefore have a requirement to support different views on the same test bed. This can be supported by using different setup files which the user can load at any particular time using the menu option 'setup...' provided on the menu bar in the animation view.

The control software that is responsible for the network analysis and setup, and message handlers that deal with messages that come out of the network, can be extended with logic for the generation of the corresponding setup files.

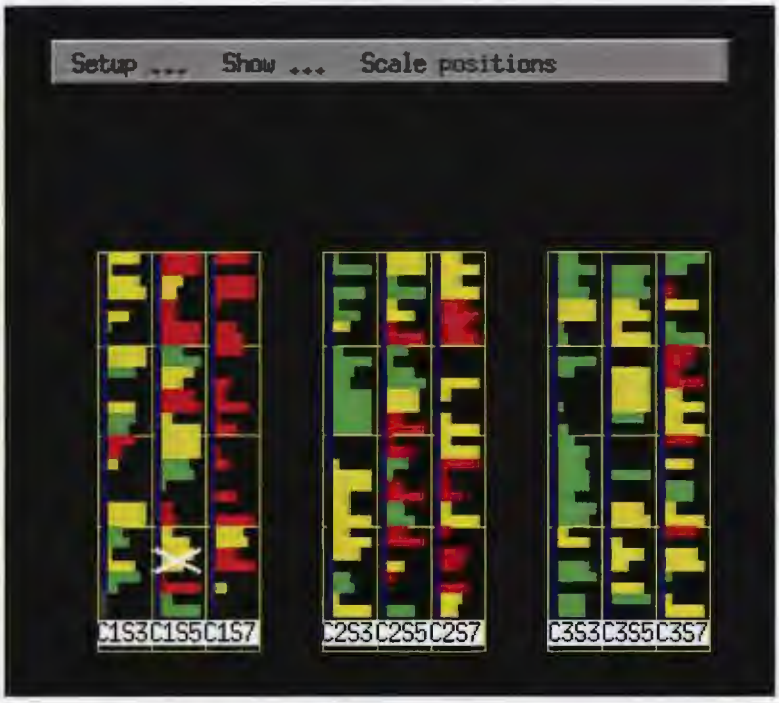

FIGURE 52. Example of a Macramé Animation View, hardware view.

A necessary enhancement to the animation view is the possibility to include additional graphical elements in the display for layout purposes only (e.g. lines and text labels) as demonstrated in the hardware view. This allows us to group nodes belonging to one board/slot/crate etc. where such detail is desired.

Space time view. A setup file specifies which nodes are shown in what order on the vertical axis. In the case of Macramé, we will have to make extensive use of the ability to visualise only a subset of the nodes given their potentially large number. Also, there is little interest in including the switches: the primary use of the space time view is the study of congestion patterns, which focuses on the node transmitters. The view is very similar to the Mona Lisa view, showing the dynamic state of the node's transmitter in a coloured bar using traffic light colours. Given the similarity to the Mona Lisa view, an example figure has not been included.

We conclude the section on Viper's desired output with the colour codings that are required to make the visualisation specification complete.

TABLE 13. Mapping of setup_state to colours

setup_state	colour
not A, not B	black
not A, used	red
A, not B	yellow
A, B, dynamic_state = undefined	grey
A, B, dynamic_state <> undefined	blue

TABLE 14. Mapping of dynamic_state to colours

dynamic_state	colour
undefined	black
idle/ok	green
communicating	green
congested	yellow
overflow	red
error	red

7.4.2 Specification of trace message types

The updates of module states, and hence the module representation, is driven by trace messages. In this section we determine what suitable trace message types we need. The reader is referred to Section 5.4.1 on page 75 where the PICL trace message standard was introduced. The discussion here is concerned with the possible values that are to be stored in this generic structure, such as the possible values for record type and event type.

The definition of PICL trace messages is based on a one-to-one correspondence between the occurrence in the network of an event that affects the state of a device, and the registration of this event by the generation of a trace message. Events are categorised into types, and each event type corresponds to one trace message type, as listed in Table 15.

Naïvely, a single event type could be defined to cover all possible state changes. However, it is not the best choice if we want to optimise the size of the trace messages. As we record different state information for different devices, it makes sense to have a specific event type (and thus trace message type) for each module type; in this way we can reduce the amount of redundant state information stored in the trace message. This leads us to define two event types: "Node changed" and "Switch changed".

For each state aspect we can distinguish a specific phase in the network simulation run where this aspect is subject to change. For instance, the information whether a module is found by *ispy* only becomes available (and therefore only causes a state change) during the setup/initialisation phase. By using different event types for different run execution phases, we can again reduce the data redundancy in a trace message.

The above leads to the following definition of event and trace message types:

TABLE 15. Overview of event types and the mapping to trace message types

event type	description	trace message type
module found	ISPY result on the network module	Module_found
module used	NDL interpretation for this network module	Module_used
Node changed	a Node's (new) state has been recorded	Node_changed
Switch changed	a Switch's (new) state has been recorded	Switch_changed

For the initial state we assume a device not to be included in the NDL file and not alive. During the setup phase, an appropriate trace message of type Module_used is gener-

ated for each device that is part of the NDL description, and a device generates another trace message itself (of type Module_used) upon coming alive. All other trace messages are generated during the actual run execution.

Viper requires trace messages to be formatted according to the PICL standard and only accepts numerical fields. Also, Viper requires a sequence number in every trace message. The resulting trace message template is described in Table 16

TABLE 16. Trace message template.

field	value	description
record type	2	indicates nature of event (start, end, instant). we only have instant.
event type	30, 31, ...	mapping of event type to integer values
time stamp	integer	Viper imposes no constraints on quality of time stamp values. Ordering of trace messages is established using the sequence number.
processor id	integer	id of the component that generated the trace message
process id	0	not used.
nr. of data fields	integer	nr. of data fields (they contain the actual state information)
type of data fields	2	we only have numerical data fields.
data field 1	integer	data field
data field 2 etc...	integer	data field

Applying this template to each trace message type we get:

TABLE 17. Overview of trace message types and their formatting.

trace message type	format
Module_found	2 30 <time_stamp> <id> 0 1 2 <seq>
Module_used	2 31 <time_stamp> <id> 0 1 2 <seq>
Node_changed	2 32 <time_stamp> <id> 0 5 2 <seq> <transmitter_state> <transmitter_rate> <receiver_state> <receiver_rate>
Switch_changed	2 33 <time_stamp> <id> 0 2 2 <seq> <state>

where

- <id> stands for the component identification number (consecutive from 0 onwards).
- <seq> stands for the sequence number of the trace message for component <id>. Trace messages are numbered per component consecutively from 1 onwards.
- <state> stands for the numerical value of dynamic_state for component <id>. Mapping of dynamic_state values to numbers is by enumeration, starting at 0. Similar definitions for <transmitter_state> and <receiver_state>.

<transmitter_rate> and <receiver_rate> stand for the data_rate value for the transmitter and the receiver of component <id>.

7.4.3 Performance improvement

The number of Nodes in the Macramé project will be significantly larger than those encountered in our Mona Lisa examples. The Viper application is definitely not geared

up to supporting a thousand graphical objects in the Animation view without significantly slowing down the trace message processing rate. For Macramé, the following performance target is set: With the animation view opened, Viper should be capable of processing from file 1000 trace messages per second. This corresponds roughly to 1 update per node per second, and hence a complete 'refresh' of the animation view each second.

This performance target is to be achieved on the fastest platform available to the GP-MIMD group, a SPARC workstation running Solaris.

7.5 Design implications for Viper

From the previous section we can summarise the new requirements for Viper on which its usability for Macramé depends:

1. The ability to support different module types (node and switch) that exhibit different state behaviour.
2. The ability to graphically represent different modules in different ways, so that for example a node and a switch can have a different appearance on the screen (for Mona Lisa we could represent each module by the same, coloured circle). In addition, the following graphical elements should be available for layout purposes in the animation view: lines, rectangles and text labels.
3. The ability of Viper to display a network of a thousand nodes in the animation view, and to provide multiple views (topology and hardware) that the user can switch between at any time. The latter is done by using different setup files which the user can load at any particular time using the menu option 'setup...' on the menu bar in the animation view.
4. The ability of Viper to reach a sustainable update rate of this view in the on-line mode, in the order of 1000 node state changes per second.

Some of the functionality that is essential for Mona Lisa is not required here: the interaction between network components at the individual node level is not a relevant visualisation aspect. In particular, a consistent run construction is not an issue. In the following sections we will discuss how the new requirements affect the Viper design.

7.5.1 Viper customisation

This section discusses in more detail the implementation of the concept of customisation as discussed in Section 7.2. For the proper modelling of the state behaviour of the Node and Switch modules we have two design options:

1. implement two separate class definitions of type CustomModuleState, one for each module type.
2. implement one CustomModuleState class that contains the superset of state components of all module types.

We decided to implement option two, mainly to keep the complexity of additional class definitions to a minimum.

For the CustomModuleState definition in Macramé this means we define the following class:

```
class Customised_module_state : public Module_state {
 public:
  Setup_state setup_state;
  Node_state overall_state;
  Node_state receiver_state;
  int receiver_rate;
  Node_state transmitter_state;
  int transmitter_rate;
  void clear() {
    setup_state = NOT_F_NOT_U;
    overall_state = UNDEFINED;
    receiver_state = UNDEFINED;
    receiver_rate = 0;
    transmitter_state = UNDEFINED;
    transmitter_rate = 0;
    Module_state::clear();
  }
};
```

As can be seen in the class definition, this means that the Switch module contains the state components only relevant for the Node module and vice versa. This introduces a certain level of redundancy, but this is not considered to be too critical. The advantages of a single class definition outweigh the disadvantages: limited change in Viper's design, and limited effort required for defining a customisation.

The code package that contains all the customisation of Viper includes much more than just the state class definition. We mention here: primitive classes, trace message type classes, the associated abstract factory classes and the graphics classes that need to be customised to generate the desired look-and-feel in the animation view. The same principle has been followed for these classes and is not discussed further in this thesis.

7.5.2 Adaptation of the animation view

The animation view has to be adapted in a number of ways to support Macramé:

- The setup file needs to be extended with an additional column where we specify for each module what representation we require.
- The setup file definition needs to be extended with a section where we can specify layout elements (lines, rectangles and text labels).
- The setup file needs to be re-loadable at any time to switch between alternate views.
- The graphics handling in this view needs to be optimised to achieve the required performance increase to approx. 1000 updates per second.

The implementation of these features is straightforward from a Viper design perspective and is straightforward software development.

7.5.3 Animation view setup files

The setup files are generated by the same tools that generate the NDL files. This systematic linkage ensures the files are synchronised. Manual generation or indeed any manual intervention in the file generation process would increase the probability of setup errors.

7.6 Results

At the time when Viper was adapted for Macramé, the project was in its initial phase and the testbed still under construction. As a result, Viper could only be tested in a limited way. An integration test was performed with the first Node board produced for the Macramé testbed. This test demonstrated the viability of the use of Viper from a functional point of view.

To test Viper's capability to deal with a large number of Nodes, a fictitious trace file was produced to simulate a Macramé setup of 64 x 16 = 1024 nodes in an 8x8 mesh. A trace file processing rate of approx. 800 trace messages per minute was achieved, which was considered to be acceptable.

From a functional point of view, this simulation test was a success: on a 21" monitor, the size of an individual node was reduced to approx. 0.7 x 0.7 cm., but this was still large enough to show all the relevant detail such as congestion state of the transmitter, and the qualitative data rate of both the transmitter and receiver.

The speed at which Viper proved to be adaptable to Macramé's requirements was an important success: within 2 months we demonstrated that Viper could fully support the visualisation of Macramé test runs. This equated roughly to the total effort required on the testbed side to implement the generation of the required PICL trace messages. This highlights an important point of visualisation tools: their dependency on quality trace information is crucial. Even with the most flexible and adaptable visualisation tool, a lot of hidden effort can go into preparing the basic infrastructure such as trace message generation.

At the Macramé project review held in November 1996 for the ESPRIT review board, a real-time demonstration was given using Viper, where it visualised a setup of 256 nodes. As a conclusion we can say that the Macramé project was used to demonstrate some important design objectives: the general purpose nature of Viper, and the re-usability of its design.

Chapter 8

Discussion and conclusion

In this last chapter we will discuss some aspects of the work performed that are relevant to the criteria, described in [Bco94], against which a designer's thesis is tested.

At the start of this designer's PhD, the following objectives were formulated in the project proposal [Schi93/2]:

- Realisation of a Mona Lisa software engineering environment, comprising all the indispensable tools for the life cycle of parallel program development.
- Case study of the parallelisation of a representative HEP event reconstruction program.
- Publication of the case study results, comprising not only a parallel program but also design criteria for parallel HEP reconstruction programs.
- Publication of aspects of Viper that are of scientific importance.

We will discuss the extent to which this plan was executed in its original form, and the results obtained. Finally, the scope for further work is outlined.

8.1 Viper

The project has demonstrated that the visualisation requirements as formulated in the GP-MIMD project can be met with the Viper tool. The flexibility of the graphical representation allows the tuning of the visualisation to the problem domain. Furthermore, it supports scaling to a large numbers of elements. This makes Viper suitable for visualisation of massively parallel applications - the core objective of our work.

Using Viper for parallelising HEP software

Specifically within the context of the GP-MIMD project, Viper has been developed to serve as the core component in the Mona Lisa software engineering environment (the first objective mentioned at the start of this chapter). The CPREAD case study has subsequently demonstrated that, within the context of Mona Lisa, Viper assisted in the construction of the parallel version of CPREAD. From a behavioural point of view, Viper was used to verify that there was reasonable opportunity for load balancing. From a program tuning point of view, Viper demonstrated the performance bottleneck that was incurred by the usage of the INRGLB primitive call. Finally, some debugging support was also provided during the first test runs that lead to the discovery of the software bug in the Mona Lisa communication library.

Using Viper outside the HEP parallelisation context

In this thesis we have stressed the benefit of programming paradigms with a high level of abstraction. Nevertheless, it is vital that Viper itself does not rely on any specific paradigm - it merely has to be able to exploit the presence of one. Paradigm independence has been a key consideration in our design decision to separate the visualisation process from the model to be visualised. Instead of letting an object in the state model visualise itself, a strict separation has been introduced which means that each object has a (configurable) visualisation object associated with it.

The demonstration of the achievement of this paradigm independence took place in the Macramé project. Here, Viper visualised traffic patterns in a large network of communicating nodes, with objectives, similar to those for parallel programming support: behavioural analysis and debugging of the testbed configuration. The systematic steps used to map the new problem domain to a set of visualisation objects and trace message types, showed how Viper can be adapted quickly to a different problem domain.

Comparison of Viper with other visualisation tools

In Section 2.4 an overview of visualisation tools has been given, including a detailed description of one of the most prominent members ParaGraph. We outline here in summary where Viper's added value lies compared to ParaGraph:

- Ability to provide adaptable views at the programming paradigm level, without compromising the general purpose nature of the tool. This was demonstrated by the work on CPREAD and Macramé.
- Ability to construct a consistent observation by exploiting the properties of the paradigm, without having to rely on the accuracy of the trace message time stamps.
- Ability to provide real-time visualisation whilst maintaining observation consistency. Especially in the context of unreliable time stamping, this is a unique feature of Viper.

These features clearly distinguish Viper from ParaGraph, or any other available visualisation tools that were researched at the start of the project. The innovative nature of these features resulted in a publication [SS95].

The Mona Lisa case study proved the added value of paradigm related visualisation as provided by Viper compared to the ParaGraph views. An important question however concerns the synthesis with ParaGraph: is the existence of Viper on its own justifiable? Should there not be a design where Viper and ParaGraph are incorporated in the same tool, with a paradigm dependent part for the Viper views ?

The answer to that last question may well be yes. The boundary conditions in which the work has been carried out did not allow for this though. It is important to recognise that a well-defined scope definition on a project, resulting for instance in a fixed time period and effort, is designed to deliver its benefit in a way which is not necessarily ideal, but nevertheless cost effective (where cost refers to any scarce resource which needs to be expended within the project).

A full integration with ParaGraph was never in the scope of the project, simply because of the effort required. Nevertheless, the combination of Viper and ParaGraph is an area where significant work can still be done. Future work can address both the details of

the proposed interaction between Viper and ParaGraph using trace files, which was originally intended to be demonstrated using a message passing paradigm. Also, the scenario of a single application design incorporating the concepts of both tools merits further study.

8.2 CPREAD case study

The CPREAD case study constitutes the second objective as listed at the start of this chapter. The primary conclusion here is that with limited code change, a reasonable speed-up at single event level can be achieved, although within limits. However, the economics (business case) needs to be assessed for each application individually, given the varying levels of investment and return involved. The results of the case study, together with the application design criteria that alleviate the investment required for parallelisation, are published [SF95].

8.3 Evaluation of Mona Lisa

Mona Lisa was a critical case study for Viper to highlight the added value of paradigm related visualisation. In this context, we have shown that paradigms are useful for enriching the information presented to the user (e.g. showing primitive calls being executed). When Mona Lisa was put to the test with the parallelisation case study of CPREAD, the results from a performance point of view were less encouraging. Although an interesting concept, Mona Lisa had inherent scalability weaknesses. This weakness was particularly noticeable with the Chorus operating system implementation, which used the relatively inefficient Chorus communication mechanisms.

The conclusions reported in the publication article for RT'95 mention this, in addition to the design criteria for parallel HEP reconstruction programs. As a result, our focus on the implementation of a full Mona Lisa development environment (as per original plan) shifted towards other, more relevant tasks such as finding a solution for the consistent run observation under unreliable trace message time stamping.

8.4 Evaluation of the design process

Original time plan versus execution

The original project proposal already contained a global time plan. Most of these tasks have been executed according to plan in a straightforward manner, such as the CPREAD case study and the publications. However, other tasks were re-defined as a result of project scope changes.

The disappointing scalability of Mona Lisa resulted in an important decision - no further investment would be put into this development environment. In addition, the unexpected difficulty encountered with trace message time stamps meant a shift of priority: instead of working on the integration of Viper with a sequential debugger, we tackled the consistent observation problem. This proved to be such a sizeable task, that the work on the analysis tool for process-to-processor allocation strategies, as originally foreseen, was taken out of the project scope.

Finally, there was another unforeseen task: the re-usability of Viper was validated in the Macramé project.

Software engineering approach

Viper's adaptation to Macramé's problem domain demonstrated the importance of software changeability during an application's lifecycle. Indeed, it is typical for software applications to undergo rework at some point as a result of changing requirements in the environment. In Viper's case, the dimensions where flexibility was required were relatively unknown at the start. The participation in Macramé, for instance, was not foreseen. As a consequence, we introduce risk when taking design decisions that limit flexibility. For Viper, a well-designed framework and an incremental development process proved to be a successful combination to manage this risk.

Adoption of this approach in commercial environments where similar risks exist (e.g. applied research institutes) could have the additional benefit that incremental investment can be weighed more explicitly against the incremental benefit - thus ensuring a focus on adding value during the software development process.

The iterative software approach also proved to be successful in coping with the project scope changes that were mentioned before. Finally, the iterative approach was also useful in dealing with the heterogeneous user groups (GP-MIMD project members for CPREAD; the experimental Mona Lisa user community; Macramé project members for the test bed visualisation). Using intermediate software versions, it was possible to portray the possibilities of the Viper tool at an early stage, and this stimulated the discussion which drove the tool's development in the desired direction.

Future work

As discussed, most of the original project goals were reached. Where substantial work is still required, is in the integration with ParaGraph, and the integration with a sequential debugger for variable value inspection. Finally, although in a sense aesthetics always plays a role when creating a tool with a prominent user interface, specific objectives for user friendliness where not specified in Viper's design process. The tool might well benefit from some further work in this area.

8.5 Conclusion

Viper is a visualisation tool that supports a developer in the construction of parallel programs. It does so by providing graphical feedback at the programming paradigm level (real-time if so desired), to facilitate the developer in his programming task. In the context of HEP, we successfully demonstrated this capability with the parallelisation of CPREAD using the Mona Lisa paradigm. This case study also showed that with limited code rewriting, a reasonable speed-up of sequential HEP applications at single event level can be obtained. The general purpose nature, re-usability and scalability of Viper were demonstrated in the Macramé project where it visualised traffic patterns in a network of up to 1024 nodes.

Appendix A

A Mona Lisa program example

To illustrate the Mona Lisa concept, we introduce here an example parallel program. The bulk of this program is assumed to be written in the (sequential) language Fortran. The Mona Lisa parallel programming paradigm has been used to implement the logic required to obtain a parallel program.

Our Mona Lisa program is called **Farm**, and consists of 5 modules: one master and four slaves. The program performs a repetitive task: the master repeatedly reads in an image from a file, processes this image and writes the results to file again. The slaves assist the master in the image processing. When the master starts on a new image, it partitions the image data in four parts. Each partial image acts as a job specification for one slave. The job specifications are distributed over the slaves, who start to process the (partial image) data. When a slave has finished, it makes the resulting data available to the master. When the master has collected and combined the data from all slaves, it performs the final image processing stage, writes the image to file, and reads the next image to be processed. This continues until the whole image file has been processed.

Figures 53 and 54 show the Mona Lisa program in a stylistic way; only the code sections relevant to this example are shown. We assume the master to read from the file INFILE, and to write to the file OUTFILE. The image data is initially read into the variable IMAGE and consists of 1600 pixels (of type INTEGER). A subrange of the array (i.e. the partial image data) is assigned to JOB and exposed in a DO loop.

Each slave executes the same infinite loop, so we can suffice with a replicated module definition. A loop iteration starts with reading the data stored in JOB into the variable RESULT. This data is then processed, and the variable RESULT is exposed. The slave then immediately tries to read the next JOB. This primitive call will block until the master is ready to expose JOB again, with JOB containing data of the next image.

The master collects the results from the slaves through a loop, storing the data in the variable IMAGE. After some final processing, it then writes the data to file, and proceeds with the next image until the end of the file is reached. When the master terminates, the slaves (that are still trying to read the variable JOB of the terminated master module) are automatically terminated by the program manager as well; the program manager can detect that no further progress of computation is possible.

```
      MODULE MASTER
      GLOBAL INTEGER JOB(400)

      PROGRAM MAIN
      INTEGER IMAGE(1600)

      <open files, initialise program>
****** WHILE NOT EOF DO
100     READ(INFILE,END=400) IMAGE
        DO 200 I=0,3
          JOB=IMAGE(1+I*400:(I+1)*400)
          EXPOSEGLB("JOB")
200     CONTINUE
        DO 300 I=0,3
          SLAVE.INRGLB("RESULT",IMAGE[1+I*400:1+(I+1)*400])
300     CONTINUE
        <do final processing>
        WRITE(OUTFILE)IMAGE
        GOTO 100
****** END OF MAIN LOOP
400   <close files, terminate program>
      END PROGRAM

      END MODULE MASTER
```

FIGURE 53. Code for the module MASTER.

```
      MODULE SLAVE[4]
      GLOBAL INTEGER RESULT[400]

      PROGRAM MAIN
100   MASTER.INGLB("JOB",RESULT)
      <process data, stored in RESULT>
      EXPOSEGLB("RESULT")
      GOTO 100
      END PROGRAM

      END MODULE SLAVE
```

FIGURE 54. Code for the SLAVE modules.

Appendix B

Module setup file syntax

The syntax of Viper's Module Setup File can be defined easily using an LL-1 grammar definition:

```
File::          Module_count_header;
                module_count;              (nr of modules in program)
                Group_count_header;
                group_count;               (nr of module groups)
                [group_mapping]            (range of module id's per group)
                Module_def_header;
                {Module_def}*
                Global_id_def;             (definition of group comprising all modules)
                {Group_def}*
                {Attribute_def_header;Attribute_list}*
Group_mapping::
                Group_mapping_header;
                {Module_range}*
Module_range::
                module_id;                 (module with smallest id in group)
                group_size                 (size of module group)

Module_def::
                module_file;               (file name of module executable)
                module_name;               (name of the module)
                module_id                  (id nr of the module)
Group_def::
                "-";
                group_name;
                group_id;
Attribute_list::                           (list of attributes of a particular module)
                {Attribute_def}*
Attribute_def::
                attribute_name;
                attribute_id;
                attribute_type

Module_count_header::
                "MODULES:"
Group_count_header::
                "GROUPS:"
Group_mapping_header::
                "RANGE FOR GROUPS:"
Module_def_header::
                "DEFINITION TABLE:"
Attribute_def_header::
                "#ATTRIBUTES for module ";
                module name;
```

```
            "#"
Global_id_def::
            "- # all_modules #";
            module_id
```

Grammar terms that start with a lower case letter, not further defined in the above rules, are either integer or string values (which one should be clear from the context).

The following example file corresponds to the example program Farm, introduced in Appendix A. The Mona Lisa pre-processor generates this file automatically when it parses the Mona Lisa source code.

```
MODULES:
5
GROUPS:
1
RANGE FOR GROUPS:
1 4
DEFINITION TABLE:
file name               module / group name        id
farm.MASTER             module MASTER              0
farm.SLAVE1             module SLAVE[1]            1
farm.SLAVE2             module SLAVE[2]            2
farm.SLAVE3             module SLAVE[3]            3
farm.SLAVE4             module SLAVE[4]            4
-                       # all_modules #            5
-                       replication group SLAVE    6

#ATTRIBUTES for module MASTER#
attribute name          id                         type
JOB                     0                          INTEGER(400)
#ATTRIBUTES for module MASTER#
attribute name          id                         type
RESULT                  0                          INTEGER(400)
```

FIGURE 55. Example of a module setup file.

Summary

This thesis describes the development of Viper. Viper has been developed during a designer's Ph.D. project carried out at CERN, the European Laboratory for Particle Physics, in collaboration with the Eindhoven University of Technology. Viper, which stands for VIsualisation of Parallel program Execution at Run-time, aims at supporting the parallel program developer in tuning and debugging activities using visualisation techniques, thereby raising the level of understanding of parallel program behaviour .

The basis of Viper's capabilities lies in the parallel program's event history: data recorded during the program's execution for debugging and/or performance evaluation purposes. Viper replays these events, thereby providing the user with a sequence of parallel program state changes in graphical form. The user can analyse the interaction pattern of the sequential components of the parallel program, which data is exchanged when etc.

The performance requirements on the software supporting High Energy Physics (HEP) experiments is continuously increasing. Parallel computing is used at present, and will be used increasingly to meet future demands. CERN's research efforts into parallel computing include the GP-MIMD ESPRIT project, the context in which this work has been performed.

One of the underlying aims of the GP-MIMD project was to investigate and demonstrate the potential benefits of parallel computing for existing HEP applications. On the whole, these are very large applications written in Fortran77. This relatively old language does not have the expressive power to allow for a simple transition to parallel machines. A new paradigm called Mona Lisa was developed within GP-MIMD that provides (through a language extension) the functionality needed for constructing parallel applications. This allows a large part of the Fortran77 software to be maintained in its present form while migrating sequential applications to parallel machines.

Viper has been successfully applied to the parallelisation of the HEP application CPREAD. In the case of CPREAD we learned that a strategy of minimal code change limits the amount of parallelism that can be introduced and gives only a small gain in execution time. Higher gains would have to come progressively from significant code rewriting. Future HEP applications provide a more profitable area for parallel computing where the parallel perspective can be taken into account at an early design stage.

Viper has been developed addressing the following issues: the consistent observation of a parallel computation under unreliable trace message time stamping; support of parallel programming paradigms with a high level of abstraction such as Mona Lisa; and finally, programming paradigm independent design of the application. The latter is illustrated by the tool's use in a completely different context from Mona Lisa and CPREAD: the investigation of traffic patterns in very large switching networks.

Samenvatting

Dit proefschrift beschrijft de ontwikkeling van Viper. Viper is ontwikkeld als proefontwerp, uitgevoerd bij CERN, het Europees Instituut voor Hoge Energie Fysica, in samenwerking met de Technische Universiteit Eindhoven. Viper, wat staat voor Visualisatie van Parallelle Programma Executie gedurende Run-time, beoogt de ondersteuning van programma-ontwikkelaars bij afstel- en debugactiviteiten, door middel van visualisatie technieken die het inzicht in het parallelle programma gedrag vergroten.

Aan de grondslag van Viper's capaciteiten ligt de event historie van het parallelle programma: gegevens, geregistreerd tijdens de programma-uitvoering voor debuggen en/ of prestatie evaluatie doeleinden. Viper herspeelt deze events als een rij van programma status veranderingen, en wel in grafische vorm. De gebruiker kan vervolgens het interactiepatroon van de sequentiële componenten van het parallelle programma analyseren, welke gegevens zijn uitgewisseld etc.

De prestatie-eisen gesteld aan software ter ondersteuning van hoge energie fysica experimenten nemen continue toe. Parallelle berekeningen zullen in toenemende mate worden toegepast om aan de vraag naar rekencapaciteit te voldoen. Het GP-MIMD ESPRIT projekt is een voorbeeld van CERN's onderzoeksinspanningen op het gebied van parallelle berekeningen, en vormt de context waarin dit werk is verricht.

Eén van de hoofddoelstellingen van het GP-MIMD project was het onderzoeken en demonstreren van de potentiële voordelen van parallelle berekeningen toegepast op bestaande fysica applicaties. Deze applicaties zijn merendeels erg groot en geschreven in Fortran77. Deze relatief oude taal staat een eenvoudige transitie naar parallelle machines in de weg. Een nieuw paradigma genaamd Mona Lisa en ontwikkeld in GP-MIMD, verschaft d.m.v. een taal extensie de benodigde functionaliteit voor de constructie van een parallel programma. Dit stelt ons in staat een groot deel van de Fortran77 software intact te laten in het process van migratie naar parallelle machines.

Viper is succesvol toegepast in de parallellisatie van de fysica applicatie CPREAD. In deze specifieke studie leerden we dat een strategie van minimale code verandering de graad van parallellisme die kan worden geïntroduceerd gelimiteerd is, en dat dit slechts in een kleine verbetering in de executietijd resulteert. Meer voordeel zou behaald kunnen worden ten koste van progressief code herschrijven. Toekomstige fysica applicaties bieden een aantrekkelijker gebied voor parallelle berekeningen waar het parallelle perspectief reeds in het ontwerpstadium in beschouwing kan worden genomen.

Viper's ontwikkeling addresseert de volgende wetenschappelijke uitdagingen: een consistente observatie van een parallelle berekening onder onbetrouwbare tijdstempels van trace messages; ondersteuning van parallelle programmeerparadigmas met een hoog abstractieniveau zoals Mona Lisa; en tenslotte, programmeer paradigma onafhankelijk ontwerp van de applicatie. Dit laatste is geïllustreerd middels de toepassing van Viper in een volledig andere context als Mona Lisa en CPREAD: het onderzoek van communicatiepatronen in grote schakelnetwerken.

Acknowledgements

There are many people who have helped and aided me in the work that has culminated into this thesis. In particular I would mention Peter van der Stok, Ian Willers, Brian Martin and Bob Dobinson, for providing me with the opportunity to carry out the research under their support and guidance.

Peter's personal qualities, and his perseverance as Ph.D. supervisor and thesis reviewer have been especially appreciated, without which it would have been impossible to bring this thesis to an end. Thanks go to Ian for ensuring CERN's sponsorship, and for his helping hand in getting the project started. Appreciation also for Bob's unrelentless drive to stimulate young talent in his group. Brian will always be remembered for his marketing skills, his pantomime writing talent, and his wit which kept us all in good spirits. Thanks go out to Peter Hilbers, for his willingness to support my work under his professorship.

Amongst the many colleagues with whom I have had the opportunity to work with, I would like to mention the following people that have contributed to Viper and this thesis directly: my close friend José Pagès, who taught me the essentials of HEP during long nights and contributed to focusing me on the right things; Michael Ward, whose expertise in Chorus was essential to me; Dave Francis, for his collaboration on the CPREAD parallelisation study; Stefan Haas, for the integration of Viper and the Macramé testbed; and last but not least my friend Roger Heeley who provided a soundboard on many design issues, and supported me in the T9 implementation of CPREAD.

In addition, the following friends and colleagues have contributed a great deal indirectly: Adrian Mills, who taught me the importance of knowing your system administrator and introduced me to Right Said Fred; Erco Argante, who taught me a lesson or two in skiing, unhindered by his Dutch nationality; and Bernard Sarosi, who was exemplary of the superb life style in Pays de Gex. My wife Isabelle is mentioned here for her understanding and persuasion that enabled me to finish my thesis in the odd hours that private and working life left available.

Finally, this work would not have been possible without its sponsors. It has been jointly funded by the Ph.D. programme at CERN and the designers Ph.D. programme of the Stan Ackermans institute for continuing education (SAI) at the Eindhoven University of Technology, the Netherlands.

Curriculum Vitae

René Schiefer

20th of December 1969	Born in Rotterdam, the Netherlands
1981 - 1987	Secondary school (VWO) Katholieke Scholengemeenschap Etten-Leur
1987 - 1991	Computing Science Eindhoven University of Technology
1991 - 1993	Software technology (post-masters programme) Eindhoven University of Technology
1993 - 1996	Ph.D. research Eindhoven University of Technology, Executed as Doctoral Student at CERN, Geneva
1996 - 1999	Business Consultancy Shell International B.V., Den Haag
1999 - Present	Asset Management Tools in Private Banking Deutsche Bank, Geneva

References

[Arg98] E. Argante, "CoCa: a model for parallelization of high energy physics software", Ph.D. thesis, University of Eindhoven, 1998.

[Atl94] ATLAS Collaboration, "The ATLAS Technical proposal", CERN/LHCC/94-43, LHCC/P2, ISBN: 92-083-067-0, December 1994.

[Bbn95] Bot Beranek and Newman Inc., TotalView User's Guide v3.4, March 1995. See also http://www.bbn.com/tv/.

[Bco94] Bestuurscommissie Ontwerpers- en korte Onderzoekersopleidingen, "Notitie: Op weg naar promotie op proefontwerp", February 1994.

[Boe88] Boehm, IEEE Computer Vol. 21, no. 5, 1988.

[Boo94] G. Booch, "Object Oriented Analysis and Design with Applications", The Benjamin/Cummings Publishing Co., ISBN 0-8053-5340-2 1994.

[BH92] M. Brown, J. Hershberger, "Color and Sound in Algorithm Animation" in IEEE Computer, December 1992.

[CG90] N. Carriero, D. Gelernter, "How to write parallel programs", The MIT Press, Cambridge, 1990.

[Cho94] Chorus systèmes, "Chorus Kernel v3 r5 for T425/T805, Overview", Technical Report CS/TR-94-75, May 1994.

[Cod94] CodeME S.A.R.L. France, "CMZ User's guide", 1994.

[Com95] IEEE Computer Vol. 28 No. 11, November 1995. Special issue on parallel and distributed processing tools.

[Cul99] David E. Culler et al., "Parallel Computer Architecture", Morgan Kaufmann, 1999.

[Dob95] R.W. Dobinson et al., GP-MIMD deliverable D5.1/4: "Demonstration and final report on parallelisation of High Energy Physics analysis software", 1995.

[Ffr95] FFREAD, Long Writeups I302, CERN Program Library Office, 1995.

[Gam95] E. Gamma et al., "Design Patterns", Addison-Wesley, 1995.

[Gea95] Geant package, CERN Program Library Office, CERNLIB, CERN, 1995.

[Gei93/1] A. Geist et al., PVM 3 User's Guide and reference manual, Oak Ridge National Laboratory, May 1993.

[Gei93/2] A. Geist et al., "Visualisation and Debugging in a Heterogeneous Environment", Computer, June 1993.

[Gpm94] GP-MIMD Consortium, GP-MIMD P5404 Technical Annex for Esprit, 1994.

[Hbo95] HBOOK, Long Writeups Y250, CERN Program Library Office, 1995.

[HE91] M. Heath, J. Etheridge, "Visualising the Performance of Parallel Programs", IEEE Software, September 1991.

[Hey89] A. J. G. Hey, "Experiments in MIMD parallelism", in Lecture Notes in Computer Science, Vol. 366, (Springer-Verlag 1989), pp.28.

[Hil95] P. A. J. Hilbers, "Parallel rekenen: een uitdaging voor industrie en wetenschap", inauguration speech at the Technical University of Eindhoven, September 1995.

[Hoa86] C. A. R. Hoare, Communicating Sequential Processes, Prentice Hall 1986.

[Inm93] Inmos Ltd, Occam 2 Reference Manual, Prentice Hall ISBN 0-13-6293 12-3, 1993.

[Koe94] C. Koebel et al., "The High Performance Fortran Handbook, MIT Press, Cambridge, MA, 1994.

[KZ94] H. Klein, J. Zoll, "Patchy Reference Manual", CERN Program Library Office, 1994.

[Lin92] Mark Linton et al., "InterViews Reference Manual", 3.1 edition, Stanford University, 1992.

[Map93] L. Mapelli, LHC Era Computing, CERN Acadamic Training, May 1993.

[Mar98] B. Martin et al., "Realisation and Performance of IEEE 1355 DS and HS Link based, High Speed, Low Latency Packet Switching Networks", IEEE Transactions on Nuclear Science, Vol. 45, No. 4, August 1998.

[Mic92] M. Michelotto, "Delfarm, the delphi offline production event farm", Proceedings CHEP92, pp.459, 1992.

[Moo95] J. Moonen, "A test bed for ATLAS level 2 trigger communication using a C104-based architecture", OOTI report, Technical University of Eindhoven, August 1995.

[MS95] J. Maillard, J. Silva, "Track Parallelisation in GEANT Detector Simulations", Collège de France, Paris, France, 1995.

[Mul93] S. Mullender, "Consistent global states of distributed systems", Distributed Systems, Addison Wesley, 1993.

[OMI95] OMI/Macramé P8603 ESPRIT, http://www.pact.srf.ac.uk/macrame/welcome.html.

[Pea93] K. Peach et al, "The Ongoing Investigation of High Performance Parallel Computing in HEP", CERN/DRDC 93-10, 12 Jan 1993.

[Pre94] R. S. Pressman, "Software Engineering, a practitioner's approach", McGraw Hill, 1994.

[RA90] Sarah Rolph, Tammy Alfano, "Learning StateMate by Example", i-Logix Inc., 1990.

[Rum91] James Rumbaugh et al., "Object-oriented Modeling and Design", ISBN 0-13-629841-9, Prentice Hall 1991.

[Sar93] Bernard Sarosi, "Using a Network of IBM-RS/6000 Workstations to run Coarse Grained Parallel Applications", presented at SHARE Europe, 1993.

[Scha96] J. P. M. van Schalkwijk, Syllabus 5514, "Informatie theorie I", Technical University of Eindhoven, 1996.

[Schi93/1] R. Schiefer, "Visualisation of Parallel Program Execution, a case study", OOTI report, Eindhoven University of Technology, March 1993.

[Schi93/2] R. Schiefer, "Project Proposal for a Designers PhD: Data synchronised parallelism in High Energy Physics software development", Eindhoven University of Technology, November 1993.

[Schn93] A. Schneider, "Programming Parallel Computers", Ph.D. thesis, University of Geneva, 1993.

[SF95] R. Schiefer, D. Francis, "Parallelisation of an Existing High Energy Physiscs Software Package", IEEE Transactions on Nuclear Science, 1995.

[SS95] R. Schiefer, P. D. V. van der Stok, "Viper, a tool for the Visualisation of Parallel Programs", Proceedings of the EuroMicro Workshop on Parallel and Distributed Processing, IEEE Computer Society Press ISBN 0-8186-7031-2, 1995.

[Str92] B. Stroustrup, "The C++ Programming Language", Addison-Wesley, 1992.

[TU94] G. Tomas, C. Ueberhuber, "Visualization of Scientific Parallel Programs", Lecture Notes in Computer Science 771, Springer Verlag 1994.

[War95] M. P. Ward, "A Transputer based Scalable Data Acquisition System", Ph.D. thesis, University of Liverpool, 1995.

[Wor92] P. Worley, "A new PICL trace file format", Oak Ridge National Laboratory, 1992.

[ZC91] H. Zima, B. Chapman, "Supercompilers for Parallel and Vector Computers", ACM press, 1991.

[Zeb95] Zebra, entries Q100 and Q101, CERN Program Library Office, 1995.

Index

Stellingen behorende bij het proefontwerp

Viper, a Visualisation Tool for Parallel Program Construction

door René Schiefer

[1] With increased knowledge about event patterns, the conditions under which Lamport's algorithm can be used to construct a consistent run with logical clocks can be weakened and a stronger algorithm can be used.

References:
S. Mullender, "Consistent global states of distributed systems", Distributed systems, Addison Wesley, 1993.
R. Schiefer, "Viper, a visualisation tool for parallel program construction", Chapter 5, 1999.

[2] Visualisation tools are essential for dealing with parallel program tuning and debugging given the sheer complexity of the task at hand.

[3] The way towards better visualisation tools for parallel program construction lies not only in further EC funded research, but also in successful commercial exploitation of these tools. An open and flexible interface architecture is needed, permitting inter-operability between the tool and a wide range of parallel computing platforms.

[4] The use of explicit parallelism in its current form, where it is regarded as an additional level of program sophistication, complexity and thus source of error, is unlikely to be the way forward in developing high quality software and is more a symptom of immaturity.

[5] With complex IT systems, a good design rarely comes about without serendipity. Paradoxically, a good design process would more likely demonstrate a lack of serendipity.

[6] The way organisations conduct core business, shapes the way they deploy IT. Where Shell's engineering culture leads to an appreciation for IT architectures, its long-term planning and risk mitigation in the oil business is sometimes reflected in a careful approach to follow the swift pace of change in IT.

[7] "Was mit der Hand nicht geht, geht mit EDV[1] auch nicht, nur schneller" (a personal quote of my dear friend Manfred Kneip), is one of the axioms of the application of IT in business. Yet, when compiling user requirements it appears to be a non-triviality to most users.

[1] EDV : Elektronische Daten Verarbeitung

[8] The debate on the topic of the UK joining the EMU should be fought less on political ground and more based on sound economical observations such as the fact that there is a phase difference of several years between the macro-economic cycle of the UK and Continental Europe.

[9] The instability that is introduced by the globalisation of financial markets, as witnessed by the collapse of Asian and Russian economies, demonstrates the weakness of mankind in deploying its intelligence and advances in science to improve the stability of ordinary people's existence.

[10] Corporate change programmes, initiated to improve shareholder value, require a certain amount of consensus to be present for change to be implemented effectively. Paradoxically, a state of continuing profits can stand in the way of obtaining this consensus, even if these profits are below the norm.

[11] It is not an uncommon perception that, to be successful as an individual in a multinational environment such as CERN or Shell, the *speaking* of different languages is a major advantage. However, a lack of *listening* skills or understanding of foreign culture is a greater barrier for effective communication.