A Framework for Modelling and Simulating Data Flows in Distributed Computing Systems

Christoph von Praun
CERN, 1211 Genève 23, Switzerland

Abstract

This paper introduces a framework for modelling and simulating the behaviour of distributed computing systems. The framework comprises both means for abstract modelling and an implementation that allows the evaluation of abstract models by means of discrete event simulation.

The basic characteristics of distributed computing systems used for physics data processing at CERN are presented, motivating the purpose and need for modelling and simulation. As an example, the central data recording system of a current High-Energy Physics experiment is introduced.

The modelling framework provides concepts to structure relevant state information of a system. The most important factor to set up the structure of components encapsulating state information is thereby given by dynamic aspects of the system, i.e. by identified data flows. Traditional approaches to modelling often tend to focus purely on static, architectural system aspects.

All abstract concepts will be motivated and demonstrated using an example. Results of the simulation are compared to figures measured during real system operation, rating the proposed framework and its underlying modelling approach.

I. INTRODUCTION

Since the beginning of the 90ies, distributed computing systems at CERN have grown from homogeneous workstation clusters having tens of individual nodes to heterogeneous systems comprising hundreds of independent systems implemented by a wide variety of technologies.

Each individual system contributes to the overall task performed by the distributed system. Depending on its role and its resource limitations, a particular subsystem may critically influence the overall behaviour. Given different load scenarios, co-operation and dependencies among subsystems, the identification of bottlenecks is not obvious any more.

It is desirable to identify system components as potential bottlenecks already at the stage of system design. Given the uncertainties of system characteristics at the design stage, several design alternatives of the system and different load profiles have to be evaluated. As prototypes usually represent only a part of a system, the investigation of scalability to larger scale systems is an important issue. Modelling and simulation provide powerful means to meet this challenge on the basis of an abstract, theoretical approach.

A particular application of distributed computing at CERN is given by so-called Data Acquisition (DAQ) systems. DAQ systems collect physics data from High-Energy Physics (HEP) experiments, reduce the data volume and archive data to permanent storage. Those systems are characterised by quasi real-time constraints as follows: limited computing, transmission and storage resources are facing continuous high-bandwidth data-flows. The sole description of static system properties, such as the hardware configuration or steady load patterns is not sufficient to fully characterise the behaviour of such systems. Dynamic load and the distribution of data are important properties and dominate the overall behaviour.

Modelling and simulation are essential methodologies to estimate the system behaviour already at the time of system design. For that purpose, an abstract state-based modelling paradigm is proposed which has been adopted from software engineering. This leads to a visual formalism characterising the so-called conceptual model. The efficient evaluation of a model is done using an operational model [1]. The framework offers a C++ class library that allows the set-up of an operational model on the base of a conceptual model.

II. THE MODELLING MECHANISM

The following section will present a mechanism to set up conceptual models of distributed computing systems. The need to model different aspects of such systems separately is motivated and a visual formalism is introduced to denote conceptual models.

A. *Overview*

It is evident that a modelling mechanism has to account for various aspects of a system that is to be modelled. To gain flexibility in modelling, self-contained aspects should be modelled separately if possible. For the kind of systems concerned, two categories of system aspects have been identified.

The first aspect, i.e. the modelling of components, addresses system properties that are determined at system start, such as hardware characteristics, potential system states, structure and algorithms. Components represent the system state in a structured manner and control state changes.

Complementary to components, processes act as the second kind of system aspect. Processes model system dynamics, i.e. aspects that trigger state transitions on components. Both categories are modelled separately and are integrated into a conceptual model. Sections B and C discuss each category.

Traditional techniques to model systems using a state-based paradigm, such as finite state machines enhanced by an event-action mechanism, tend to encounter a considerable complexity already for models of small systems. For statecharts, a hierarchical structuring of states and concepts to express concurrent state transitions have been added [2]. The

0018-9499/98$10.00 © 1998 IEEE

mechanisms to express causal and temporal dependencies between state transitions are however still integrated and hard-coded into the state/transition model.

Formal languages such as SDL [3] or LOTOS [4] have been designed to denote specifications of distributed concurrent information processing systems. Both formalisms combine the notion of processes and state in a general way. The languages are destined as specification mechanisms and allow, being formal description techniques, a specification of behavioural system aspects in a rigorous mathematical sense. The analysis of models that base on formal specification languages is mainly done using an algebraic foundation. This very general and formal nature of such modelling approaches results in a considerable complexity of models representing general purpose computing systems concerned in this paper.

Regarding the issue of simulation, a more pragmatic approach was chosen in this paper targeting an event-based simulation mechanism. Typically, system and workload modelling are distinguished in the field of computer performance modelling and simulation [5]. The two-fold modelling approach developed in this note takes up this philosophy of separate system and workload models and puts them into a more general context. Speaking in terms of SDL or LOTOS specification languages, the notion of processes has been refined and adopted to the application domain. The component model features processes instances that persist during system operation. The process model – in the sense of this paper – denotes process types whose instances have temporary existence.

B. Component Model

The basic means to characterise state and structure of a system are given through so-called components. Components stand for self-contained, not necessarily physical, parts of a system. The self-containment assumes that the characterisation of state can be done independently from other components. State information is maintained and accessed from outside the component through an interface. The type of a component is understood as the name of the component's interface. Instances of component types are referred to as components. A component interface declares attributes and services as follows:

- Parameter attributes keep initialisation information of instances of component types. The values of parameter attributes are constant. The type vector of parameter attributes for a given type C is denoted as A_C.
- State attributes represent variables that determine the state of a component. The type vector of state attributes for a given component type C is be denoted as S_C. The values of state attributes change over simulation time with respect to services executed on the component.
- Components offer services that are initiated by processes. Services correspond to process types, which means that there is a homomorphism for each component type that maps accepted process types to services. The precise mechanism of interaction between processes and components will be characterised in section C. The execution of a service will change the values of state attributes of the component. This change happens instantaneously relative to simulation time.

Figure 1 illustrates the graphical notation of components as ovals denoting the identifier c and the type C.

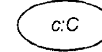

Figure 1: Notation of a component

C. Process Model

Although components are designed to be independent abstractions acting in different roles, they play as a team making up the unique overall system. This circumstance raises the need for communication between components. Speaking in terms of modelling, components have to synchronise the state information they embed. We have seen that the state of a component is maintained through services defined by the component type. The execution of a service is initiated by processes. In that sense, processes can be understood as a kind of glue between components characterising the way they depend on and interact with each other.

The Notion of Processes

A process is a sequence of service executions. Processes are modelled as objects containing information on the context and in particular on the progression of process execution. Processes are instanced from process types that declare context attributes. These attributes are initialised at the time of process creation and are updated by service executions. The domain of context attributes for a type P is denoted by K_P.

Besides initialisation information, the context information of a process comprises for example the sequence of components to traverse, the number of times the process has already initiated service execution on a particular component and the simulation time at which the process will initiate the next service.

Figure 2 introduces the graphical notation of processes for a process p of type P.

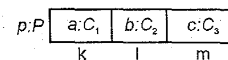

Figure 2: Notation of a process

In the figure, the process p targets three components a, b and c. The services initiated on each component are executed k, l, and m times respectively. The variables k, l, m are derived from context attributes of the process p. If a component is targeted only once, no number is denoted. The array of targets indicates several slots being capable to address components of the types C_1, C_2, and C_3 in the given order.

The flow principle of a process through components is similar to source routing within computer networks. The sequence of components to traverse can be regarded as an array whose slots carry references to the components. At the time of process creation, at least the first slot has to be initialised with a valid reference to a component. The routing array is maintained like other context attributes by the service a process currently executes. Service execution can shift the

array pointer denoting the current component one slot to the right causing the process to execute services on the subsequent component. A pointer past the routing array indicates that the process is completed. It is important to emphasise that process creation, the initiation of services and the completion of a process have zero duration relative to simulation time. Simulation time can be delayed between subsequent service executions. The term 'process execution' will refer to the sequence of service executions triggered by a particular process and the simulation time delayed in-between.

Semantics of Service Execution

Services are executed on components. The execution is initiated by a process and has zero duration relative to simulation time. Given a component of type C, the service s is uniquely determined by the type P of the process that initiated the service. The effect of execution of a service s of a component of type C on a particular process of type P, i.e. the values of its context attributes K_P, is characterised by a function f_s having the signature:

$$f_s : K_P \times S_C \times A_C \rightarrow K_P \qquad (1)$$

The effect of service execution on the executing component can be defined similarly. According to the assumption that components are self-contained with respect to their state and change of state, the execution of a service s on some component of type C is characterised as a function g_s having the signature:

$$g_s : K_P \times S_C \times A_C \rightarrow S_C \qquad (2)$$

In addition, further processes can be created through service execution. Let Π be the domain of processes of any type. The set of processes issued by the execution of a service s of a component of type C is denoted as function h_s having the signature:

$$h_s : K_P \times S_C \times A_C \rightarrow \Pi^+ \qquad (3)$$

The set of processes yielded by a function h_s is never empty and contains at least the process of type P that initiated the service.

The fact that components act as way stations and creators and of processes is illustrated in figure 3.

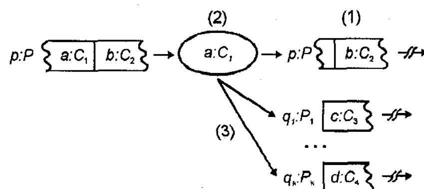

Figure 3: Illustration of service execution semantics

The numbers in the figure refer to the corresponding effects of service excution characterised by the formulas (1) to (3). Component a of type C_1 acts as way station for a process p. The process p initiates a service on component a and is forwarded after service execution to some component b. The process may delay simulation time before service execution on b starts. Besides the effect on context attributes of p (1) and the state attributes of b (2), the service execution issues processes $q_1 ... q_k$ (3) that are subsidiary processes of p. As with process p, the execution of the first service for $q_1 ... q_k$ can be delayed relative to simulation time. This is indicated by truncated arrows.

Several paradigms exist to finitely specify the infinite notion of semantics in this context. The definition of the semantic functions f, g and h can be done using arbitrary paradigms. In software engineering, a common approach to formally characterise the behaviour of components is given through finite state machines and their enhancements. The expressiveness of such an approach however limits the domains of the formulas (1), (2) and (3). For a deterministic finite state machine with output, for example, the domain S_C is limited to a finite, non-empty set of states. The domain of parameter attributes A_C would denote the initial state and K_P would enumerate the alphabet of input characters, i.e. a set of events triggering state transitions. The function f would characterise the transition relation of the state machine. The function g would be obsolete as processes are featured by input characters that carry no context information. The function h would map to the domain Π characterising sets of output characters tagged with a reference to the target state machine and the time of execution.

Another approach to characterise the functions f_s, g_s and h_s for each service s of a component, is to define each function as vectorial equation. At the first glance, this approach induces mathematical complexity. For those who aim to simulate a model using the simulation library, the implementation of such vectorial equations is done in the shape of algorithms. Given such an operational model, the mathematical equation might actually never be explicitly formalised.

Summarising, the functions f_s, g_s and h_s characterise the semantics of a service s of a component of type C. The fact that semantics are defined on the basis of functions does not necessarily guarantee determinism. Given one execution of a service s, the functions f_s, g_s and h_s are evaluated using the same set of input parameter values. Thus, for one service invocation, the order of evaluation does not affect the result. If several processes initiate services on the same component at the same time in simulation, the order of evaluation of the semantics functions f, g and h among the processes is not determined. For distinct processes, the result of the functions can depend on common input and output attributes of the domain S_C, i.e. the state attributes of the component. Other approaches to modelling timed systems, such as statecharts, suffer similar problems when defining deterministic behaviour with respect to coeval events, i.e. race conditions. This problem of simultaneity is a pure artefact of modelling and simulation and has no counterpart in real systems. There are techniques such as minimal random delays or priorities on events to overcome strict simultaneity in simulation practice. The issue will not be discussed further in this context.

Relationships among Processes

The concept of processes presented so far enables the characterisation of sequential service executions for a single

process. Several process instances can initiate service execution concurrently and independently from each other.

In real systems however, process executions can depend on each other raising the need for synchronisation among processes. The synchronisation requires logical synchronisation points within the trace of execution of a process. These synchronisation points are naturally given through service executions.

The execution of a service may involve the creation of further processes. Those processes are subsidiary processes, also called sub-processes, relative to the process that initiated the service, i.e. the originating process. This process – sub-process relationship can be regarded as a causal dependency between processes. Causal dependencies are phrased for example by "some process p will be created only if process q exists" or "some process r should start and run to completion until process s continues". The creation, i.e. the fork, of one or more sub-processes can happen in a blocking and non-blocking manner relative to the originating process.

Blocking sub-processes stop the execution of their originator until all of them are finished. In the case of non-blocking sub-processes, the execution of the originating process is resumed immediately. The graphical notation is enhanced as follows:

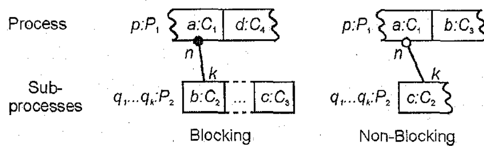

Figure 4: Notation for blocking and non-blocking sub-processes

The left-hand side of figure 4 illustrates the notation of a blocking fork. The component a issues k processes of type P_2 at service execution for process p. The new processes $q_1,...,q_k$ have a causal relationship to the originating process p, they are sub-processes of p. All k sub-processes start service execution on the component b. The fork is, as part of the execution of a service, done instantaneously. The execution of p is suspended until all k sub-processes have run to completion. The procedure, i.e. the service execution for p on component a, repeats n times until the process p initiates a service on the component d.

The mechanism provided for non-blocking forks, as shown in figure 4 on the right, is similar. At each service execution for process p, k sub-processes $q_1,...,q_k$ are forked. The originating process p however, immediately continues execution. After the service for process p has been executed n times on component a, the process targets the next component b.

So far, dependencies among processes were expressed in an originator – sub-process relationship featuring a causal dependency. In addition, temporal dependencies between processes at the same level of causality should be expressed. Temporal dependencies can typically be phrased as postulations of the shape "process p must not start before process q has ended" or "process r starts only after process s has been launched. Figure 5 illustrates the notation of concurrent and sequential sub-processes.

Any temporal dependency among processes is controlled by a common originating process, denoted as process a in the figure. Two kinds of temporal relationships are foreseen, concurrent and sequential execution of processes. The initiation of concurrent processes is done by an originating process in either blocking or non-blocking manner. The upper half of figure 5 illustrates concurrent processes that are forked in blocking (upper left) and non-blocking manner (upper right) by their common originating process. The service execution of process p in component a issues k and l instances of sub-processes of type P_2 and P_3 respectively. As the execution of the sub-processes is initiated blocking (upper left) to p, another of the n iterations will be started only if all processes $q_1,...,q_k$ and $r_1,...,r_l$ have completed. Similarly, concurrent processes can be forked in a non-blocking manner by the originating process (upper right). Process execution of p resumes immediately after k sub-processes of type P_2 and l sub-processes of type P_3 have been forked at the same time in simulation. This procedure repeats n times.

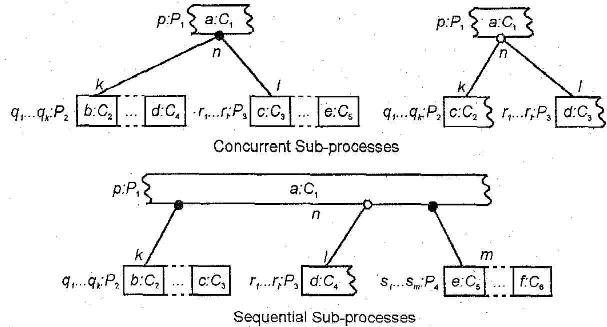

Figure 5: Concurrent and sequential sub-processes

The lower part of figure 5 illustrates sequential temporal dependencies among processes. The processes $r_1,...,r_l$, for example, are not started until k processes $q_1,...,q_k$ of type P_2 have completed execution. Similarly, the processes $s_1,...,s_m$ will not be started before l instances of processes $r_1,...,r_l$ of type P_4 were forked. The procedure, i.e. the sequence of three service executions on component a, repeats n times.

D. *Hierarchical Structuring of Models*

The composition of components and processes integrated static and dynamic aspects of a system into a conceptual model. Such a self-contained model is understood as

- a set of component type specifications,
- a set of process type specifications,
- a set of component instances provided with initialisation information for their parameter attributes. Components are instanced from the given component types and persist throughout the whole system lifetime.
- a set of initial processes featuring external stimuli. The processes are instanced from the given process types. They trigger initial service executions and their existence is usually intermediate.

The notion of a model in the given sense resembles a reactive system [3] similarly to components themselves. A generalisation of the component concept leads to so-called hyper-components. Contrary to ordinary components, hyper-components define in their scope attributes and behaviour in terms of further components and processes. The interaction between processes and hyper-components happens through an interface as for ordinary components.

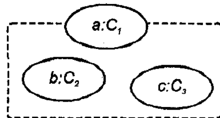

Figure 6: Notation of a hyper-component

Figure 6 illustrates a hyper-component a having the interface named C_1. Further components of the types C_2 and C_3 with identifiers b and c, are contained in the scope of the hyper-component. From outside the hyper-component, the components b and c, i.e. their services, are accessible only indirectly through a. This means that services of a initiate sub-processes on the components b and c.

The concept of hyper-components provides a seamless extension of the notion of components and enables a powerful mechanism of hierarchical and structured modelling.

III. The Simulation Library

Corresponding to the theory of components and processes, a programming library has been implemented that instruments the concepts presented. The library design is based on an object-oriented foundation, the components and processes are reflected by abstract base classes. For a particular model, the base classes are refined through inheritance and instanced to actual components and processes. The library implements a process scheduling mechanism that enables the initiation of services at certain points in simulation time. The library is implemented in C++, recent techniques such as the use of the Standard Template Library (STL) guarantee efficiency and a powerful interface to the modeller [7]. The use of the library requires C++ skills. Heuristics for the transition from the abstract conceptual model to the C++ implementation have been conceived.

IV. Example: Modelling a Central Data Recording System

The modelling theory and the simulation library were applied to model the back-end of a DAQ system for one of CERN's experiments, namely the NA48 experiment. At first, the system configuration and the application requirements are characterised. A short sketch of relevant parts of the component and process model will be given. Finally, simulated and measured performance figures are illustrated, analysed and compared.

A. Overview on the Application

In the case of a Central Data Recording system (CDR), the collection, analysis and storage of physics data is done simultaneously to the running experiment. For the NA48 experiment, a continuous stream of data of about 15 MB/s is at first stored on a large pool of parallel file systems. CPU-intensive processes read this data, analyse it and write results back to file systems. Finally, raw and analysed data are dumped to tape storage. The kind of application is characterised by high disk and network I/O and extensive computing demands.

The implementation of the CDR system uses about 32 nodes of a scalable parallel computer, namely a Meiko CS2 system. Each node is an independent symmetric multi-processing unit comprising two CPUs and 128 MB of main memory. All nodes are interconnected by a switched high-speed network capable of 35 MB/s duplex speed per pair of nodes. Disks are attached to the system in stripes of 4 or 5 nodes providing a logical NFS based parallel file system (PFS). The storage capacity spread over several of those logical parallel file systems amounts to approximately 1 TB. A more detailed report on the system configuration and the CDR application is given in [6].

The aim of the modelling and simulation is to evaluate the overall performance of different configurations of the contingent of 32 nodes. One configuration would partition nodes in a disjoint manner, dedicating each node to a particular task like disk I/O or CPU services for physics analysis processes. Another feasible configuration would disperse file systems further and run analysis processes also on disk servers. For the purpose of the study, a detailed model of the overall system has been developed using the framework.

B. A Partial View of the Model

In the following, parts of the component and process model for the CDR system of the NA48 experiment will be illustrated. Due to the limited scope of this paper, the precise interfaces of object types will not be specified. The interaction between components and processes however will become obvious through the given figures.

Several components have been conceived to model parallel file systems and computing resources. Paralleling the hardware configuration of the real system, the component model conceives one component per node. The type of component for nodes is *NODE*, instances have the identifiers *cerncs2-20* and *cerncs2-21* in figure 7. The type *NODE* itself is a hyper-component. Thus, a self-contained sub-model determines the behaviour of a node facing I/O or CPU loads. This sub-model comprises instances of the component types *CPU* and *IO*. Moreover, the component type *PARALLEL_FILE_SYSTEM*, although physically dispersed among several nodes in the real system, models one parallel file system as a single logical unit.

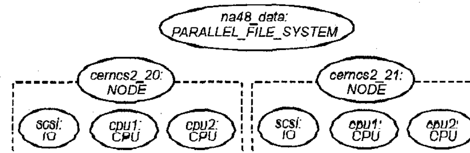

Figure 7: Component model for computing nodes and a parallel file system

The mentioned components depend on each other for fulfilling their services. To illustrate the dependencies, the process instances for a particular read access to the parallel file system is sketched in figure 8. The read from the file system is issued by a process of type *READ_PROCESS* that initiates a service on the instance *data_40* of type *PARALLEL_FILE_SYSTEM*.

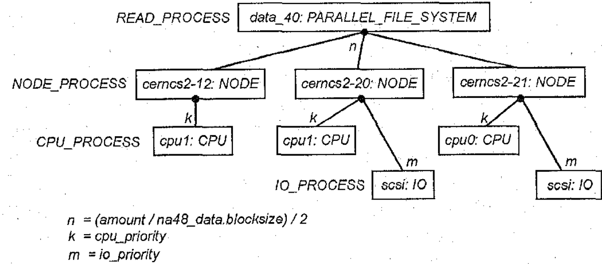

Figure 8: Process instances for a read access to a 2-striped parallel file system

The design of the parallel file system implements data access in an equally distributed manner among the nodes the file system is attached to. In the example, the file system is assumed two-striped and two nodes, *cerncs2-20* and *cerncs2-21*, each fulfil half of the computing and data transfer demands of the read process. The computing requirements needed on the client side, i.e. the node *cerncs2-12*, are accounted for in sub-processes that each run concurrently to the corresponding processes on the server nodes. The synchronisation between the file system stripes on the server side and the client side happens on the base of equal charges of data which size depends on the blocksize of the *PARALLEL_FILE_SYSTEM* instance. This fine granularity for modelling access to the parallel file system enables an interleaved execution of several processes accessing one file system instance concurrently. For each data block transferred from a stripe of the parallel file system, a process is created initiating service execution on the component that models the node the file system stripe is attached to. This component has the type *NODE* and is indeed a hyper-component. The hyper-component delegates the demands of a *NODE_PROCESS* to a particular *CPU* instance, e.g. *cpu1* for the *NODE* instance *cerncs2-20*, and an *IO* instance named *scsi*. For that purpose, processes of type *CPU_PROCESS* and *IO_PROCESS* are forked. The number of forked processes per process type, i.e. the values of k and m, depend on the priority and potential multi-threading of a process. Depending on the state of the components *cpu1* and *scsi*, i.e. the current workload, the processes delay a certain amount of simulation time until they complete. The principle of sub-processes instrumented by the fork mechanism ensures that blocked originating processes of type *NODE_PROCESS* and *READ_PROCESS* continue and finish after a certain number of sub-processes are completed.

C. Analysis of Simulated and Measured Figures

Before simulation and analysis could be done, the conceptual model, which was partly sketched in the previous paragraph, has been implemented as operational model using the simulation library. Performance figures have been measured on the real system and served to parameterise and calibrate the operational model. Examples of performance figures are the required amount of server CPU system time per MB read from a parallel file system or the relation of scheduling priorities between user and system tasks. In the following, partial aspects of analysis and simulation results are presented and compared to measured figures.

The analysis presented in the context of this paper will be done using so-called Kiviat Graphs [5]. Kiviat Graphs illustrate an even number of dimensions in a circular graph. Adjacent dimensions are arranged in a way that one represents the figures of an HB metric (higher is better) and the other one an LB metric (lower is better). For the example, HB metrics are given as the rate of disk I/O and the share of user CPU. LB metrics are horizontally sketched and refer to the share of system CPU and the share of CPU in idle or wait I/O state.

Related to the analysis using Kiviat Graphs is the calculation of a summary figure, the so-called figure of merit (FOM). This figure expresses the averaged LB and HB metrics and summarises how efficiently the capabilities of a system are exploited. As all dimensions are equally accounted, the interpretation of this metric should be done with care. The precise calculation is given in [5]. The FOM figure peaks at a value of 100, increasing values indicate a more efficient system usage.

Both Kiviat Graphs and the FOM figure consider a snapshot or average figures of a system serving certain tasks. In the case of the analysis done for the computing facilities of the NA48 experiment, typical scenarios have been defined, assuming that nodes were either dedicated to act as disk server, CPU server or as combined CPU/disk server. For the three scenarios presented in the following figures, the tasks of a node, i.e. the demanded disk I/O rate and the number of physics analysis processes per node, were determined. The illustrated dimensions assume 100% CPU usage for two saturated processors on a node and 100% disk I/O for a transfer of 8 MB/s per node. Corresponding to the conventions met for the dimensions of axes in terms of HB and LB metrics, a balanced system is considered to have a slant Kiviat Graph covering a wide range of the vertical axis.

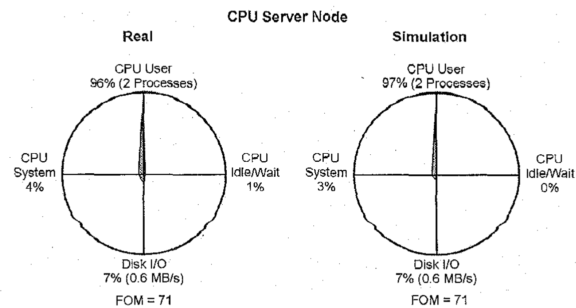

Figure 9: Kiviat Graphs for a CPU server node

On CPU server nodes, typically two single-threaded user processes are executed. Those physics analysis processes are CPU-intensive and have a working set of 35 MB of main memory. Each analysis process requires disk I/O at a rate of 0.3 MB/s. This scenario is illustrated in figure 9.

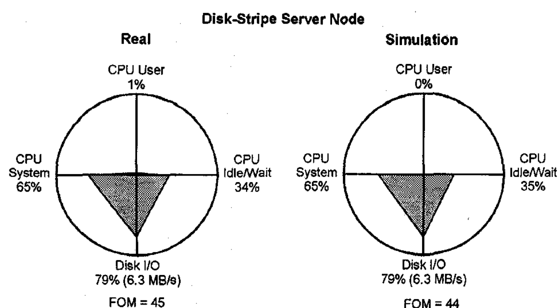

Figure 10: Kiviat Graphs for a disk server node

A common scenario on disk servers in a configuration of separated disk and CPU servers is given in figure 10. The I/O rate demanded per server node of a file system stripe is assumed 6.3 MB/s.

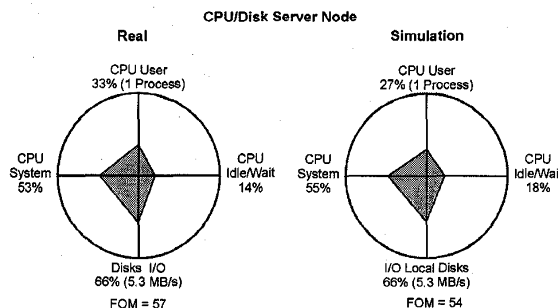

Figure 11: Kiviat Graphs for a CPU/disk server node

For the combined CPU/disk server scenario sketched in figure 11, a single physics analysis process and a demanded I/O rate 5.3 MB/s have been assumed.

The result of the analysis showed that additional user processes on disk servers could increase the overall usage of resources on the server without compromising the overall disk I/O performance.

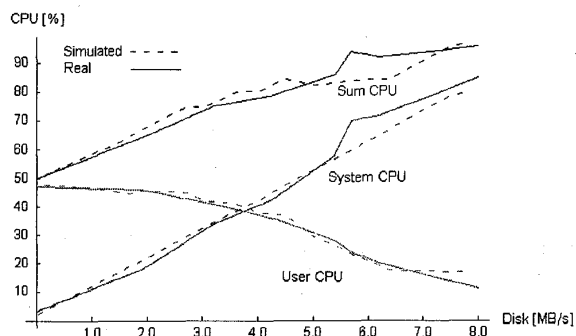

Figure 12: Comparison of simulated and real figures on a CPU/disk server configuration

For the three scenarios illustrated, the processes i.e. the demanded disk I/O rate and the number of user processes per node were fixed. The comparison between real and simulated figures however has to consider a variation in the demands of tasks scheduled on a node. Figure 12 illustrates a variation of disk I/O demands for a combined CPU/disk server and the resulting CPU utilisation. One CPU-intensive physics analysis process was permanently executing on the node, while the rate of disk I/O was varied. The figure indicates that simulated figures deviate less than 15 % from real figures.

V. CONCLUSION

The proposed approach faces the requirements of modelling and simulating distributed computing systems by conceiving a theoretical concept for modelling and a corresponding implementation part that allows the simulation of models.

One of the important characteristics of the framework is to separate static and dynamic aspects of a system. Static issues address qualities of a system that remain invariant during its operation. Dynamic characteristics of a system relate to processes that perform the actual evolution of the system during its operation. The general concept of state is split up into component state and process context. Several techniques have been developed to relate processes regarding their causal and temporal dependencies.

A part of the data acquisition system of the NA48 physics experiment has been modelled using the theory of components and processes. The precise results yielded by the simulation of the operational model helped to tune the final system configuration. Following up this positive experience further development and application of the framework is foreseen in the field of modelling and simulation of large-scale PC farm architectures.

VI. ACKNOWLEDGEMENT

For various discussions, their advice and practical experience, I especially thank Les Robertson and Frederic Hemmer.

VII. REFERENCES

[1] J. Banks, J.S. Carson, B.L. Nelson, "Discrete-Event System Simulation", New Jersey: Prentice Hall, 1996.

[2] D. Harel, "Statecharts: A Visual Formalism for Complex Systems", *Science of Computer Programming 8*, Elsevier Science Publishers B.V., 1987.

[3] CCITT, Recommendation Z.100, "Functional Specification and Description Language (SDL)", *Blue Book*, 1988.

[4] ISO, International Standard ISO 8807, "Information Processing Systems – Open Systems Interconnection – LOTOS – A formal description technique based on the temporal ordering of observational behaviour", 1989.

[5] R. Jain, "The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modelling", John Wiley & Sons, 1991.

[6] E. McIntosh, B. Panzer-Steindel, "Parallel Processing at CERN", *HEPIX Caspur Rome*, CERN, October 1996.

[7] D.R. Musser, A. Saini, "STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library", Addison Wesley, 1996.