THESIS
3
2000

LIBRARY
Michigan State
University

This is to certify that the

dissertation entitled

DESIGN AND ENGINEERING OF COMPLEX REAL-TIME SYSTEMS

presented by

Aleksandar M. Bakic

has been accepted towards fulfillment
of the requirements for

Doctoral degree in Computer Science & Engineering

Major professor

Date May 3, 2000

MSU is an Affirmative Action/Equal Opportunity Institution

O-12771

PLACE IN RETURN BOX to remove this checkout from your record.
TO AVOID FINES return on or before date due.
MAY BE RECALLED with earlier due date if requested.

DATE DUE	DATE DUE	DATE DUE

11/00 c:/CIRC/DatesDue.p65-p.14

Design and Engineering of Complex Real-Time Systems

By

Aleksandar M. Bakić

A Dissertation

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

Department of Computer Science and Engineering

2000

Abstract

Design and Engineering of Complex Real-Time Systems

By

Aleksandar M. Bakić

Complex real-time systems are emerging parallel or distributed, heterogeneous computer systems with many disparate constraints and requirements. Their subsystems and components are designed using appropriate, multiple models from real-time scheduling theory, resource allocation and quality-of-service managements schemes. Issues in the design, engineering and deployment of complex real-time systems addressed by this research include problem-solving approaches for finding satisfactory values of system parameters according to the real-time models and their integration; dynamic reconfiguration; instrumentation; and on-line performance analysis and visualization.

Four main objectives of this research resulted in proof-of-concept contributions. A compiler-based approach to design and engineering of complex real-time systems was designed, implemented and evaluated that represents a systems engineering framework particularly suitable for this systems domain. A distributed instrumentation system kernel was developed and evaluated in order to investigate performance and

portability issues of the instrumentation of parallel and distributed, heterogeneous systems. A comprehensive on-line performance analysis and visualization technology, for the same domain and with an emphasis on complex real-time systems, was developed that incorporates common properties of extant tools and provides a basis for advanced on-line performance analysis and visualization tools. A mainly automated technology that combines the preceding results was developed in an attempt to integrate systems engineering and on-line performance analysis and visualization in a way that facilitates on-line reconfiguration of complex real-time systems.

© Copyright 2000 by Aleksandar M. Bakić

All Rights Reserved

To my parents

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Matt W. Mutka, for his continued guidance and support during my research. I have been fortunate to have an experienced researcher and talented pedagog to supervise me. He made the pursuit of my Ph.D. an enjoyable and invaluable professional experience.

I am thankful to Dr. Diane T. Rover for providing me with my first research opportunity and continuing to support me as a Ph.D. student. Her ideas and constructive critiques in our collaboration made a difference so many times.

I thank the other members of my guidance committee, Dr. Anthony S. Wojcik and Dr. James Stapleton, for their helpful comments and suggestions.

I wish to thank Dr. George C. Stockman again for extending me an offer to come to the MSU in the first place. I am indebted to Dr. Lionel M. Ni for valuable lectures in practical, and Dr. Eric K. Torng in theoretical aspects of Computer Science; as well as to other MSU professors who taught me advanced and exciting.

Many of my fellow students helped me and collaborated with me over the course of study. I would like to thank them all, especially Peter C. Wong, Paul A. Reed, Hugh M. Smith, Wenting Tang, Kuk-jin Lee and Abdul Waheed. I would also like to

thank the CSE and ECE departments for the excellent facilities, and technical and administrative support.

I am grateful to my family for their love, support and patience. Special thanks to my wife Vera for her love and understanding, and to our little daughter Jovana for bringing joy to our lives.

TABLE OF CONTENTS

List of Tables

List of Figures

Chapter 1

Introduction and Motivation

A broad variety of different software systems—ranging from small, embedded controllers with microsecond response times to large, heavyweight systems with response times of seconds or even minutes—have been designated "real-time." For both a power plant control system and a global air-ticket reservation system, as well as an abundance of other diverse systems, certain common characteristics can be summarized in the following definition by Selic [100]:

Definition. *A real-time system is a software system that maintains a timely and ongoing interaction with its environment.*

Unlike other system resources in conventional software systems, such as memory and processing power, the (real) time cannot be controlled; only its progress can be measured. The timeliness property of a software system is a function of the timeliness of individual activities (or tasks) that the system supports. A real-time system achieves its timeliness by managing its resources accordingly. Jensen further defines "real-time-liness" of a task [62]:

Definition. *A task is real-time to the degree that its completion timeliness predictability is part of its logic.*

The ongoing interaction property of a software system relates to a quality that is often referred to as *reactivity* [77]. Translated from a logic-theoretic framework to English:

Definition. *A reactive system persists across some interval of time (possibly indefinitely) during which it responds to inputs as they occur.*

In this setting, the dominant feature of a real-time system is its structure, rather than its function. To reason about the above two properties of a real-time system, its structure must be fixed, while the function may be modified. Consequently, the focus of software system design shifts from the traditional, algorithmic to structural approaches. The structural approaches range over different domains. On one end, real-time scheduling theory is mainly concerned about low-level issues, such as physical resource allocation. On the other, formal software-engineering methods address more abstract real-time requirements, such as timing relations among different application states.

1.1 Focus of the Dissertation

Complex real-time systems are emerging parallel or distributed, heterogeneous computer systems with many disparate constraints and requirements. For example, an embedded complex real-time system may possess limited system resources (for computation, communication and I/O) that need to be shared by many of its components.

It must perform certain tasks in a timely fashion under possibly large variances in the amount of data and events that it processes. If the system or some of its components are critical for the operation of its embedding system and/or environment, they must be made fault-tolerant. The dynamics of the environment may span from static models to stochastic ones, to unpredictable ones.

Consequently, components and subsystems of a complex real-time system are designed using appropriate, multiple models from real-time scheduling theory, resource allocation and quality-of-service management schemes. Issues that arise in the design, engineering and deployment of complex real-time systems include the following.

1. Detailed models are complex and require exhaustive search through the design space to find values of model parameters for which the system satisfies imposed requirements. Engineers of complex real-time systems often cannot afford this, and use fast heuristics or simple, ad hoc schemes instead.

2. As the above mentioned models and schemes may vary greatly in their assumptions and goals, it is hard to analyze how they interact when the components and subsystems are to be integrated. The integration further increases the complexity of the system.

3. One design solution most likely cannot satisfy the requirements when the system parameters vary significantly and dynamically. In a dynamic system reconfiguration, the quality of a new design solution is almost always traded for the speed of the reconfiguration.

4. System performance data needed to decide whether and what kind of dynamic reconfiguration should be performed, has to be collected and analyzed on-line. In integrated complex real-time systems with end-to-end requirements, justified decisions can only be made using global information.

5. To collect the performance data from a distributed, heterogeneous real-time system, a portable distributed instrumentation system (IS) is needed. It has to be designed with low real-time intrusion and flexible performance in mind.

6. Although the performance data can be analyzed completely automatically, it is often required that a human operator has insight into the system performance and makes reconfiguration decisions. To present system performance information at a high level, a performance visualization technology is crucial.

Problem Statement. *The central focus of research presented in this dissertation has been a unifying, extensible, compiler-based framework for the design and engineering of complex real-time systems. Its novelty is manifest in a comprehensive application of a constraint programming technology in model-based design and integration of complex real-time systems. In addition, the framework lends itself to further extensions, such as a novel integration of on-line performance visualization and dynamic system reconfiguration. In this broader context, the dissertation research foci are in the areas of distributed system instrumentation and on-line performance visualization, aiming at improved testing and deployment support for complex real-time systems.*

The unifying property of the framework is related to the goal of supporting disparate real-time models of system components and their integration. The compiler-based property is aimed at allowing users of the framework to specify complex real-time systems at a high level, and have a knowledge-based compiler that would handle the complexity behind high-level specifications. The extensibility property applies to both the front and back ends, and is achieved through an extensible syntax and add-on modules that cooperate in constructing a model of the whole complex real-time system. The framework supports solving for the system parameters using multiple, controllable problem-solving approaches. It also integrates an on-line performance visualization technology and facilitates creating real-time model- and system-specific performance visualizations. This integration is especially suitable for visualization-driven on-line system reconfiguration.

The general problem and solution of this dissertation should be regarded from a systems engineering point of view. As defined in [55], systems engineering is a discipline that develops and exploits structured, efficient approaches to analysis and design to solve complex engineering problems. Its focus is on methods to solve problems, not the solution of the problems per se.

In the following section, the motivation is discussed for the research in the areas of the design and engineering of complex real-time systems, distributed system instrumentation, and on-line performance visualization, respectively.

1.2 Motivation

In [104], Stankovic concludes that, despite extensive results in the area of real-time scheduling in recent years, the state of the art still provides piecemeal solutions. There are many realistic issues that have not yet been addressed in an integrated and comprehensive manner. New scheduling approaches should be analyzable and comprehensive enough to simultaneously handle multiple characteristics of real-time systems, such as: task preemptiveness, periodicity, importance, and grouping; precedence, placement, and end-to-end timing constraints; communication, resource, and fault tolerance requirements; tight and loose deadlines, and normal and overload conditions. The solution should be integrated enough to handle interfaces between

- CPU scheduling and resource allocation,
- I/O scheduling and CPU scheduling,
- CPU scheduling and real-time communication scheduling,
- local and distributed scheduling, and
- static scheduling of critical tasks and dynamic scheduling of essential and non-essential tasks.

These issues are too complex to be addressed all at once, and it will probably take years of researchers' work before first comprehensive and tightly integrated real-time models are devised. However, there are two constituents of a solution to the general problem: one is the real-time theory (including scheduling and formal methods), and the other is systems engineering. Tools are needed that will bridge the gap between theory and practice in an efficient manner.

1.2.1 Design and Engineering of Complex Real-Time Systems

An idea behind this dissertation that has led to the choice of the main parts of the compiler-based framework—the target language and compiler back-end—is that a complex real-time system could be regarded as a complex, highly non-linear network (or circuit), seemingly analogous to those from, e.g., control theory. (Although the similarity comes from non-linear relations between interconnected components—in particular, the way changes in operation of an element affect the operation of other elements—it is difficult to see whether analogues of the control theory approaches can help with solving real-time computer system problems. Almost the opposite, real-time control systems, are common nowadays [14]. There has been theoretical research [110] in applying the control theory to proving the stability of self-stabilizing distributed algorithms. However, no real-time properties of algorithms or systems have been analyzed.) In such a network, real-time activities are performed according to real-time models, across interconnected bounded hardware resources. Relations among the activities and the resources, within subsystems and among the subsystems, are relatively simple and best termed as time and resource constraints expressed via equations and inequalities of the real-time models. This common underlying property of real-time models has been a major motivation for adopting a constraint-programming approach to solving complex real-time system problems. To solve such a network means to solve for unknown real-time model parameters and perform the resource allocation. This inevitably requires combinatorial search in general.

A new class of programming languages combining the declarativity of Logic Programming with the efficiency of constraint solving is Constraint Logic Programming (CLP). New application areas, including combinatorial search problems such as scheduling, planning or resource allocation can now be tackled, which were intractable for logic programming [36]. The declarativity of a Prolog-like CLP language provides an elegant way of expressing the disparate requirements of complex real-time systems in an integrated manner.

Among the research foci proposed by a group of leading experts for real-time systems [103] is the development of requirements and internal documentation which can state real-time requirements precisely and can be used for maintenance and inspection of safety critical systems. Specifying a complex real-time system means specifying its structure and relations among individual components and subsystems, accompanied with real-time semantics. This task can be broken into two orthogonal ones. First, the structure (such as the breakdown of computation and communication onto tasks and messages) and basic relations (such as resource allocation) are independent of the heterogeneity present in complex real-time systems that primarily comes from disparate real-time models. Their specification can be supported by a fixed syntax. Second, each real-time model defines certain semantics, and some semantics are attached to the rules of integration of two different real-time models. These rules can be viewed as a variable, model-specific syntax.

The Standard Generalized Markup Language (SGML) [42] is a meta-language and information infrastructure for structured documents and specifications that has been used, among many other applications, for maintaining technical documentation. It

allows for explicitly specifying the structure of a document and the relations among its elements, while the checking of the document semantics is performed by tools called SGML applications. The motivation to use SGML for the compiler front-end comes from an idea that a complex real-time specification can also be viewed as a multi-purpose specification document. In other words, the compiler front-end is an SGML application.

1.2.2 Distributed Instrumentation Systems

Besides specification and problem-solving tools addressed in the previous subsection, tools are also needed that can provide feedback from tested and deployed complex real-time systems. The feedback in the form of collected system performance data can be analyzed on-line or most-mortem, and is necessary for validating the design assumptions. Furthermore, the performance data collected and analyzed on-line can be further fed to a problem-solving tool, in order to find new values of system parameters that might better handle a current situation imposed onto a deployed system by the environment. It should also be possible to enforce the new parameters onto the running target system. Altogether, a distributed instrumentation system, which can be used for collecting performance data in real time and without significant intrusion on the target complex real-time system, is necessary. Since it would have an infrastructure to link the real-time system performance data and the problem-solving tool in one way, reusing the infrastructure for steering the target system (i.e., enforcing the new parameters) is possible. Finally, the heterogeneous and still evolving

nature of complex real-time systems prevents making most of detailed design and implementation decisions of such a distributed IS at once and in advance. Hence, the needed IS should rater be designed and seen as a kernel than a final and complete implementation.

Designers and users of parallel and distributed systems have applied a variety of monitoring methods and instrumentation techniques to gather information for testing, debugging, and analyzing performance and optimizing systems [112]. These methods and techniques require development of systems for instrumentation data collection, management and analysis, which are themselves distributed systems. A distributed instrumentation system is typically specialized to an application domain and/or computing environment. Moreover, the distributed nature often makes it harder to use and adapt. A significant investment in time and effort may be needed to understand the IS implementation sufficiently to port and/or configure it for another application and/or environment.

An off-the-shelf distributed IS that is robust, portable and flexible would benefit both designers and users of a wide range of parallel and distributed systems. It would allow them to instrument and begin monitoring their system rapidly, and as needed subsequently, to optimize or extend the IS for their environment. High performance and openness of the IS implementation are equally important for its success as general-purpose systems software.

Obviously, the requirements of a distributed IS for instrumenting and steering complex real-time systems are broad, and even exceed those for instrumenting conventional parallel and distributed systems. A viable approach to the design and

implementation of such an IS could be one of a portable and flexible distributed IS kernel, augmented with features that are needed in the complex real-time system domain. One such IS feature, for example, is being able, to a certain degree and with cooperation with the operating system, to schedule its computation and communication activities in a way that reduces its intrusion, in the real-time sense, on the target system.

1.2.3 On-Line Performance Analysis and Visualization

With the advent of computer technology, systems are becoming more and more complex and, at the same time, there is more and more computing power available and needed for supporting the systems themselves, in various forms of on-line analyses. One large class of the on-line analysis deals with the system performance, trying to summarize and/or explain various performance metrics, such as the efficiency of resource usage or quality of service provided for higher-level activities. Advanced analyses attempt to diagnose performance problems and suggest actions that may lead to the performance improvements. In the domain of complex real-time systems, the on-line analysis places emphasis on different measures of the real-time-liness, both hard and soft, individual and collective ones.

Performance analysis and visualization (PAV) tools are crucial components of an effective development cycle, as well as deployment, of parallel and distributed applications. On-line PAV is even becoming necessary for the latter. Since the amount of performance data to be analyzed and visualized increases with the size of a target par-

allel/distributed application, on-line PAV itself should be distributed. Heterogeneous systems, in addition, need PAV tools that provide flexible integration and configuration support for heterogeneous performance data. Extant generic and library-specific PAV tools for parallel/distributed systems can cover only low-level performance aspects, provided that the target systems fit into their generic schemes and/or use specific libraries, such as PVM [32] and MPI [31]. A wider range of performance aspects, at multiple levels, global and local, are needed to capture and visually explain the behavior of a heterogeneous system.

As the performance data are gathered from different subsystems and components of a complex real-time system by a distributed IS, they need to be analyzed according to general, but also model-specific real-time-liness criteria. One example of model-specific criterion is that in all sequences of n invocations of a periodic activity, at least m ($m \leq n$) must meet their deadline [15]. This requires a PAV tool that can be extended to support new analyses. There are also real-time quality-of-service metrics of interest that have a more complex semantics than, for example, the just mentioned m-out-of-n criterion. Such metrics are often used for applications running atop of systems that provide generic real-time support. In these cases of vertical real-time integration, automated analyses that are comprehensive and helpful are hard to devise, and it may be necessary to provide visualizations of simpler performance metrics for a human operator to analyze ad hoc instead. Finally, complex real-time systems are supposed to guarantee a degree of aggregate real-time-liness through the integration of its subsystems and components. However, if certain temporary conditions, such as overloads, violate the design assumptions, the guarantee is voided.

If the assumed real-time-liness is found to have deteriorated, it may be safer to treat the situation as unpredicted than to analyze it automatically. Again, the human operator needs a potentially large amount of performance information presented in the form of visualizations in order to make a dynamic reconfiguration/recovery decision.

Altogether, on-line PAV tools can greatly benefit the testing and deployment of complex real-time systems. If well integrated with a suitable model-based problem solving framework, it would extend the latter from a static to a dynamic one, and make it achieve even higher level of abstraction (from textual specifications to visualizations) provided for the system designers and users.

1.3 Research Objectives, Activities and Contributions

This section states the objectives of the research presented in the dissertation. The statements are followed by the description of activities undertaken towards meeting the objectives, and a summary of contributions that have resulted from the activities.

There are four research objectives:

1. To devise a systems engineering framework particularly suitable for the design and engineering of complex real-time systems. This includes a systems specification language and multiple problem-solving approaches, as well as an automated mapping between them.

2. To investigate performance and portability issues of the instrumentation of parallel and distributed, heterogeneous systems. The performance issues include real-time properties relevant to the instrumentation of complex real-time systems.

3. To discover common properties of extant on-line PAV tools for parallel and distributed, heterogeneous systems, as well as gather desired properties of the on-line PAV viewed as a real-time middleware. Using the findings, to devise a comprehensive PAV framework as a basis for advanced extensions.

4. Finally, to integrate the systems engineering and on-line PAV frameworks in a way that facilitates on-line system steering/reconfiguration of complex real-time systems. The integrated approach should be semi-automated, and allow algorithmic and visual analyses of instrumentation data received from the target system to utilize problem-solving tools.

The activities in pursuit of the above objectives were carried in the following order, with certain overlappings:

1. The first version of an object-oriented, on-line PAV software was designed and implemented.
2. The CLP technology was chosen as appropriate and promising for solving design and engineering problems of complex real-time systems.
3. A simple distributed instrumentation system was designed and developed.
4. Several real-time models and small test systems were used for determining the power of a publicly available CLP tool. The CLP code was either written manually or generated by utility tools.

5. The distributed IS was made more portable, augmented with performance tuning options and reconceived as an extensible IS kernel.

6. The first version of a complex real-time system specification language and its translator to CLP were designed and implemented.

7. The on-line PAV software was vertically extended into a component-based technology, and supported by a high-level language and compiler.

8. A new version of the systems specification language was introduced together with an extensible compiler. The first complex real-time system was modeled.

9. The distributed IS kernel was evaluated, and then extended to provide instrumentation control and steering.

10. The on-line PAV technology was ported, in addition to the Unix/X11 domain, to the Java/WWW domain.

11. A real-time test-bed and client-server system were prepared for evaluating the three constituents (the systems engineering framework, the on-line PAV technology, and the distributed IS kernel) of the integrated approach together.

12. The compiler for the complex real-time system specification language was rewritten and extended to generate system-specific code that is input to the on-line PAV technology.

This research has made four notable contributions to the state of the art in the design and engineering of complex real-time systems, and instrumentation and on-line performance analysis & visualization of a wider range of parallel and distributed, heterogeneous systems:

1. **A compiler-based approach to design and engineering of complex real-time systems.** Based on a comprehensive usage of the CLP technology, this contribution addresses the first objective and problem statement by (1) enabling high-level specifications of complex real-time systems that place emphasis on the system structure and real-time models involved; (2) automated handling of model-specific details, including the integration of different real-time models;

and (3) providing multiple problem-solving approaches, such as finding optimal solutions and repairing partially correct solutions.

2. **A portable and flexible distributed instrumentation system kernel.** This contribution addresses the second objective and part of the problem statement by providing (1) a flexible and robust tool for instrumenting complex real-time systems; (2) base for developing elaborate, domain-specific distributed ISes; and (3) test-bed for experimenting with IS managing polices and IS-specific distributed algorithms.

3. **An on-line performance visualization technology.** Centered around an object-oriented, portable and distributable, adaptive, system- and application-specific on-line performance analysis and visualization framework that supports rapid prototyping, this contribution addresses the third objective and important part of the problem statement related to the dynamic reconfiguration/steering of complex real-time systems. The technology extends the core framework with (1) a PAV-specific architecture and very high-level language designed atop of it; and (2) component-based approach to development of PAV tools; which together form (3) a base for automated and advanced PAV systems, such as the last contribution in this list.

4. **An integrated approach to real-time system design and on-line PAV with steering.** This contribution extends the compiler from the first contribution with support for generating code that is treated as input to the PAV technology of the third contribution; it also utilizes the distributed IS kernel

from the second contribution to instrument and steer complex real-time systems. The fourth objective and problem statement are addressed by a novel, integrated technology that tightly links a target complex real-time system, its specification, an appropriate PAV, and PAV-driven and model-based dynamic reconfiguration; and does so in a mainly automated manner.

Overall, all the contributions were supported with a proof of concept. Whenever the time and technical conditions permitted, performance evaluations were carried out additionally to support the contributions.

1.4 Organization

Chapter 2 discusses background and related work. Several approaches to the design of distributed real-time systems are described first. Constraint Logic Programming is introduced and several works that use it as support for the specification and verification of real-time systems are described next. They are followed by related work on checking of explicit timing constraints. Issues in distributed instrumentation are discussed, and an overview of several distributed ISes is presented. The chapter ends with a discussion on on-line performance analysis and visualization (PAV).

Chapter 3 presents the compiler-based approach to design and engineering of complex real-time systems, the central focus of the dissertation. First, the Real-Time System Markup Language (RTSML) is described. An example complex real-time system is presented next, followed by the process of compiling its RTSML specification to CLP. Salient characteristics of the compiler back-end and the descriptions of

two real-time model-specific modules are presented. Results of experiments with the example complex real-time system are analyzed.

Chapter 4 presents a portable and flexible distributed instrumentation system kernel called BRISK. Along with the description, approaches that provided IS performance gains are discussed. The objectives of BRISK, approaches taken in its design and implementation, the architecture and implementation details, and results of evaluating its performance and scalability are presented.

Chapter 5 presents a software technology for on-line performance analysis and visualization of parallel and distributed, heterogeneous systems. A visual object framework is described, followed by an example of its successful use for PAV of a distributed multimedia real-time application. A PAV architecture for the visual objects, a markup language based on it called VOML, and the development environment are presented next. The chapter concludes with an example of a VOML specification and corresponding performance visualizations.

Chapter 6 presents an integrated approach to the design and engineering of complex real-time systems, their instrumentation, on-line PAV, and dynamic system reconfiguration. Figure 1.1 illustrates this approach; it also serves as a reference by showing the relations between different parts of the work presented in this dissertation. A real-world target distributed real-time system is described, followed by a description of its design and engineering performed using the approach described in Chapter 3. Technical details of the integration are explained next, and the operation of the new, integrated technology is shown on an example execution scenario.

Chapter 7 concludes the dissertation with a summary of the major contributions and future directions in areas that include complex real-time system problem-solving related optimizations, extensions of the distributed IS kernel, advanced on-line PAV tools, and different integrations of the systems engineering and on-line PAV frameworks.

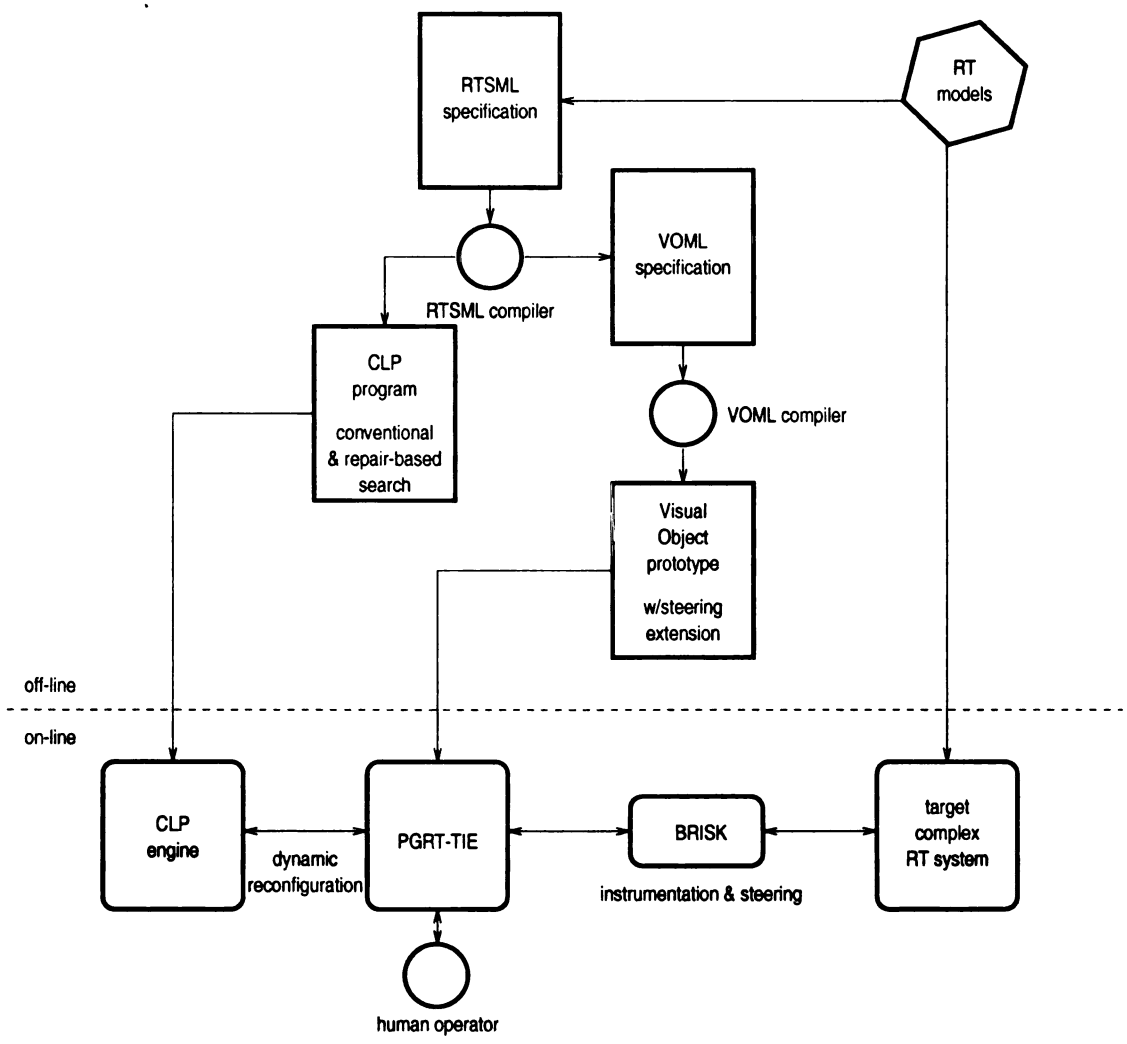

Figure 1.1: Overview of the integrated approach

Chapter 2

Background and Related Work

This chapter discusses background and previous work related to the dissertation research presented in the following chapters. The first section covers several approaches to the design of distributed real-time systems. The second section provides background in Constraint Logic Programming (CLP) and presents several works that use it as a tool in the area of real-time systems. The third section overviews other works in the area of real-time systems that deal with explicit timing constraints (as opposed to implicit ones, such as those contained in the constraint formulae from real-time scheduling theory models). The fourth section describes several distributed instrumentation systems and their relation with BRISK. Finally, the fifth section describes related work in the area of system performance visualization, including a few more comprehensive, integrated approaches.

2.1 Distributed Real-Time System Design

As was stated in Section 1.2, the state of the art in the area of real-time scheduling still does not provide integrated and comprehensive solutions for realistic scenarios. Among many fragmented design and engineering approaches for parallel and distributed real-time systems, some concentrate on integration and decomposition. More comprehensive approaches usually propose a search algorithm for solving system parameters, such as branch-and-bound, simulated annealing, or even greedy. For specific classes of distributed real-time systems, an approach may include run-time support as well. The descriptions of some related approaches are presented below and ordered approximately from simpler to more complex ones.

In [102], Spuri and Stankovic address the problem of integration of task precedence constraints with resource sharing in real-time scheduling. Their motivation is to give more freedom to the scheduler so that more dynamic real-time systems can be supported. They derive analytical task schedulability formulae that can be applied in more real-time system situations than previously developed.

An engineering technique for decomposing end-to-end delays in distributed real-time systems is proposed by Saksena and Hong [97]. In effect, a global distributed scheduling problem is transformed into a set of single-processor scheduling problems with local deadlines. The problem solving approach consists of an approximate technique to quickly generate an initial solution, and an iterative method to fine-tune the initial solution.

In an end-to-end approach to the design of real-time systems by Gerber et al. [39], real-time applications are structured as a set of process components, connected by asynchronous channels in which the end-points are the system's external inputs and outputs. End-to-end propagation delay, temporal input-sampling correlation, and allowable separation times between updated output values, are postulated as end-to-end constraints. The problem solving approach is a multi-stage procedure that involves an augmentation of the problem, an optimization algorithm that generates a set of intermediate rate constraints, and a domain-specific constraint solver.

An approach to scheduling and allocation in multiprocessor real-time systems is described in [23]. Cheng develops a hybrid timings model that combines absolute and relative timing constraints on tasks. Based on this model, the simulated annealing technique is applied as the overall search algorithm to find feasible schedules over multiple processors. A task replication technique (for the purpose of improving the scheduling) is developed and embedded into the simulated annealing algorithm.

A framework that provides a systematic approach to designing distributed, heterogeneous real-time systems that utilize resources in an efficient, pipelined and predictable manner, is proposed by Chatterjee and Strosnider in [22]. It defines abstractions for representing real-time applications and capturing the fundamental properties of distributed pipelining; a flow control mechanism; a decomposition of the multi-resource scheduling problem into a set of single resource scheduling problems with well-defined interactions; an analysis methodology to support heterogeneous scheduling policies among system resources; and a delineation of how manipulating

system configuration parameters affect various application timing metrics. It does not specify any particular mapping or optimization technique.

In [84], Mutka and Li describe a tool that finds feasible processor allocations for sets of rate-monotonically scheduled (RMS) tasks over a set of heterogeneous processors. It uses three different RMS tests, considers possible task blocking due to priority inversion, allows task co-allocation, and can perform task transformations if needed. Communication among the tasks is not considered. The problem solving approach is based on a branch-and-bound algorithm.

An optimal solution to the problem of allocating communicating periodic tasks to heterogeneous processing nodes in a distributed real-time system by Peng et al. is presented in [87]. The maximum normalized task response time is minimized subject to the precedence constraints resulting from intercommunication among the tasks to be allocated. The task system is modeled with a task graph in which computation and communication modules, communication delays, and intertask precedence constraints are described. These tasks are assigned to processing nodes by using a branch-and-bound search algorithm.

In [66], Kang et al. describe a distributed real-time system model with statistical, end-to-end constraints. It exploits both discrete-time Markovian analysis and real-time scheduling theory, and uses several approximations to avoid modeling the entire system. A system is modeled as a set of chains, where each chain is a distributed pipeline of tasks, and a task can represent any activity requiring non-zero load from a processor or network resource. Every chain has two end-to-end constraints: delay and minimum allowable success rate for outputs that meet their delay constraints. The

search algorithm uses two heuristics, which help in significantly reducing the number of potential feasible solutions but, at the same time, can miss some and report a failure.

PERTS [101] is a commercial prototyping environment based on the rate monotonic analysis (RMA), initially developed at the University of Illinois at Urbana-Champaign. It integrates multiple analysis tools allow a real-time system designer to test and evaluate the system against various design scenarios. Several real-time scheduling algorithms and protocols are supported, and end-to-end analysis for single- and multiple-node architectures is provided. It interfaces to Real-Time CORBA [27], ObjecTime [74] and a few more real-time software technologies.

An approach due to Welch et al. for engineering time-constrained systems which must operate in dynamic environments (with potentially unknown worst-case scenarios and large, unpredictable variances in system parameters) is presented in [120]. A specification language was developed that enables the description of environment-dependent features. An abstract model constructed from the specifications is augmented dynamically with the state of environment-dependent features. It is also used to define techniques for quality-of-service monitoring and diagnosis, and allocation analysis. A prototype resource-management middleware was developed to experimentally evaluate the approach.

In the light of an overall approach that would be able to handle all the issues brought out by Stankovic and listed in Section 1.2, the approach presented in this dissertation attempts to address the need for a comprehensive computer-aided design and engineering framework for complex real-time systems. However, it fundamentally

differs from one that implicitly follows from the above-mentioned motivation in [102] by the same author. The latter is based on the need for a single real-time scheduling approach that would address as many realistic scenarios at once as possible. The approach presented in this dissertation aims to integrate and compose extant real-time scheduling approaches, which address only specific scenarios, into a complex, more realistic one. Intuitively, a single complex scheduling approach might not scale well in complex real-time systems, which are parallel/distributed systems that should scale with respect to the system real-time-liness. (It is easy to see that for a single resource, such as CPU, the addition of support for more realistic scenarios in real-time systems, such as fixed priorities to activities and/or resource sharing among activities, strengthens schedulability conditions, resulting in a lower resource utilization.) Such an approach might overly underutilize the resources, leaning toward extensive resource sharing and tight integration. In other words, an inherent property of loosely-coupled systems, such as distributed computer systems, is that the operating time scale is larger than that of tightly-coupled ones, such as symmetric multiprocessors. This is rather relevant to realistic real-time systems, in which the operating time scale affects the real-time-liness. (As a side note, anticipating as many realistic scenarios at once as possible is similar to analyzing worst cases in the hard real-time scheduling theory. For a reference, Jensen in [63] argues that hard real-time scheduling does not scale.) Another important difference between the two approaches is that Stankovic' one is more scheduling theoretic, while the one presented in this dissertation is a systems engineering approach. The former attempts to devise real-time scheduling models that would support more realistic scenarios without strengthening feasibility

conditions. The latter attempts to build realistic real-time systems by placing an emphasis on the use and integration of appropriate extant real-time models.

2.2 Constraint Logic Programming and Real-Time Systems

Constraint Logic Programming tools have matured over the last decade and solvers exist for a wide range of problem domains. For example, solvers for combinatorial problems over finite domains and sets, systems of equations and inequalities involving rational and real numbers, and solvers for systems of non-linear equations over real numbers can be integrated within a CLP tool by the means of a Logic Programming language, usually extended Prolog. CLP has been used in a variety of applications, such as scheduling, resource allocation, timetabling, financial planning, frequency assignment for cellular phones, etc. [116] Research in various areas of engineering have used CLP for solving practical industrial problems. Constraint Logic Programming has been used in some areas of computer-aided engineering (e.g., in mechanical engineering [69, 107]; for VLSI design of electrical circuits [13]; in manufacturing [124]; in computer system performance analysis [75]) in the recent past. In the context of Artificial Intelligence, a more general framework of Constraint Programming has been used in reactive systems [35] and electro-mechanical machines [114].

Before more related work, which uses CLP in the area of real-time systems, is presented in Section 2.2.2, CLP background is given in Section 2.2.1. Section 2.2.3 describes ECL^{i}PS^e, the CLP tool used in the work presented in this dissertation.

2.2.1 Constraint Logic Programming Background

The insight which led to the design of the CLP framework is the observation that the algorithm of unification used in Logic Programming (LP) is a constraint solving algorithm and as such it could be combined with, or replaced by, various other constraint solving algorithms. In other words, LP offers the means to create a single and powerful framework for various cooperating constraint solving algorithms [116].

More formally, CLP is a many-sorted generalization of LP, in which different sorts are associated with different interpretation domains, and corresponding formulae are manipulated using predefined constraint solvers. Special classes of formulae, called constraints, are not handled using traditional resolution, but are interpreted under a predefined specific interpretation and handled by external constraint solvers. Basic definitions of the formal CLP framework presented in [71, 56] are given below.

A CLP language is built upon a set Σ of function and constraint predicate symbols, called *signature*. Primitive constraint predicates (including the equality symbol $=$) belong to Σ and are interpreted with respect to a predefined interpretation structure, called Σ-*structure*, while user-defined constraint predicates are subject to the user definitions.

A term is an object created using function symbols from Σ and a collection of variables. It can be either a simple variable or an application $f(t_1, \ldots, t_n)$ of an n-ary function symbol $f \in \Sigma$ to n terms $t_1, \ldots, t_n$ $(n \geq 0)$. An atom is an application $p(t_1, \ldots, t_n)$ of a constraint predicate symbol p to n terms $t_1, \ldots, t_n$. If p is a primitive constraint predicate, then the atom is called a *primitive constraint.* Every *constraint* is built from primitive constraints. A program is composed of a collection of clauses, where each clause has the form:

$$head : -c \quad | \quad b_1, \ldots, b_k$$

where *head* and $b_i (i = 1, \ldots, k)$ are user-defined atoms, while c is an arbitrary conjunction of constraints. The symbol | is used in the body of the clause to separate the constraint part from the goals and can be read as "and," in the same way as a comma in the body of the clause.

A Σ-structure D consists of a set D and an interpretation function I_D. A constraint c is solvable if $D \models \exists(c)$, where the notation $\exists(\phi)$ denotes the existential closure of the formula ϕ (i.e., each variable in ϕ is within the scope of an existential quantifier). A Σ-theory is a collection of closed Σ-formulae (i.e., formulae built over Σ). A solution θ for c is a mapping from the variables in c to D, such that $D \models c\theta$.

The execution of a constraint program requires the use of constraint solvers capable of deciding the solvability of each possible constraint formula. Resolution is extended in order to embed calls to constraint solvers. If $? - c_1 \quad | \quad g_1, \ldots, g_n$ is a

goal, and $p : -c_2 \quad | \quad b_1, \ldots, b_k$ is a clause in the program, then the resolvent of the goal wrt. the given clause is

$$? - (c_1, c_2, g_1 = p) \quad | \quad b_1, \ldots, b_k, g_2, \ldots, g_n$$

as long as $D \models (c_1 \wedge c_2 \wedge (g_1 = p))$. The symbol $=$ is an abbreviation for the conjunction of equations between corresponding arguments of g_1 and p, if g_1 and p have the same predicate symbol. The constraint solver is used to test the validity of the condition on the constraints.

The idea behind the introduction of the CLP framework is that a logic-based programming language, its declarative and operational semantics and the relationships between these semantics can be parameterized by a choice of the domain of computation and constraints. The resulting scheme defines the class of languages CLP($\mathcal{X}$) obtained by instantiating the parameter $\mathcal{X}$ [116].

The parameter $\mathcal{X}$ stands for a 4-tuple $(\Sigma, D, \mathcal{L}, \mathcal{T})$, where Σ is a signature, D is a Σ-structure, $\mathcal{L}$ is a collection of Σ-formulae and $\mathcal{T}$ is a first-order Σ-theory. The Σ determines the predefined predicate and function symbols with their arities, D is the structure over which computation is to be performed, $\mathcal{L}$ is the class of constraints which can be expressed, and $\mathcal{T}$ is an axiomatization of some properties of D. The pair $(D, \mathcal{L})$ is called a *constraint domain.* One such domain, *FD*, will be described in Section 2.2.3.

2.2.2 Use of CLP as Support for Real-Time Systems

CLP and its more general version, Constraint Programming, have successfully been used as a tool for solving specific problems in the area of real-time systems as well as other areas with certain real-time aspects, ranging from temporal reasoning and scheduling to formal methods, to resource allocation. This fact supports the motivation for using CLP as an elegant way of expressing the disparate requirements of real-time systems, in Section 1.2.1.

A family of logics and associated programming languages for representing and reasoning about time is introduced in [37]. The family is conceptually simple while allowing for different models of time. Formulae can be labeled with temporal information using annotations. Unlike temporal logic [88], both qualitative and quantitative (metric) temporal reasoning about definite and indefinite information with time points and time intervals in different models of time are supported. The introduced temporal annotated logic can be made an instance of annotated constraint logic, and there is a systematic was to make a clausal fragment executable as a CLP program.

A new class of application domains for Constraint Programming is introduced in [98], due to the emergence of special real-time systems, enjoying increasing popularity in the areas of automotive electronics and aerospace industry. Real-time systems of this kind are time-triggered in the sense that their overall behavior is globally controlled by a recurring clock tick. An off-line scheduling approach maps infinite, periodic processing onto a single time window of a fixed length. The authors also

describe which techniques from traditional scheduling and real-time computing led to success and which failed, when confronted with a large-scale application of this type.

In [33], the author first explains how the bottom-up evaluation method of Revesz for computing least-fixed points of CLP programs can be adapted from the domain of integers to the domain of reals. The procedure is applied on a state reachability problem in timed automata [2], also including certain extensions of timed automata. It has been successfully experimented for proving automatically the correctness of a sophisticated reactive program that controls dataflow rates on ATM networks.

A CLP-based framework is developed in [46] for specification and verification of real-time systems that is based on the notion of timed automata. A user models the ordering of real-time events as the grammar of a language accepted by a timed automaton, and real-time constraints on these events are then captured as denotations of the grammar productions specified by the user (i.e., the valuation function of the associated denotational semantic maps into the time domain). The resulting specification is a CLP program that is executable. Many interesting properties of the real-time system can be verified by posing appropriate queries to this CLP program. The approach is also constructuve in the sense that conditions can be computed under which a property will hold for a given real-time system.

In [45], the author applies CLP on a problem of global optimization of DSP application mapping onto parallel architectures. The problem is characterized by numerous resources (number of processors, bandwidth, memory size), real-time constraints (latency, sampling) and many non-linear constraints. The author also notices the capacity of CLP to compose several concurrent system models. Certain aspects

of the problem are presented in [3]. It is shown how it is possible to handle and solve three system models at once, under architectural and real-time constraints: a data-partitioning model equivalent to that supported by HP Fortran; a fine-grained (at macro-instruction level) scheduling; and a capacitive, distributed shared memory model.

2.2.3 Overview of ECL^{i}PS^e

The ECL^{i}PS^e platform integrates a number of constraint solvers, including ones for solving sets of constraints over finite domains CLP(FD), real and rational numbers (CLP(R,Q) [52]), and intervals over real numbers (RIA) [53]. A special library called REPAIR [96] allows the user to start with a tentative solution, which can be modified, or repaired, if it turns out inconsistent with the constraint set.

The finite domain support consists of three libraries: for symbolic finite domains; for handling integer variables and numerical constraints on these variables; and with built-in complex constraints. The second library is the major one, propagating equations and inequalities between linear expressions. A linear numeric expression is one that can be written in the form $Term_1 + Term_2 + \ldots + Term_n$, where each term can, in turn, be written as $Number$ or $Number \cdot Variable$. According to the definitions in Section 2.2.1, the set D in this case is the set Z of integers, and finite domain constraints are existential positive formulae built up from the linear expressions and the five predicates $=$, $>$, $\geq$, $<$ and $\leq$, interpreted in Z, and infinitely many membership predicates $\in [a, b]$, one for each finite interval $[a, b]$ of Z. The

primitive constraint $x \in [a, b]$ is interpreted by the empty set when $b < a$; similarly, $x = a$ and $x \in [a, a]$ are identified. Solving finite domain constraints is both NP-complete and very important for practice, which has favored the use of a practically efficient technique called *constraint propagation.* It consists of a set of transformation rules such that each primitive ordering or equality constraint c between the variables $x_1, \ldots, x_n$ in c, whose domains are defined by membership constraints $x_i \in [a_i, b_i]$, induces new restrictions on these domains, resulting in new membership constraints. A *reduced domain* $RD(x_j, c)$ of the variable x_j for c is the smallest interval $[a'_j, b'_j]$ containing all $m \in Z$ such that the constraint obtained by substituting x_j by m in $c \wedge \left(\bigwedge_{i=1}^{i=n} x_i \in [a_i, b_i]\right)$ is satisfiable in FD.

In practice, a typical CLP(FD) program consists of three sections. First the domains of program variables are declared (the notation is slightly different than above; comma is used to separate intervals, and intervals are specified using ..). For example, the variable declaration `Task_0_TimeAt1MIPS :: [0, 1000 .. 2000]` could be interpreted as task 0 being either inactive (does not use any CPU time) or active such that it may require between 1,000 and 2,000 milliseconds of CPU time on a unit-capacity CPU. (Some program variables, a.k.a. meta-terms, may be assigned attributes other than finite domains, and multiple different, cooperating constraint solvers may be involved.) Second, constraints are stated that are used to build a constraint network at run time. As opposed to the generate-then-test approach to searching of logic programming, CLP uses a much more efficient constrain-then-generate approach. The last program section defines the order the program variables will be assigned values, which are consistent with the constraints, and the order

those values will be tried; this is called *labeling*. Sometimes, stating the constraints is enough to reduce the domains of program variables to single values and obtain solutions, if any. In other situations, the CLP system cannot decide the problem with this information only, i.e., it may return domains of the program variables' values for which there *might be* solutions. As the program variables are assigned values in an user-specified order, their domains are further reduced by the constraint propagation. If the CLP system detects that the constraints cannot be satisfied with current value assignments, backtracking is performed and most recent assignments are undone. Other CLP solvers may handle their constraints more or less differently than CLP(FD).

2.3 Explicit Timing Constraint Checking

In the area of specification and verification of real-time systems, explicit timing constraints have been a research focus during the last years. They have been studied in two main contexts. The first one is formal methods, where theories based on the timing constraints have been developed, independently of the real-time scheduling theory, for expressing and statically checking temporal properties of real-time systems. The other context is real-time languages and systems, where the timing constraints are checked dynamically at run-time.

Major work in the context of formal methods has been done by A. Mok and F. Jahanian et al. Their work started with Real-Time Logic (RTL) [59], a formal language for the specification of real-time systems. RTL formulae are constructed

using addition and subtraction of event occurrence functions (which map to the time domain) with integers, (in)equality predicates, universal and existential quantifiers, and logic connectives. The semantics of RTL is based on the occurrence of events (using the absolute timing, not only ordering) that are based on the execution of a real-time system, such as an event coming from the environment, the start and end of code blocks or the assignment of values to a state variable. Checking whether a set of timing constraints in RTL is satisfied is generally undecidable, although certain classes of RTL formulae are easier to check [57, 106]. Algorithms were developed for checking safety properties [60] and partial event-traces [57].

The research was later directed toward the other context. A distributed on-line monitoring and checking tool was described in [24] and [58] that allows for specifying timing properties in a subset of RTL and checking for their violation. The work describes how to store events coming from the system, how to define timing constraints based on these events, and how to evaluate the constraints in a distributed environment. A graph-based algorithm was developed that evaluates these RTL formulae whenever a new event arrives or when a timeout expires (at earliest possible time).

In the context of dynamic timing constraint checking, a real-time system is equipped with means for detecting and/or preventing timing constraint violations. When a timing constraint violation is detected, the system tries to adapt by, for example, activating stand-by resources, rescheduling of the remaining resources, or executing alternative algorithms for solving the problem under different conditions.

A methodology for specifying and checking timing constraints in a distributed object-oriented environment is presented in [40]. The focus there is on how object

orientation can be utilized to simplify the specification and the checking of timing constraints and how this can be integrated in an existing programming language like C++. The methodology integrates a precompiler that generates instrumentation code and a constraint checker based on RTL-like timing constraints specified for each class, and a distributed instrumentation system. The functional and timing specifications are semantically separated. Notifications by constraint checkers in the form of events are used as the feedback from the target system.

Another approach to dynamic timing constraint checking in distributed object-oriented systems is presented in [93]. Like the previous approach, it separates real-time requirements from individual commands in a program. The real-time requirements are timing constraints described by declarative synchronization code between the interfaces of objects, expressing common, message-based temporal coordination patterns. Objects in the system are based on the Actor model [1], and a high-level programming language construct called *RTsynchronizer* is defined that specifies a collection of temporal constraints between actors. The run-time system is able to dynamically enforce timing constraints, based on the principles of *safe progress* and *unsafe wait*, by e.g. delaying messages. A group of actors may be constrained by overlapping RTsynchronizers, which may be dynamically added or removed.

In [85], the work has been extended and the actors and synchronizers have been assigned formal operational semantics. The actor semantic interprets actor primitives as two-phase transitions between configurations, which are modeled by ordered pairs (α, μ), where α represents actor states and μ is the set of pending messages. As for the synchronizers, a constraint configuration is similarly defined as an ordered pair

(χ, ξ), where χ is a multi-set of demands for message invocations corresponding to the constraint ξ; transitions are determined by constraint *firings*, whenever a message invocation matches a pattern in a constraint. A collection of synchronizers is termed an interaction constraint system configuration. Finally, the two operational semantics are composed into one. The overall model is yet to be implemented, and the authors have identified three main tasks: a (so-called constraint-directed) scheduling strategy that fits the synchronizer approach; constraint propagation by the compiler and run-time system; and the distribution of synchronizer entities.

A number of other formal methods for real-time systems treat explicit timing constraints using various time models. For static checking, the CLP approach described in this dissertation could support some of the formal methods on the grounds of common underlying frameworks such as logic and Presburger (linear) arithmetic. An interesting issue is one of the integration of these models with models from real-time scheduling theory within the compiler-based framework.

While the behavior of a real-time system based on static analysis becomes undefined as soon as some of its design assumptions are violated, it is difficult to provide any guarantees for a real-time system equipped only with dynamic timing constraint checking. The approach to dynamic system reconfiguration described in this dissertation attempts to combine static analysis and constraint checking with on-line system performance analysis. Similarly to, for example, common analysis of missed task deadlines, the on-line performance analysis may include dynamic timing constraint checking as described above. Namely, parts of real-time models that allow for static analysis are evaluated in advance by the CLP tool to derive unknown system param-

eters. Run-time system performance degradations and constraint violations are then fed back to the CLP tool, which tries to solve for new, better values of the system parameters by taking into consideration all the static requirements that result in certain guarantees. While a run-time system for dynamic timing constraint checking can prevent some constraint violations and quickly detect unpreventable ones, it itself lacks complete information to steer the target system toward a provably better configuration.

2.4 Distributed Instrumentation Systems

Many distributed ISes have been developed over the past decade (e.g., [30, 80, 47]), usually as components of larger software toolkits for analysis of parallel/distributed systems. Only a few have been ported to multiple platforms and made available to broader usage. Several such systems are briefly discussed here. The usage of BRISK in the context of the objectives and features of these distributed ISes is based on certain inferences only, and their developers may have different recommendations.

The Automated Instrumentation and Monitoring System (AIMS) [123] contains **xinstrument**, a source-code instrumentor for Fortran77 and C, and **monitor**, a library of time-stamping and trace-collection routines that generates trace files. Tools for off-line processing of trace files include a utility for removing monitoring overhead and maintaining consistency of causally-related events, and a trace file animation and analysis toolkit. If **xinstrument** were modified to support BRISK, BRISK could replace the rest of the IS.

The Pablo [91] software instrumentation system contains an instrumenting parser with a GUI, and a performance data capture library for generating trace files in Self-Describing Data Format (SDDF) [4]. The instrumentation software supports tracing, interval timing, and counting. The instrumentation library monitors the instrumentation overhead and volume of data, and can automatically adjust the intrusion on the target application by changing the monitoring method. The basic implementation of BRISK could be extended to support SDDF and automatic intrusion adjustment.

Paradyn [79] takes an approach called dynamic instrumentation for dynamically controlling the performance data to be collected. The design of the tool is based on two data abstractions: metric-focus and time-histogram. The dynamic instrumentation environment includes a compiler and code generation, structural analysis of the binary, and an instrumentation manager that allows code to be inserted and removed from the running program. It also incorporates a strategy for describing performance information to users of high-level parallel languages. This is an example of a comprehensive, specialized distributed IS, although ported to multiple platforms, which probably would not benefit fundamentally from BRISK as its kernel.

Falcon [43], a monitoring and steering system, uses a low-level sensor specification language and mechanisms for on-line capture and analysis of application-specific information about large-scale parallel programs. It includes a semi-portable binary I/O facility and a tool for instilling a partial order on an unordered event stream. The latest BRISK implementation provides generic capabilities that can be used to emulate those of the core Falcon system (i.e., not including auxiliary tools).

The JEWEL distributed measurement system [70] uses low-cost software sensors in the form of cpp macros, assumes synchronized hardware clocks, and has a data collection and reduction infrastructure based on the External Data Representation (XDR) protocol [54]. It is integrated with a configurable graphical presentation facility providing a set of single-metric views, and an interactive experiment control system. JEWEL components are based on a custom configuration language and data and control protocols. A distributed IS with JEWEL's features and slightly different architecture could be built on top of an extended BRISK.

Vista [117], a C++ framework for development of domain-specific distributed ISes, is based on a generic distributed IS model that defines three components of a distributed IS: (1) local instrumentation server (LIS), (2) IS manager (ISM), and (3) transfer protocol (TP). In the interests of flexibility, this simple model has been adopted for the BRISK design. For performance reasons, the BRISK LIS implementation is based on JEWEL's internal and generic external sensors.

In Cristian's distributed (centrally-controlled, master-slave) clock synchronization algorithm [29], a master node polls slave nodes in so-called *rounds*. In each round, the master queries each slave for its current time, and waits for an answer; when the answer is returned, the master computes the time difference between the pair. This querying is repeated a number of times for each slave to average the results. At the end of each round, the master sends the time differences to the slaves to adjust their clocks. In effect, the master maintains the slaves' clocks synchronized relative to each other (a.k.a. internal synchronization) by maintaining their values within a maximum deviation from its clock value (a.k.a. external synchronization). Furthermore, the

algorithm discards measurements whose round-trip time is longer than one calculated based on the assumed clock drift, shortest round-trip, and achievable clock error. It also adjusts the number of queries in a round based on the observed probability of a successful query. A variant of this algorithm has been used in a distributed IS called DTM, described in [47].

Kimelman and Zernik in [67] present a technique for on-the-fly ordering and matching of causally-related event data records that are being produced by a number of distinct processors, engaged in multiple one-to-many and many-to-one communications. The technique is optimal in terms of the amount of space required, and in terms of the amount of additional delay incurred prior to delivery of an event data record to its ultimate destination. The time-stamps of out-of-order causally-related event data records are used, instead of a distributed clock synchronization algorithm, to estimate clock drifts of the processors, and these estimates are used for correction of the time-stamps of the preprocessed event data records.

The primary source of software-based instrumentation intrusion is execution of additional instructions, while side-effects include changes in memory reference patterns, event reordering and even CPU stalls. Malony presents in [76] two time-based intrusion and perturbation analyses of software-based tracing on a multiprocessor. (In [76], the term perturbation is used for any—not only critical—change in event timing due to instrumentation.) Based on two derived performance perturbation models, trace event times are adjusted to (1) remove delays due to the measured costs of instrumentation, and (2) reorder the event sequence based on knowledge of event dependencies, maintaining causality. Experimental results show that it is possible to

characterize perturbations through simple models and recover the actual execution timing with up to 20% error.

It is clear that extant distributed ISes have different goals in different domains, but share many similar needs and characteristics. The related work discussed in this section supports the motivation from Section 1.2.2 for the development of BRISK as a distributed IS kernel, with an emphasis on features that are useful for instrumenting complex real-time systems.

2.5 Performance Analysis and Visualization

In this section, several related PAV tools, systems and environments are described. The PAV environments are progressing with features to incorporate new analysis and display modules. Visualization environments are not only becoming extensible, but retargetable to different analysis scenarios.

ParaGraph [49] is a graphical display tool developed by Michael T. Heath for visualizing the behavior and performance of parallel programs that use PICL (Portable Instrumented Communication Library) [122] or MPI (Message-Passing Interface) [31]. The visual animation of a parallel program is based on execution trace information gathered during an actual run of the program on a message-passing parallel computer system. The resulting trace data are replayed pictorially to provide a dynamic depiction of the behavior of the parallel program, as well as graphical summaries of its overall performance. The same performance data can be viewed from many different visual perspectives to gain insights that might be missed by any single view. The

necessary execution trace data are produced by a tool called MPICL, developed by Pat Worley of Oak Ridge National Laboratory, which uses the profiling interface of MPI to provide timestamped records of MPI events.

Pablo took the PAV environment research one step further by incorporating support for performance analysis environment prototyping [92]. The Analysis GUI component of the Pablo Performance Analysis Environment consists of a performance visualization system that provides a portable, scalable, and extensible tool for the analysis and display of data written in the Pablo Self-Defining Data Format (SDDF) [4]. The user interacts graphically with the Analysis GUI to build a data-flow graph whose nodes organize, transform, analyze, and display data read from Pablo SDDF files. Typically, the input files are generated with a trace library component.

Viz continues in this direction by focusing on the visualization technology required for application-specific performance visualizations [50]. It was created out of a need to support rapid visualization prototyping in an environment that could be extended by abstractions in the application problem domain. Viz provides this in a programming environment built on a high-level, interactive language (Scheme) that embeds a 3D graphics library (Open Inventor), and that utilizes a data reactive model of visualization operation to capture mechanisms that have been found to be important in visualization design (e.g., constraints, controlled data flow, dynamic analysis, animation). The strength of Viz is in its ability to create non-trivial visualizations rapidly and to construct libraries of 3D graphics functionality easily. Although our original focus was on parallel program and performance data visualization, Viz applies beyond these areas.

Avatar [90] is a virtual reality framework, built on the Pablo performance analysis kit that supports multiple metaphors to display dynamic data. By separating the metaphor display software from the data processing and interaction components, Avatar's software architecture has allowed for quickly creating new display metaphors. Three different display metaphors for performance data have been developed to date: time tunnels (time lines and event driven graphs of task interactions), scattercubes (3D generalizations of 2D scatterplots that support analysis of high-dimensional time-varying data), and geographic displays (texture-mapped spheres with source-destination arcs and stacked bars). Avatar has been used to study two types of high-performance input/output of the Portable Parallel File System (PPFS): parallel scientific codes and WWW servers.

Rivet [18] is an information visualization environment that provides a cohesive platform for the analysis and visualization of modern computer systems. It uses a component-based architecture in which complex visualizations can be composed from simple data objects, visual objects and data transformations. Additionally, it provides powerful coordination mechanisms, which can be used to add extensive interactivity to the resulting visualizations. Rivet has been successfully applied in focused studies of a wide range of computer systems: parallel applications, superscalar processors, memory systems, and wireless networks.

Lucent Technologies' Visual Insights [115] offers a set of interactive and linked data visualization components for the Microsoft ActiveX developer market that help software developers create more flexible, animated ways to display trends in vast stores of information.

In Table 2.1, PG^{RT} visual objects that are described in Chapter 5 of this dissertation, are compared with the above-described related PAV tools and systems. While some of the latter have gone further in specific directions, such as the graphical metaphor, the design decisions taken in the approach presented in this dissertation were primarily based on the requirements stated in Section 1.2.3. (There is also a concern that insisting on use of the state-of-the-art, academic software technologies in PAV tools and systems, such as, for example, lazy functional languages, could limit their accessibility by non-experts who use legacy software tools. Instead, using technologies that are gaining acceptance, such as SGML (e.g., in the Chemical Markup Language [83]) and use of structure, components and scripting (e.g., in VRML [109]), increases the chance of the PG^{RT} visual objects contributing to the PAV community.)

Table 2.1: Performance visualization tools and systems

Tool/System	On/off line	Graphical metaphor	Underlying graphical technology	View classes	Reusable
ParaGraph	off-line	data-flow	X library	generic	no
Pablo widgets	off-line	data-flow	X library	generic	yes
Avatar	on-line	data-flow	VRML	three metaphors	n/a
Viz	on-line	data-reactive	Open Inventor	domain-specific	yes
Rivet	off-line	data-flow	OpenGL	domain-specific	yes
Visual Insights	mostly off-line	n/a	n/a	generic	yes
PG^{RT} VO	on-line	data-flow	low-level VO implementations	domain-specific	yes

Among the few more comprehensive PAV approaches that try to link on-line PAV with system modeling, design, engineering, and steering, PERTS [101] and Falcon [43] come close to what is the self-contained result of this dissertation, presented in Chap-

ter 6. PERTS facilitates the design and engineering of real-time systems based on the rate-monotonic analysis, and interfaces Wind River's WindView visualization technology [121]. Falcon is not specifically aimed at real-time systems, but integrates an application-specific on-line monitoring system, an interactive steering mechanism, and a graphical display system, in an effort to affect the performance and/or execution behavior of large-scale parallel programs. A good source of related work is An Annotated Bibliography of Interactive Program Steering [44], which places emphasis on monitoring, information presentation and steering.

Chapter 3

A Compiler-Based Approach to Design and Engineering of Complex Real-Time Systems

This chapter describes the unifying, compiler-based approach to the design and engineering of complex real-time systems. The Real-Time System Markup Language (RTSML) is described in Section 3.1. An example of a complex real-time system accompanies the description, in order to make it easier to explain how the compiler generates CLP code in Section 3.2. Results of experiments with the example system are presented in Section 3.3. Section 3.4 briefly comments on the scalability issues in the approach. Before a summary, Section 3.5 discusses correctness issues in the approach and RTSML-to-CLP compilation.

3.1 Real-Time System Specification in RTSML

In this section, the RTSML language is described using excerpts of an example complex real-time system that is motivated by a shipboard control system described in [119]. Figure 3.1 shows the hardware configuration and a detail from the software configuration of the system. The squares represent processors, circles represent channels, and small squares labeled R represent routers; the lines connecting the processors and channels show possible routes among the processors.

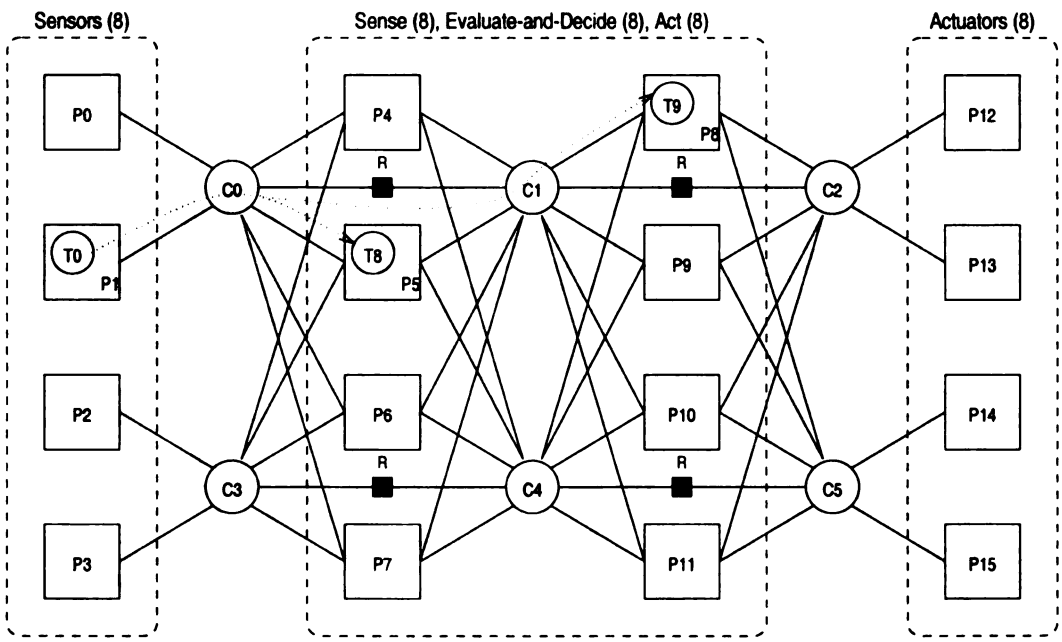

Figure 3.1: An example complex RT system

There are 40 periodic tasks, divided into five groups of 8 tasks each. Sensor tasks make the first group, to be allocated over the four leftmost processors. Sense, evaluate-and-decide, and act tasks are all to be allocated over the eight processors in the middle. Actuator tasks are to be allocated over the four rightmost processors. The communication among the tasks goes from left to right: each sensor sends a

```
<!DOCTYPE RTSML PUBLIC "-//MSU-PGRT//DTD RTSML 2.0//EN">
<rtsml>
...
  <system id="s0" type="common">
    <processor-group id="pg0" type="common">
      <processor id="p0" type="rms (mips (initial 10)
        (domain (10)))"></processor>
    </processor-group>
...
    <processor-group id="pg1" type="common">
...
      <processor id="p8" type="rms (mips (initial 10)
        (domain ((range 8 10))))"></processor>
...
    </processor-group>
    <task-group id="tg0" type="common">
      <task id="t0" type="common (period (initial 50)
        (domain ((range 50 80)))) (deadline (initial 50)
        (domain ((range 40 50)))) (time@1mips (initial 180)
        (domain ((range 100 300))))" processors="pg0">
      </task>
...
    </task-group>
    <message-group id="mg0" type="common">
      <message id="m0-8-9" type="common
        (period (initial 500) (domain ((range 500 600))))
        (deadline (initial 15) (domain ((range 10 30))))
        (length (initial 17) (domain ((range 8 32))))"
        source="t0" destinations="t8_t9">
      </message>
...
    </message-group>
    <task-group id="sense" type="common">
...
      <task id="t9" type="common
        (period (initial 1200) (domain ((range 800 1600))))
        (deadline (initial 1000) (domain
        ((range 800 1200)))) (time@1mips (initial 2300)
        (domain ((range 2000 4000))))" processors="pg1">
      </task>
...
    <channel-group id="upper" type="common">
      <channel id="c0" type="rms (mbps (initial 10)
        (domain (10)))"> ...
    </channel-group>
...
    <routing-table id="rt0" type="common">
      <route id="r0-8" type="common" source="p0"
        destination="p8" channels="c0 c1">
...
    </routing-table>
...
```

Figure 3.2: RTSML specification excerpts

Table 3.1: Ranges of system parameters

Task, message group	period [ms]	deadline [ms]	demand, length [kI, kB]
Sensors	25–120	25–75	40–400
Sense	800–2500	800–1800	1500–6000
Eval & Decide	800–2500	800–1800	1500–6000
Act	800–2500	800–1800	1500–6000
Actuators	35–120	30–75	40–400
Sensors to Sense	200–650	10–75	8–80
Sense to E&D	200–650	10–75	8–80
E&D to Act	200–650	10–75	8–80
Act to Actuators	200–650	10–75	8–80

periodic message to two sense tasks (as shown for task T0 sending to tasks T8 and T9); each sense task sends a periodic message to two evaluate-and-decide tasks; each evaluate-and-decide task sends a periodic message to two act tasks; and each act task sends a periodic message to two actuator tasks. Table 3.1 lists the ranges of system parameters. For any specific values of, for example, a task's period and deadline, additional constraints will be enforced according to the real-time model. For example, the rate-monotonic model requires the deadline to be less than or equal to the period. The (idealized) capacity of the processors is 10 MIPS, but the available capacity of any of the eight processors in the middle can possibly drop to 8 MIPS; the channels have the capacity of 10 MB/s. The upper part of the last column shows processor time demands in milliseconds assuming the capacity of 1 MIPS, which is equivalent to demands in kiloinstructions (kI); the lower part shows message lengths in kB.

Parts of the RTSML specification that are relevant to the details shown in Figure 3.1 are highlighted in Figure 3.2. The system consists of processor, channel, task

and message groups, which are specified using corresponding RTSML *elements* (e.g., `processor_group`). The *attributes* of the elements are used to either specify relations among the elements (e.g., the `processors` attribute of the `task` element), or their properties with respect to a real-time model (the `type` attribute). The attributes in dotted boxes of Figure 3.2 show that task `t0` (i.e., `T0` in Figure 3.1) can be allocated to a processor in processor group `pg0` only (lines 5 and 20; corresponding to the leftmost dashed oval rectangle in Figure 3.1), and that task `t9` can be allocated to a processor in processor group `pg1` only (lines 10 and 39; corresponding to the dashed oval rectangle in the middle in Figure 3.1). The attributes in solid rectangles of Figure 3.2 (lines 6, 12, 43 and 48–49) show that processors `p0` and `p8` can communicate via route `r0-8` going across channels `c0` and `c1`; routes are unidirectional and there may exist more than one route between two processors. The attributes in dashed rectangles of Figure 3.2 (lines 17, 25, 29 and 35) show that message `m0-8-9` is multicast by task `t0` to tasks `t8` and `t9`.

As a markup language, RTSML is not intended to capture the semantics of various real-time models. Extending its document type definition (DTD), given in the Appendix A, with model-specific elements and attributes would require the core of the compiler to be modified with the addition of each new real-time model or scheme. Instead, attribute `type` is added to each RTSML element that is used to store the type of the real-time model which the corresponding element instance abides, along with model-specific parameters. In this way, it was possible to design the compiler so that elements of different real-time "types" are handled by different compiler modules, which can be added without modifying the core. In other words, these compiler

modules are responsible for capturing the semantics of real-time models and their integration; it is the responsibility of a module implementor to exploit available constraint solvers efficiently and to provide interface (in terms of generated CLP code) for integration with other modules.

The model-specific parameters are given in the form of S-expressions, a notation close to markup, which are easily parsed. For example, at line 6 of Figure 3.2, the underlined value of the `type` attribute of the first **`processor`** element will be parsed by a compiler module responsible for the Rate-Monotonic Scheduling (**`rms`**) real-time model. This module handles both computation and communication resources (i.e., processors and channels). In the case of processor **`p0`**, the module will determine that it has a constant capacity of 10 MIPS, while in the case of processors in processor group **`pg1`** it will determine that their initial capacities are 10 MIPS, but down to only 8 MIPS may be available at times. (This does not mean a non-deterministic model. This attribute is an input parameter in practice, whose value may vary at times. However, it is beneficial to know its bounds.)

Another compiler module that has been integrated in the current implementation of the RTSML compiler is called **`common`**. It handles elements with a default, or common, real-time model assigned. In the case of tasks and messages, by default they are periodic, have deadlines, and require certain amounts of resources (CPU time required by a task, or communication time required to transfer a message). We have chosen milliseconds and kilobytes as the units for time and message length, respectively. Like the resources, tasks' and messages' parameters may take values

from a finite integer domain, including ranges. The `common` routing model, used in this example, is static.

As new real-time models are supported, the problem of *model compatibility* arises. Namely, it is hard to integrate components or subsystems of a complex real-time system that have been designed separately, based on independent real-time models. The developer of a new model-specific compiler module will define how the new module integrates with the other modules. For example, how to insure that a periodic message arrives before its deadline across a number of rate-monotonically scheduled real-time channels? The `rms` module will cooperate with the `common` module and generate CLP code such that, for example, the message's deadline is divided by the number of channels it traverses, and the floor of the result is used as the message deadline when crossing each channel. Similarly, a stochastic version of the rate-monotonic scheduling model [111] would see a `common` task's fixed CPU requirement as a degenerated distribution. Vice versa, the deterministic `rms` model could work with tasks with stochastic CPU requirements by taking into account the upper bounds of their CPU requirement distributions. A dynamic and/or fault-tolerant routing model, over a set of processors and channels, would probably require more effort to integrate.

All the above examples could be classified into the horizontal system integration. For the benefit of such integration of complex real-time systems, it might be useful to draw from approaches to integration of low-level digital systems. Extending the discussion at the end of Section 2.1, there are numerous examples of successful integration of digital devices implementing different functional subsystems/components by means of their timings specifications. In complex digital systems, there are also

multiple, special-purpose busses and multiple timing protocols. Vice-versa, real-time models might be more integrable if they were devised with the horizontal integration with some—not as many as possible at once—other real-time models among the goals.

Vertical integration of complex real-time systems deals mainly with coupling lower, system-level subsystems/components with higher-level software. In many cases, the real-time properties of the latter are specified using software-engineering formal methods. Future support for integration of formal methods in RTSML is envisaged as their application on special cases of real-time system components that obey the above-mentioned models. One example of such integration, also referred to as method integration, is given in [89]. A real-time design method, called HRT-HOOD [21], that constrains the designer to produce a design that is amenable to scheduling analysis (of periodic and sporadic tasks) is integrated with the Modecharts [81] formal method for real-time systems. In effect, a Modechart design is constrained so that it can be analyzed for schedulability; Modecharts are applied to only the most "important" parts of the design by applying them to the relative HRT-HOOD objects. The following section will show how the CLP approach can handle a schedulability analysis, while Section 2.3 discussed checking of RTL constraints used in Modechart specifications. The future RTSML support might include (1) if it turns out beneficial, syntactic extensions of the core RTSML to support embedding of formal method expressions, and (2) adapting the existing CLP-based (such as in [46]) or other verification/constraint-checking approaches to that of the compiler presented in this dissertation.

3.2 Process of Compilation to CLP

The RTSML compiler is implemented in CLOS [17] and based on an SGML conversion library called STIL [99]. Its core compiles an RTSML specification to a CLP program in four phases:

1. parsing of the RTSML system specification, and creation of a hierarchy of initial objects that contain data collected during the parsing,
2. traversing the object hierarchy and invoking real-time model-specific modules' methods to upgrade the initial objects (by changing their class to its subclass) to module-specific objects with real-time parameter slots and methods for generating chunks of corresponding CLP code (variable declarations and constraints),
3. structure-based (generic and module-specific) cross-linking of the objects in the hierarchy to create a complete internal system representation needed by the CLP-generating methods, and
4. controlling of the CLP code generation.

Each module defines methods, according to an object-oriented compilation protocol established by the compiler core, for creating internal representations of the system structure specified in RTSML, for all the RTSML elements that have the **type** attribute. The internal representation of an RTSML element instance (e.g., **task**), in the context of an RTSML specification, is an object of a class that corresponds to the RTSML element. The object contains

- other objects according to the RTSML specification (e.g., a **task** object contains the list of **processor** objects to which it can be allocated, and vice versa),
- real-time parameters derived from the S-expressions in the RTSML element instance's attributes, and
- dynamically created individual (per-instance) methods that generate parts of the CLP program:

- declaration generator method, for declaring program variables used by the module,
- constraint generator method, for setting constraints upon the program variables, and
- variable generator method, for generating lists of program variables and their values, used as arguments of predicates generated by the compiler core.

These methods may handle CLP code optimization.

The program variables are divided into four groups. *Parameter* variables are used as input arguments to constraint predicates (e.g., `Task_`i`_Period`, denoting the period of task i, is usually known and assigned a value in advance). The values of *solution* variables are to be determined by the CLP program (e.g., `Task_`i`_Processor`, denoting the processor allocated to task i, is to be determined if a goal of the CLP is to find a feasible allocation for this task). *Cost* variables are used to compute the cost of a solution and possibly as arguments to optimization predicates (e.g., the number of tasks allocated the same processor could be represented by such a variable; an optimization predicate could use these variables to find a solution which minimizes the maximum number of tasks allocated to the same processor). Finally, *supplementary* variables are not needed as part of a solution, but must be instantiated in order to guarantee the existence of a solution (otherwise, a possible contradiction might not be inferred if these variables remain represented by non-singular domains at the end of search).

Some implementation details of the two compiler modules, both of which use CLP(FD) as the target language, are described in the remainder of this section. The modules are integrated via shared program variables, which are in turn constrained

by both modules. In other words, the high declarativity of CLP allows that real-time subsystems be specified by laying out (in)equalities over parameters according to their models, and integrated by laying out more (in)equalities. The more these (in)equalities have the nature of simple constraints (as opossed to, e.g., having high computational complexity), the better performance could be expected from the CLP approach.

3.2.1 Module `common`

Task objects created by the `common` module, among other information, contain functions that generate declarations of program variables `Task_`i`_Period`, `Task_`i`_Deadline`, `Task_`i`_TimeAt1MIPS` and `Task_`i`_Processor`, where i is the task index. For each task, the first three variables are parameter variables, and the last one is a solution variable. Messages are handled similarly, but since a message is to be allocated to a number of channels, additional variables need to be declared for storing the route and channels over which a message is transferred to a task, the number of hops along the route, and the per-hop deadline. Since the routing model supports multicasts, this module may, in order to better exploit available resources, generate additional variables and constraints for insuring that (a) if multiple tasks that receive a message are on the same processor, only one route is used for transferring the message to them, and (b) if a message is multicast via multiple but intersecting routes to multiple processors, at most one route can be

used between the intersecting points. Routing implementation details are omitted for brevity.

The `common` module also handles processors and channels with simple, utilization-based task/message feasibility checks. The checks may be used for, e.g., round-robin non-real-time scheduling, but are also strong enough for, e.g., real-time resource management based on the earliest-deadline-first (EDF) criterion. An example of use is given in Chapter 6.

3.2.2 Module `rms`

Processor objects created by the `rms` module, among other information, contain functions that generate declarations of parameter variables `Proc_`k`_MIPS`, where k is the processor index. For the purpose of expressing rate-monotonic (RM) scheduling constraints, additional variables are used, including `Task_`i`_preempts_Task_`j, `Task_`i`_on_Proc_`k and the others that appear in the following examples. This module generates constraints that consider all possible allocations of tasks to processors and message to channels, and set the RM priorities to the tasks and messages. Hard real-time schedulability constraints are generated based on a generalized version of the well-known formula from [113] that guarantees the RM schedulability of a task set (considering the worst-case scenario of a so-called critical instant),

$$(\forall_{1\leq i\leq N} i)(\exists_{0\leq t\leq D_i} t) \sum_{j=1}^{i} C_j \lceil \frac{t}{T_j} \rceil \leq t. \tag{3.1}$$

In the formula, N is the number of tasks indexed by priority and allocated to the same processor, D_i is the deadline of task i, C_j is the time demand of task j on the processor, and T_j is the period of task j. Unlike the formula, the constraints must capture any subset of the N tasks that are actually allocated to the processor in a solution. Additionally, since the tasks' periods are input parameters, the tasks cannot be indexed in advance; their relative priorities must be computed by the CLP program. Finally, the above formula is expressed using the following four types of constraints (see Table 3.2 for the meanings of the constraints):

1. **#<=(Task_i_Period, Task_j_Period, Task_i_preempts_Task_j)**,
   where $i \neq j$, for determining the relative task priority,

2. **ceiling_d(Task_i_on_Proc_k_Checktime, Task_j_Period, Arrivals_of_Task_j_on_Proc_k_by_Task_i_Checktime)**,
   if tasks i and j could be allocated to the same processor k, for determining how many instances of task j could arrive before the deadline of task i,

3. **#=(Task_i_Processor,k,Task_i_on_Proc_k)**,
   for determining whether task i is on processor k (0 = "no", 1 = "yes"); and

4. **Proc_k_MIPS * Task_i_on_Proc_k_Checktime #>= Task_i_on_Proc_k * (Task_i_TimeAt1MIPS + Task_j_1_on_Proc_k * Task_j_1_preempts_Task_i * Arrivals_of_Task_j_1_on_Proc_k_by_Task_i_Checktime * Task_j_1_TimeAt1MIPS + ...)**,
   for expressing the inequality in the formula (the '...' stands for the other summands, in which j_1 is replaced by $j_2, j_3, \ldots$).

The t in the formula corresponds to the **Checktime** variable, and the processor time demand C_j (assuming some processor capacity) corresponds to the ratio of the normalized processor demand time (the **TimeAt1MIPS** variable) and the processor capacity (the **MIPS** variable). Tasks indexed by j in the formula are named $j_1, j_2, \ldots$.

Table 3.2: Some constraint predicates used

Syntax	Semantics
#<=(E_1, E_2, V_B)	$V_B = 1$ iff $E_1 \leq E_2$; otherwise $V_B = 0$
#=(E_1, E_2, V_B)	$V_B = 1$ iff $E_1 = E_1$; otherwise $V_B = 0$
ceiling_d(V_X, V_Y, V_Z)	$V_Z = \lceil \frac{V_X}{V_Y} \rceil$
E_1 #>= E_2	$E_1 \geq E_2$

E_n is a linear expression including program variables, V_x is a (finite-domain) program variable

After the constraints have been set, the domains of the Task_i_on_Proc_k_Checktimes are reduced. They are subsequently further reduced as the Task_i_Processor and then Arrivals_of_Task_j_on_Proc_k_by_Task_i_Checktime variables have been labeled. A solution could be found if all the Task_i_on_Proc_k_Checktimes had non-empty domains. This way of searching for values of t that satisfy the formula is notably different from the one used in some RM schedulability analysis tools (e.g., [84]), which checks the formula for values of t that are multiples of the tasks' periods in a generate-then-test manner.

If the variable Arrivals_of_Task_j_on_Proc_k_by_Task_i_Checktime is assigned the smallest value possible, then the minimum value from the resulting domain of the variable Task_i_on_Proc_k_Checktime will be an upper bound on the earliest completion time of task i in the worst case. The difference between the deadline of task i and this value is thus a lower bound on its worst-case maximum laxity. The smallest such lower bound among all tasks allocated to a processor, multiplied by the processor's capacity in MIPS, may be used as the lower bound on the maximum additional, unexpected load the processor can sustain so that no task fails to meet

its deadline. An optimal solution would maximize the smallest lower bound on the maximum additional sustainable load among all the processors.

The handling of channels and messages is similar. The real-time communication model is similar to the one defined in [64] for short messages (no pipelining) in multi-hop real-time networks, and based on the rate-monotonic scheduling of messages on channels. The routers are assumed to have enough processing and buffer capacity, and to insure (by delaying) that a periodic message is made ready at equidistant points in time on all channels it traverses. In the real-time communication-related constraints, some of the variables defined and computed by the routing-related part generated by the `common` module are used, such as `Message_`i`_on_Channel_`k (analogous to `Task_`i`_on_Processor_`k).

Chapter 6 presents a third compiler module, `rtlrms`, an extension of the `rms` module for modeling Real-Time Linux [12] with a RM scheduler.

3.3 Experiments

The RTSML specification of the system described in Section 3.1 was compiled to CLP code that contained 17,046 program variables and 30,466 model-based constraints. Most of these constrants were internally rewritten by ECL^{i}PS^e as several simpler constraints. The CLP code consists of predicates that, alone or combined, implement a problem-solving approach, such as search for an optimal solution, any solution, or repair of a solution.

The division of program variables (implemented in the two modules described in Section 3.2) was based on a natural approach to system design: the columns of Table 3.1 were represented by parameter variables (totalling 238), and task-to-processor and message-to-route allocations were represented by solution variables (totalling 104). (When a message is multicast, it is allocated one route for each destination task. Allocated channels are determined from the message-to-route allocations.) There were 7,104 supplementary variables that required additional labeling. (The other variables were uniquely determined by labeling the solution variables.) For each processor and channel (totalling 22), a cost variable is defined that contains a lower bound on the maximum additional load the processor (in kI) or the channel (in kB) could sustain if a task or message unexpectedly exceeded its declared resource demand, so that all tasks/messages allocated to the processor/channel still meet their deadlines. In the cost function, 1 kI is valued as good as 1 kB.

The experiments were conducted under the following scenario. The complex real-time system was specified such that initial input parameters, described in Table 3.1, took on values from the lower half of their ranges (i.e., domains), and the CLP program was run to solve the constraints for the output parameters, i.e., to allocate the tasks to processors and messages to routes/channels. After the system had been imaginarily deployed, some input parameters were changed, and the CLP program was run to redesign the system, i.e., in this case to reallocate the tasks and messages as necessary to satisfy all the constraints again.

There are eight applications in the system, each consisting of one of the sensor tasks and the tasks that process its data, as described in Section 3.1. Based on the

communication patterns, these applications form trees rooted in their sensor tasks. The changes in the input parameters (at the task, message and processor level) were modeled by changes at the application level, such as increased load due to increased amount of input data; reduced resource capacities due to uncontrollable activities outside the modeled system; and need for reduced response times (e.g., reduced periods and deadlines) in, for example, emergency situations. Five such successive changes were modeled with six input parameter assignments (IPA) described in Table 3.3.

Table 3.3: Scenario of parameter changes

IPA	Description
$P0$	Load, i.e., computation requirements and message lengths, in the lower half of their domains; available processor capacity 100%
$P1$	Load increases up to 15% wrt. $P0$ down the T0 tree
$P2$	Load down the T0 tree back to normal; load increases up to 15% wrt. $P0$ down the T7 tree
$P3$	Load down the T7 tree back to normal; available capacities of P4 and P11 drop to 8 MIPS
$P4$	P4 and P11 capacities back to normal; deadlines reduced up to 15% wrt. $P0$ down the T1 tree
$P5$	Deadlines down the T1 tree back to normal; periods reduced (i.e., priorities increased) up to 20% wrt. $P0$ down the T2 tree

Two problem-solving approaches were evaluated, using IPAs in the order as specified in the above scenario:

1. the conventional CLP search, in which the parameter variables are assigned the values of the current input parameters, the constraints are set, and labelings are performed to search for consistent values of the solution variables, and

2. the repair-based CLP approach, in which the parameter variables are assigned the values of the current input parameters, the solution variables are temporarily assigned the values from the previous solution, and repair labelings are performed that modify the temporary values until all the constraints are satisfied.

The experiments were run using ECL^{i}PS^e 4.0.2 on a 300 MHz single-CPU UltraSPARC running Solaris 2.6. It is difficult to find the relation between the constraint level and search time. Usually, the hardest problems are neither underconstrained nor overconstrained. The results are discussed in the following subsections (the search for optimal solutions has not been yet evaluated).

3.3.1 Conventional CLP Approach

Relatively consistent timings were obtained using a labeling heuristic that consists of a sequence of two partial labelings: the "smallest-domain and most-constrained first" variable and "smallest-to-largest" value ordering for the solution variables, followed by the "smallest-domain first" variable and "smallest-to-largest" value ordering for the supplementary variables. (Note that the value ordering is important for better approximating the costs, as described in Section 3.2.) The timings of constraint setting and labeling until the first solution is found are shown in Table 3.4. With the "smallest-domain first" variable ordering of the solution variables, some timings were about as good as for the repair-based approach below, but some were much worse.

Even though the conventional approach does not appear fast to be used in an incremental manner, it appears good for one-time applications. Compared to, e.g.,

Table 3.4: Conventional CLP approach timings

IPA	Setting constraints [s]	First solution [s]	Sustainable overload [kI, kB]
$P0$	13.11	7,350.09	10–8000, 0–130
$P1$	12.89	1,180.07	30–8000, 0–130
$P2$	13.09	137.03	10–8000, 0–130
$P3$	13.07	6,750.46	10–8000, 0–130
$P4$	12.86	333.80	20–8000, 0–120
$P5$	12.85	231.08	50–8000, 0–120

a search technique popular in the recent years, genetic algorithms (GA), it promises significantly better performance and ease of use on highly constrained problems, such as complex real-time systems, on which the GA perform poorly [41].

3.3.2 Repair-Based CLP Approach

A generic CLP program loop that uses the repair library of ECL^{i}PS^e is shown in Figure 3.3 for the purpose of explaining the results. The variables `Parameter`, `Solution` and `Cost` are lists of the corresponding program variables; the supplementary variables are "shown" in the figure together with the solution variables (i.e., also in `Solution`) for simplicity. A notable difference from the conventional approach is that the constraints are set using the input parameters' domains instead of their current values. The resulting constraint networks are thus weaker, but this is necessary in order to allow input parameter modifications.

The `repair_labeling` predicate finds conflicting variables and constraints, and tries to assign values to the variables that result in no further conflicts. The fastest repairs were obtained using a labeling heuristic for the solution variables that first

```
repair_loop :-
  setup_constraints_as_repairable(Parameter, Solution, Cost),
  (
    labeling(Solution),                 % find initial solution
    copy_term(p(Solution, Cost), S),    % trim associated constraints
    setval(solution, S),                % store solution
    output_solution(Solution, Cost),
    fail                                % undo labeling
  ;
    repeat,
      getval(solution, p(OldSolution, Cost)), % fetch old solution
      modify_parameters(Parameter),     % use new IPA
      tent_set(Solution, OldSolution), % set old solution as tentative
      repair_labeling(Solution, Cost), % repair tentative solution
      tent_get(p(Solution, Cost), p(NewSolution, NewCost)),
      copy_term(p(NewSolution, NewCost), S), % trim associated constraints
      setval(solution, S),              % store new solution
      output_solution(NewSolution, NewCost),
    fail                                % undo repair_labeling
  ).
```

Figure 3.3: A generic repair-based CLP program

assigns values to conflicting variables using the "smallest-domain first" variable and "smallest-to-largest" value ordering, and then variables involved in conflicting constraints using the "smallest-domain first" variable and "smallest-to-largest" value ordering. This labeling heuristic was followed by another one for the supplementary variables, in which the "smallest-domain first" was replaced by the "smallest-domain and most-constrained first" variable ordering.

The repair loop from Figure 3.3 was slightly modified to start with the IPA $P0$ and replace it by the IPAs $P1, P2, \ldots, P5$, successively. In this way, a current solution is checked against a new IPA and, if a repair is necessary, subsequently replaced by a new solution. This corresponds to a scenario in which the input parameters of a continuously-operating complex real-time system change dynamically, and the repair

Table 3.5: Repair-based CLP approach timings

IPA change	`modify_parameters` [s]	`tent_set` [s]	`repair_labeling` [s]
$P0 \rightarrow P1$	34.58	40.19	199.92
$P1 \rightarrow P2$	34.55	40.12	64.60
$P2 \rightarrow P3$	34.97	42.20	67.64
$P3 \rightarrow P4$	36.14	40.40	68.24
$P4 \rightarrow P5$	34.47	40.86	215.44

loop redesigns the system on-line. The solution variables were stored at the end of each iteration, but the supplementary variables were not, because they are local and dependent on the solution variables. Using their values under $P0$ as temporary in each iteration resulted in faster repairs.

Table 3.5 shows timings for assigning new values to the parameter variables (`modify_parameters` in Figure 3.3), assigning temporary values to the solution variables (`tent_set`), and repair labeling, relative to the moment the corresponding IPA change occured. The first solution under $P0$ was obtained using the conventional approach; this search time is not accounted.

Compared to the conventional approach, the described repair-based approach is considerably faster in incremental solving overall. However, this holds for relatively moderate parameter changes; for significant changes, this approach tends to yield timings similar to those of the conventional approach. The choice of the problem-solving approach depends on both the search time and the solving time (off- or on-line). In this approach, it is possible to further improve the search time by identifying temporarily stable parameters and incorporating their values in advance of the solving time, and also by dynamically choosing the labeling heuristics. Due to its inherent

scalability [5], and having in mind that model-oblivious repair heuristics were used and that ECL^{i}PS^e is not a heavily optimized system, the repair-based approach appears good for dynamic resource allocation in large-scale complex real-time systems.

Table 3.6: Successive solutions' distances

Next IPA	Conventional approach	Repair-based approach
$P1$	(12, 13)	(1, 6)
$P2$	(13, 21)	(0, 0)
$P3$	(15, 24)	(11, 28)
$P4$	(18, 22)	(3, 11)
$P5$	(19, 22)	(1, 5)

As an alternative for the fast repair, the repair-based approach may also be useful for minimally perturbing a previous solution. In this case, corresponding system reconfigurations are less considerable, which is sometimes desired. Some benefits of a perturbation-optimized heuristic include

- potential finding of a stable solution in situations when there is a pattern in the parameter changes, and
- reduced system reconfiguration enforcement time, which could possibly be comparable to the repair time.

A repair labeling heuristic for this purpose was similar to the one described above, with the only difference in the "smallest-to-largest" value ordering replaced by "previous-value first" for the solution variables. The timings of this heuristic are somewhat larger overall and have larger variance, whereas it is consistent in finding solutions that are relatively close to their predecessors. Table 3.6 compares the

distances between successive solutions found by the conventional approach and by this repair heuristic. The distance between two solutions is an ordered pair (tp, mr), where tp is the number of task-to-processor allocations and mr is the number of message-to-route allocations in which the solutions differ. (Notice that the solution to $P1$ satisfies the constraints under $P2$, too. The previous, speed-optimized heuristic found different solutions to $P1$ and $P2$ because of the temporary values of the supplementary variables, which were taken from the solution to $P0$ when repairing $P1 \rightarrow P2$. These values caused (artificial) conflicts, and since the speed-optimized heuristic does not try previous values of the solution variables first, it found another solution even though the old one had satisfied the constraints. However, this is a "false alarm" kind of situation that may be traded for faster repair in other situations.) The perturbation-optimized repair-based approach resulted in fewer task-to-processor and message-to-route reallocations triggered by dynamic changes in the input system parameters.

3.4 Scalability Issues in the Approach

The efficiency of a CLP problem-solving approach is highly dependent on the structure of a problem [51]. In general, the faster the search space is pruned by the constraint propagation, the easier it becomes to find a solution. A complex real-time system is in a CLP program modeled by a number of global and local constraints and variables, corresponding to integration-related and subsystem/component-specific constraints and system parameters, respectively. For the conventional CLP approach and a

tightly-integrated complex real-time system, labeling the global variables first might result in faster pruning of the search space. However, if loosely-integrated complex real-time systems were shown to scale better in the real-time sense (in the light of the discussion at the end of Section 2.1), there would be relatively less global constraints and variables (i.e., their number would slowly increase with the system size) in their CLP models, which would negatively affect the scalability with this problem structure-aware choice of labeling. On the other hand, for the repair-based CLP approach and a previous solution to a loosely-integrated complex real-time system problem, if the violated constraints were mostly local, the repair time might be almost independent of the system size. Otherwise, or if the complex real-time system were tightly-integrated, that would increase the probability of a need for global repair (due to a violation of global constraints). In these cases, the benefit of the repair-based approach might degrade.

3.5 On the Correctness of the Approach and Compilation

This section discusses correctness issues in the approach and RTSML-to-CLP compilation that are most relevant for users and compiler module developers. First, the semantics of the source (RTSML) and target (CLP) languages, and the power of the approach are addressed. Next, details and the correctness of the compiler modules used in the example of Section 3.1 are discussed. Finally, the correctness of the

approach and compilation are evaluated against a set of high-integrity compilation criteria.

3.5.1 Language Semantics and the Power of CLP Problem Solving

The RTSML is a declarative, relational and extensible language. Individual resources (**processors** for computation; **channels** for communication and I/O), activities (**tasks** for computation; **messages** for communication and I/O), and routing information of a complex real-time system are specified in a declarative manner, using RTSML attributes. Structural and other basic relations among these components and subsystems are determined by the nesting of RTSML elements (tags) and the values of attributes other than the **type** attribute, and can be intuitively interpreted from the DTD given in Appendix A. (The **sys** and **group** attributes are meant to be available for use by the modules, and are not used as structural information by the compiler core.) For example, by listing the **ids** of **processors** and/or **processor-groups** as the value of the **processors** attribute of a **task**, a relation "can be allocated to" is established between the task and designated processors. Real-time model-specific parameters and relations among the components and subsystems are determined by module-specific, extended syntax of the **type** attribute that is specified by the developer of each module.

The relations in an RTSML specification are viewed through the constraint model [65]. Unlike, e.g., the traditional relational model in which relations are represented

by ordered tuples, they are represented by finite constraint forms while the ordered tuples satisfying them are implicit. The dominant real-time model-specific parameters and relations naturally fit the constraint model, but it is not difficult to fit the basic relations into this model either. In the above example task-to-processor allocation relation, the ordered tuples are (*task*, *processor*) ordered pairs, while the relation "can be allocated to" can be represented by a constraint, in the general mathematical sense (e.g., a set membership constraint), that for the given task *task*, *processor* can only be one of the designated processors. For all supported syntax extensions and real-time models:

Definition. *The semantics of an RTSML specification is the set of all ordered tuples of all the system parameters such that all the basic and real-time model-specific constraints are satisfied.*

The system parameters include both those figuring in the basic relations, such as the task-to-processor allocations, and real-time parameters, such as the task periods. Different compiler modules consider different system parameters as input parameters and the others as output parameters, but they are all treated equally in RTSML. The mathematical domain and constraint types over a system parameter are not restricted. In the real-time scheduling theory, for example, constraints usually involve non-linear functions of real numbers.

In Section 2.2.1, it was said that an arbitrary conjunction of constraints in a CLP language is solvable if the domain of computation satisfies the existential closure of it. It is assumed that the satisfiability problem is decidable by the corresponding solver, which is sound and complete wrt. its domain and constraint types. Accord-

ingly, the semantics of such a CLP program, *without LP predicates*, is the set of all ordered tuples of values of the quantified variables for which all the constraints are satisfied by the computation domain. The ECL^{i}PS^e system allows the integration of different constraint solvers, e.g., FD and RIA. This is achieved via declaring program variables as belonging to multiple computation domains at once, and defining special constraint propagation handlers that make the different constraint solvers cooperate and maintain a consistency relationship between the multiple value domains of the program variables. Hence, the semantics of a CLP program with multiple integrated solvers/computation domains is defined similarly as above, and includes the multiple values of the multiply-declared variables for which all the constraints are satisfied by all the computation domains.

The goal of the RTSML-to-CLP compilation is to transform a complex real-time system problem to one that is decidable by one or more cooperating constraint solvers, by mapping from unrestricted mathematical domains and constraints to computational domains and constraints that can be handled by the constraint solver(s). In some cases, such as restricting a domain from reals to integers, the approach loses its completeness (and that is why the term transformation is used instead of reduction [86]). Namely, the CLP solver(s) may not find a solution to the transformed problem even if there exists a solution to the original problem. (In this section, a solution means an ordered tuple of all values of interest, of both input and output variables/system parameters.) Although the loss of completeness is difficult to avoid in practice, it can be controlled in many cases by, for example, changing the degree of approximation of real numbers by integers, and/or resorting to multiple constraint solvers with differ-

ent capabilities. On the other hand, the developers of compiler modules that handle different real-time models and their integration with other models must insure that the approach retain its soundness in all anticipated uses. In other words, a CLP program must not positively decide its transformed problem if the original problem has no solution. Beside the obvious need for a correct problem transformation (the main task of the compilation), this issue is related to the power of CLP problem solving.

Generalizing what was discussed in Section 2.2.3, a CLP solver needs enough information in order to return a (correct) solution to the transformed problem. There are correctness nuances, based on the amount of supplied information:

1. The CLP solver syntactically accepts stated constraints, even they exceed its solving power. An example are FD constraints `[X, Y] :: 0..10, X * Y #>= 5.`, because the second constraint cannot be linearized and decided. The result returned are the original domains of the variables, and is interpreted as "there might be solutions for the domains," which cannot be considered correct.

2. Some stated constraints do not exceed the solver's power, but the solution is not "good enough." For example, FD constraints `[X,Y] :: 0..10, X * 2 + Y * 3 #>= 5.` are decidable, but the solution are the original variable domains. It is interpreted as "there are solutions for the domains," which cannot, for example, help linearize an additional constraint like `X * Z #<= 5.`

3. By explicitly and significantly reducing the domains of some variables, the solutions to some stated constraints may become "good enough." In FD, this is

done by labeling the variables (instantiating them with consistent values), in RIA by invoking a strong-propagation algorithm to search for a small (bounded) consistent sub-interval of a variable's domain, etc.

The RTSML compiler module object-oriented interface includes a method that is supposed to list all program variables whose domains should be explicitly and significantly reduced in order to guarantee the correctness of solutions to the constraints generated by a module. The reduction is performed by CLP code generated by the compiler core. The following subsection shows excerpts of an example problem transformation and steps taken for ensuring the correctness of the CLP problem solving.

3.5.2 Correctness-Related Compilation Details

To summarize the presentation of the two compiler modules from sections 3.2.1 and 3.2.2, the meaning of the RTSML specification of the example complex real-time system consists of all solutions to the following constraints (stated informally with reference to the preceding sections, for brevity):

- Each **common** task is assigned a period, deadline and normalized execution time demand in ms, and is allocated a processor.
- Each **common** message is assigned a period and deadline in ms, length in kB, a source and a list of destination tasks.
- Each **common** route is allocated an ordered tuple of channels, source and destination processor.
- Each **rms** processor is assigned a capacity in idealized MIPS.
- For all **common** tasks allocated to a **rms** processor, (1) relative priorities are determined according to the RM criterion, and (2) the schedulability constraint in Equation 3.1 is satisfied.

- For each (*message*, *destination task*) ordered pair, a route is allocated from the processor allocated to the message's source task, to the processor allocated to the destination task.
- Without resource usage optimizations, allocating a route to a message means allocating all the route's channels to the message.
- The per-hop deadline of a message is equal to the message's deadline divided by the maximum route length over all its destination tasks.
- Each `rms` channel is assigned a bandwidth in MB/s.
- For all `common` messages allocated to a `rms` channel, (1) relative priorities are determined according to the RM criterion, and (2) a schedulability constraint similar to that in Equation 3.1, using the per-hop deadlines, is satisfied.

Both compiler modules use the FD solver only, so that all system parameters must be approximated by integers. Referring back to the `rms` processor-related FD constraints listed in Section 3.2.2, the meaning of the transformed problem consists of all ordered tuples of integers that satisfy FD constraints including the following (again, informally and for an illustration purpose):

- If the period of task i is less than or equal to the period of task j, the value of FD variable `Task_`i`_preempts_Task_`j is 1; otherwise 0.
- User, non-linear ceiling constraint predicate over FD is defined as `ceiling_d(X, Y, Z) :- Z * Y #>= X, Z * Y #< X + Y.`
- If task i is allocated to processor k, the value of FD variable `Task_`i`_on_Proc_`k is 1; otherwise 0.
- The CLP version of the RM schedulability constraint is a non-linear constraint over FD involving integer variables.

The `common` and `rms` compiler modules insure that their variables get properly instantiated by the time a solution is found. Some variables, such as the task periods, are input parameters and are instantiated in the beginning of the search; some are

trivially guaranteed to be instantiated, such as the `Task_`i`_preempts_Task_`j variable. Others ought to be labeled: solution variables, such as `Task_`i`_Processor`, and supplementary variables, such as `Arrivals_of_Task_`j`_on_Proc_`k`_by_Task_`i`_Checktime`. Finally, some variables are allowed to remain represented by non-singleton domains, such as `Task_`i`_on_Proc_`k`_Checktime`, which neither prevents other constraints from being solvable nor is a specific value of it needed at all.

It is easy to see from the above representative excerpts that, assuming that all the CLP constraints are decidable, the compilation is meaning-preserving in the sense that a solution to the CLP problem corresponds to a solution to the RTSML problem. In other words, the compiler-based approach is not complete, but is sound.

3.5.3 Evaluation Against the High-Integrity Compilation Criteria

Although the RTSML compilation is not an exercise in high-integrity compilation, the following criteria [105] help summarize this section.

1. *The high-level source language must have a target-independent meaning. It must be possible to deduce the logical behavior of any particular program [specification], independent of its execution on a particular target machine.*

   This property follows from the definition of the RTSML purely declarative semantics.

2. *This implies, among other things, that the source language must have a* mathematically defined semantics. *Otherwise, it is impossible to deduce what* should *be the effect of executing a particular program.*

   Although informally stated, the RTSML semantics is mathematically defined via the semantics of supported real-time models.

3. *The target machine language must also have a mathematically defined semantics. Otherwise, it is impossible to* prove *that the compilation translation is correct.*

   A CLP language also has a mathematically defined semantics.

4. *The compiler from the source to the target language must be correct. Hence it must be derived from the semantics of* both *the source language* and *the target machine's language.*

   The compiler correctness is to be shown for each module, in a way similar, but complete, to that presented in Section 3.5.2.

5. *To permit* validation, *the compiler for a high-integrity language must be* seen to be correct. *It must be written clearly, and must be clearly related to the semantics.*

   All compiler modules only transform constraints from RTSML to CLP, and follow the object-oriented compilation protocol to insure that the CLP solver(s) can decide the transformed problem.

6. *The target code produced by the compiler must be clear, and easily related to the source code. This gives the* visibility *to the compilation process that is a requirement for high-integrity applications.*

   The target code is a text file, and the names of CLP variables are explanatory.

7. *The semantics for both the source and target languages must be available for peer review and criticism.*

   Module-specific RTSML semantics are based on extant real-time models, and the semantics of model integration is also a real-time issue. The semantics of CLP languages is a well-studied issue.

3.6 Summary

An approach to the design and engineering of complex real-time systems, based on CLP and compilation from a high-level, domain-specific specification language has been presented. It is oriented toward the integration of various real-time models, applied to computation, communication and I/O. One basic and a rate-monotonic technology-based system model have been implemented and briefly described.

An example complex real-time system, consisting of dozens of tasks communicating and competing for real-time scheduled computation and communication resources, has been specified, and the approach was evaluated on the example. The timings obtained show that the approach has acceptable performance overall and that the repair-based CLP approach is promising one for coping with dynamic changes in the system parameters.

Scalability issues of the CLP problem-solving approaches have been analyzed in combination with scalability issues of target complex real-time systems. Correctness issues of the compiler-based approach have been discussed, such as the completeness of the CLP-based modeling and soundness of the CLP problem-solving.

The following three chapters describe work that covers a run-time front-end of the repair-based CLP approach. Namely, (1) a distributed instrumentation system to gather performance data from a complex real-time system; (2) a PAV tool to process and transform the performance data into the input parameters of the real-time system models of the target system; and (3) an integration of the first two, PAV-driven, repair-based dynamic system reconfiguration. Based on observed changes in the input parameters, either by a human operator or automatically, the CLP tool may be demanded to repair the current solution.

Chapter 4

A Portable and Flexible Distributed Instrumentation System

This chapter presents a basic, reference implementation of a portable and flexible IS called BRISK (Baseline Reduced Instrumentation System Kernel). Along with the description, approaches that provided the IS performance gains are discussed. Section 4.1 describes major objectives of distributed ISes, and approaches taken in the design and implementation of BRISK. Section 4.2 details the BRISK architecture and implementation. Results of evaluating its performance and scalability are given in Section 4.3.

4.1 Objectives and Approaches

BRISK is a simple distributed IS designed with the following goals in mind:

- to be portable to a majority of operating systems and platforms, and easily used with a wide range of parallel/distributed applications and systems, and
- to provide a robust implementation base for the development of future distributed monitoring and control systems.

Figure 4.1 depicts the concept of BRISK as a distributed IS kernel that could be extended to support higher-level, more specific monitoring approaches.

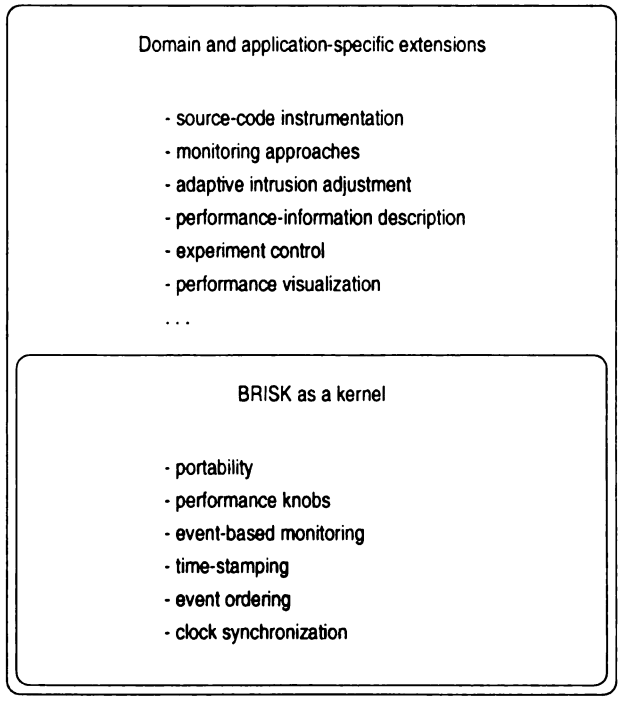

Figure 4.1: BRISK as an instrumentation system kernel

Design objectives concreting the above goals can be divided into three groups:

Performance flexibility. BRISK should offer high performance under various requirements. Simple metrics were evaluated first that are related to localized IS

performance, and to implement, evaluate and optimize relevant parts of BRISK in isolation. More complex IS performance metrics, related to IS operation on distributed applications, depend on the simple ones, plus the target application and system characteristics. Henceforth, "tuning knobs" were added to many of BRISK's subsystems, so that users can balance simple and complex IS performance metrics in a specific working environment.

Usage flexibility. BRISK should be able to support different monitoring applications, such as fine-grained instrumentation for testing, performance measurement for visualization and/or steering, etc. It should also be able to emulate various monitoring methods and techniques, such as hybrid (software instrumentation/hardware detection) monitoring for tracing or profiling. Without assuming specific hardware monitoring support, BRISK implements a generic software, event-based monitoring approach that allows (1) new users to start instrumenting their parallel/distributed applications quickly, and (2) then incrementally extend and/or specialize the approach.

On the other hand, the IS should be compatible with a variety of extant, independently-built tools and systems for the analysis of instrumentation data. Rather than impose a quasi standard of instrumentation data storage upon these tools/systems—and likely, in practice, render them unusable—it should be possible to adapt or extend BRISK to support a particular interface.

Finally, different parallel/distributed applications may require very different instrumentation/experiment scenarios. It is likely that no fixed IS control protocol

and/or graphical user interface can be powerful enough to support arbitrary scenarios of testing these complex systems. All these issues resulted in the design of BRISK as a general-purpose distributed IS *kernel*, based on a simple model described in Section 2.4.

Portability. BRISK, as general-purpose distributed software, should be able to operate in heterogeneous environments. This means that it should be designed and implemented assuming a minimal set of resources that are available in a majority of parallel/distributed environments. The basic implementation of BRISK relies on a small number of highly available systems libraries, such as those for interprocess communication through shared memory, External Data Representation (XDR), and TCP/IP protocols. (Shared memory IPC semantics may vary across different platforms, but this is a minor problem. It is important that there is probably no platform that does not support shared memory IPC.)

A number of important issues challenge the concept of a portable and flexible distributed IS. These include:

Degree of intrusion on the application. Due to the instrumentation overhead, a target system may exhibit different behavior. The overhead should be predictable and must not change the order and timing of critical events in the target system, i.e., cause perturbation. (In this dissertation, the term perturbation means only those changes in event timing that are significant from the target system perspective.). It is desired, especially for real-time systems, that IS components are schedulable with the target system, so that perturbation

analyses can be performed using schedulability theory. The BRISK design addresses this issue by defining a via-shared-memory event detection/notification protocol atop of which different event generation schemes may be used, and by having an event processing/delivery component as a separate process.

Global clock reference. Processes that make up a parallel/distributed system run on processors that may have non-synchronized clocks. It is very difficult to determine the precise global time and/or relative timings of events, which is mandatory for determining accurate global states and debugging erroneous timing behavior based on the instrumentation data. BRISK by default (1) chooses the best available systems clock call for a platform, (2) converts the current local time into the Universal Coordinated Time (UTC) format, and (3) uses a distributed clock synchronization algorithm. In this way, users who can afford synchronized hardware clocks may specialize BRISK for their type of platform. BRISK also provides a tunable algorithm for coping with network delays and minimizing the chance for the instrumentation data to be delivered to consumers out of order.

Throughput and latency of the instrumentation data transfer. Events of interest in a target system that are to be processed on-line may collectively form large volumes of instrumentation data and monopolize resources. On the other hand, in time-critical analysis, important events may need to be delivered to a central place as soon as possible. The IS should be able to adapt

to throughput/latency requirements of the target system. The basic BRISK implementation has a number of command-line parameters for this purpose.

Support for transparent monitoring. Adding significant amounts of instrumentation code to parallel/distributed systems by users is subject to errors. It is important that tools can be built based on the IS to instrument the target system automatically, so that the users need only specify *what* to monitor, from which aspect, and at which level. The IS design should be flexible enough to allow the development of such tools. This issue actually overlaps with the BRISK usage flexibility objective. The default software event generation is amenable to automation at the source level. Different aspects of the target system performance may be dynamically monitored after a tool has instrumented its source code, possibly with the user's guidance.

4.2 Description of BRISK

This section details the BRISK architecture and implementation.

4.2.1 Architecture

The architecture of BRISK is shown in Figure 4.2 (for less intrusion, a separate IS network is preferred, but not necessary). On each node, multiple user processes are instrumented using *internal sensors*. The internal sensors use cpp macros, described later in the Section 4.2.2, to write instrumentation data records to the shared memory. The shared memory is read by an *external sensor*, which runs as another process

on the same node and may be assigned a lower priority. Both the internal sensors and the external sensor (EXS) form an local instrumentation system (LIS) that sends instrumentation data to the instrumentation system manager (ISM). Time-stamps, embedded into the instrumentation data records by the internal and external sensors on different nodes, are synchronized. BRISK synchronizes LIS clocks using a modification of the Cristian's clock synchronization algorithm. This algorithm invokes a master-slave strategy whereby a master polls the slaves, determines differences between the values of its clock and the slaves' clocks, and updates the slave clocks.

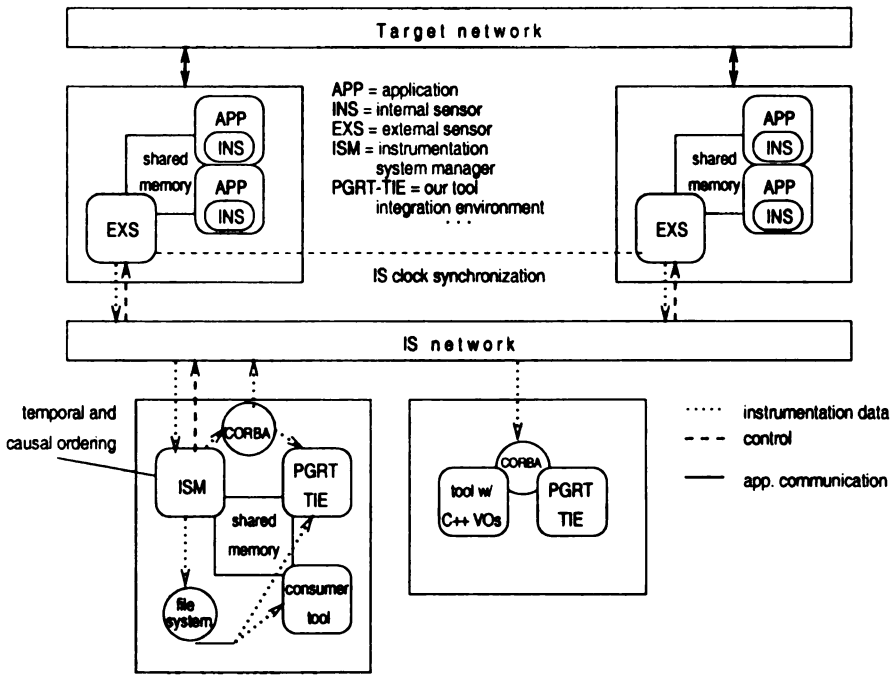

Figure 4.2: Architecture of the BRISK instrumentation system

The instrumentation data transfer protocol used between a LIS and the ISM is based on XDR, which makes BRISK amenable to a heterogeneous environment. In the ISM, the instrumentation data records are sorted on-line by time-stamp before

delivery to consumers. Special, causally-related events are additionally ordered, possibly overriding incorrect time-stamps. The default output mode of the ISM is writing to a shared memory, which is then read by instrumentation data consumer tools. Besides writing to shared memory, the BRISK ISM may log instrumentation data to trace files in the PICL-compliant [122] ASCII format, or it may pass instrumentation data to a list of CORBA-enabled *visual objects* [6].

4.2.2 Implementation

BRISK is written using the FWEB [38] literate programming package and may be compiled by a C or C++ compiler to create two executables, **exs**—external sensor and **ism**—instrumentation system manager; a header file with dynamically-typed *NOTICE macros*; a utility tool for creating faster and shorter, statically-typed NOTICE macros, and a tiny library for shared memory initialization and access. Figure 4.3 shows details of the implementation described below.

The main BRISK subsystems and components are described below. The presentation order follows that of the flow of instrumentation data from source (LIS) to destination (output of the ISM). The LIS time-stamps events with the help of the distributed clock synchronization algorithm. The transfer protocol forwards the instrumentation data over to the ISM, for on-line sorting and output.

Local instrumentation server (LIS). The BRISK NOTICE macros are internal sensors used in an application for event notifications. They are an extension of JEWEL cpp macros, which write a data record consisting of integers to a ring-buffer

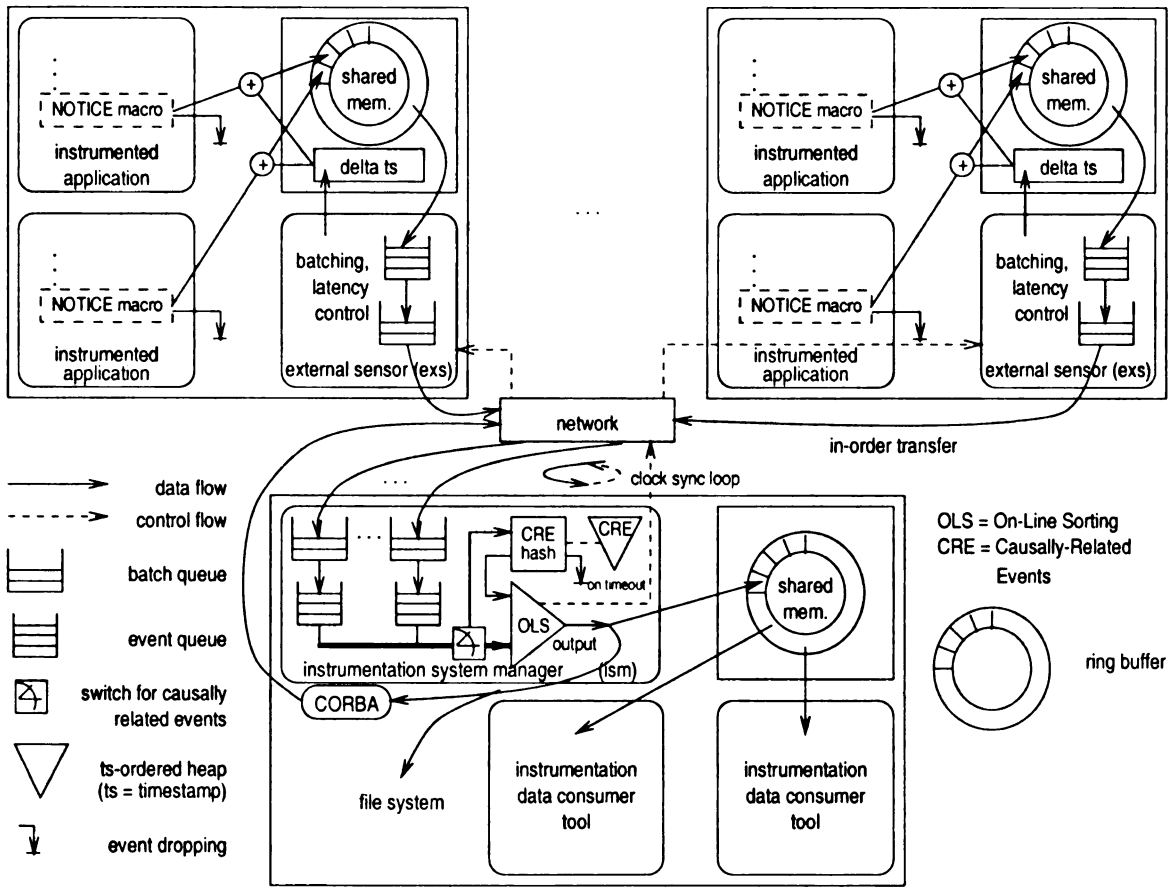

Figure 4.3: Diagram of the BRISK basic implementation

data structure in shared memory. The NOTICE macros are capable of writing heterogeneous records, with over ten basic types available for individual fields, ranging from bytes, to floats, to null-terminated strings. It is possible to modify certain parts of the NOTICE macros to affect the degree of intrusion and perturbation on the target application. The potential modifications that are not part of the basic implementation are based on the following.

- Regarding the intrusion, it can be lowered by, for example, partially evaluating the NOTICE macros in advance (as described at the beginning of this section),

or by dropping events in certain cases. Namely, the NOTICE macros use a simple ring-buffer locking mechanism to allow multiple instrumented programs to run on the same node; this mechanism could be busy-waiting, blocking or causing a NOTICE macro to be dropped if the lock is not free.

- Regarding the perturbation, the described lowering of the intrusion of the NOTICE macros is one way of avoiding it in an instrumented target application. Sometimes, a low level of perturbation in an instrumented target application may be acceptable; however, the perturbation issue may reappear if the instrumentation code is removed from the target application after testing. To avoid this reversed perturbation, the NOTICE macros may be slightly modified and left in the target application code. For example, they could be modified to write event data records to private memory when the application is not monitored, so as to take approximately the same time to execute; the drawback of the software approach is a change in memory reference patterns.

With the basic implementation, it is also possible to locally activate/deactivate event detection by dynamically toggling the value of a BRISK variable that is in the scope of the target application source code. Beside this simple option, BRISK provides a generic, flexible scheme for on-line remote control of the instrumentation and/or steering of the target application. Instrumentation data consumers can send data through BRISK in the other direction that are written in a shared memory area accessible to the EXS and target application. The organization and interpretation of

these data in the shared memory area are to be defined in a contract between the target application and a monitoring/steering tool, and can be easily automated.

Besides the data types for event data, three system types are available for coordination among BRISK, instrumented distributed applications, and instrumentation data analysis tools. A "time-stamp type," `X_TS`, allows the user to embed BRISK's internal time-stamp into an event record. The embedded time-stamp is an eight-byte `longlong_t`, representing the number of microseconds of Universal Coordinated Time (UTC). The local time is read by a call to `gettimeofday` library function embedded in the NOTICE macro code. (If `gettimeofday` is not available on a platform, a substitute should be provided.) The read value is then added to a *correction value*, maintained by the EXS. The result is a time-stamp of the synchronized EXS clock, embedded into the record before its sending to the ISM. The other two system types, `X_REASON` and `X_CONSEQ`, are used to mark causally-related events. The user supplies `u_long` identifiers for fields of these types, determining which *consequence* events must follow respective *reason* events. If BRISK's clock synchronization algorithm fails to prevent the occurrence of so-called tachyons, i.e., consequence events that appear to happen before their reason events, events marked using these system types will be post-processed by the ISM to ensure proper ordering. Without additional information, it is not possible to handle more than pairs of causally-related events. The technique described in Section 2.4 uses other event data records, specific to the PICL library, to gather information about causally-related send/receive events in collective communication.

An example NOTICE macro call is shown in Figure 4.4 (to save space, the macro code itself is not described). It writes an event record in the shared memory with the structure shown in Figure 4.5. All fields are aligned according to their types.

```
char *c = "Job XYZ";
float percent_done = 58.7;
NOTICE_3(X_TS, TIMESTAMP, X_STRING, c, X_FLOAT, percent_done);
```

Figure 4.4: An example three-field NOTICE macro call (internal sensor)

```
8-byte internal time-stamp (ITS) in UTC format
1-byte X_TS type (not followed by a value; ITS will be used instead)
1-byte X_STRING type
null-terminated "Job XYZ" (8 bytes, assuming 1-byte chars)
1-byte X_FLOAT type
sizeof(float)-byte 58.7
1-byte end-of-record/end-of-buffer marker
```

Figure 4.5: In-memory structure of the event record generated by the call in Figure 4.4

A goal is to provide the convenience of dynamic typing to new users, and, at the same time, retain the form of a macro for lower intrusion. The header file contains NOTICE macros for up to eight dynamically-typed fields, which are likely to be sufficient for many uses. More than eight dynamically-typed fields in a macro adds excessive code to a compiled instrumented application, which indirectly increases the intrusion. A utility tool is provided, accompanying the basic implementation, to create custom NOTICE macros having user-defined field types and insert them into the header file. This tool effectively supports an *on-demand* partial evaluation/specialization of NOTICE macros that results in smaller and faster code, and thus lower intrusion. It

exemplifies the flexibility of BRISK and can be used as a starting point for the other modifications of NOTICE macros mentioned above.

Distributed clock synchronization algorithm. Each LIS takes part in the distributed clock algorithm. The main difference between the BRISK algorithm and the Cristian's algorithm is that the master (ISM) time is used only as a common reference point for computing relative skews of the slave (EXS) clocks. That is, instead of directly adjusting the EXS clocks based on their skews relative to the ISM clock, like in the Cristian's algorithm, the BRISK algorithm only considers relative differences between the EXS clock values for the adjustment purpose. This modification is due to the fact that, for measurement purposes, it is important that the EXS clock values be as close to each other as possible, while it is not necessary for them to be close to the ISM clock value. At the cost of small positive drifts of the EXS clocks, the algorithm described below converges faster than the original Cristian's algorithm. Other minor differences between the two algorithms are (1) in the criterion for the measurement discards; and the absence of some optimistic enhancements in the basic BRISK implementation, such as (2) dynamical changing of the number of queries in a round based on the observed probability of a successful query; and (3) inclusion of the EXS hardware clock drift in the adjustment calculation.

Firstly, an EXS clock with the maximum positive skew relative to the ISM clock, i.e., one with the most-ahead clock, is selected, based on polling as in the Cristian's algorithm. Then, the skews of the other EXS clocks and their average are computed *relative* to the selected EXS clock, as absolute values. Finally, only the EXS clocks whose relative skews are greater than the average, i.e., only those slower than the

average, are advanced by a correction value. The purpose of this restriction is to account for the network noise and, in a conservative manner, take care not to promote another EXS clock as the fastest one erroneously. The price of this decision is potentially slower convergence. The correction value is chosen as follows: if the average value is above a small threshold, e.g., 3 times the number of EXSes, in microseconds, the correction value is set equal to the relative skew of the corresponding EXS clock; otherwise, the correction is set to a fixed fraction of the relative skew, 0.7 in the basic implementation. This reduction of the correction value is also conservative in nature, because the EXS clocks cannot be perfectly synchronized in practice.

Additionally, the algorithm was modified to detect and discard outliers occurring in a single polling period, up to a predefined number. This modification accounts for anomalies, such as unexpected, large clock value differences obtained during the polls.

Transfer protocol (TP). For instrumentation data transfer between the LIS/EXS and ISM, the XDR protocol has been chosen in the basic BRISK implementation. BRISK's data transfer protocol does not, however, use XDR in the typical way, with **rpcgen** and static typing, as in JEWEL. Instead, each dynamically typed instrumentation data record is sent with a meta-information header needed for it to be correctly received. The external sensor packages instrumentation data in XDR format with the meta-information header compressed, and sends them to the ISM over a TCP stream socket. Minimizing the slack in instrumentation data messages is important since transferring of likely large volumes of event records through the network is several orders of magnitude slower than through shared memory.

Instrumentation system manager (ISM). When the ISM receives an instrumentation data batch from an EXS, it stores it in the corresponding queue; the in-order arrival of these batches is guaranteed by the socket stream protocol. For dynamic merging/on-line sorting and extracting instrumentation data records from multiple queues, the ISM uses a heap having one entry for each queue.

Each instrumentation data record, after being extracted from the ISM's heap, is written to a shared memory buffer using the same binary structure used by the NOTICE macros. Optionally, a PICL-compliant trace record can be generated with the time-stamps either in the UTC format or as the floating-point number of seconds passed since the ISM has been run. The visual objects mentioned in Section 4.2.1 are components of an object-oriented framework for the development of on-line performance visualization of parallel/distributed systems. Through an optionally linked, portable implementation of CORBA 2.0 called MICO [94], the ISM can call remote visual objects' methods and pass instrumentation data records to be processed as PICL-compliant strings. Other consumers can read the ISM's shared memory buffer, e.g., using BRISK library code that creates such strings.

On-line sorting algorithm. Before the ISM finally delivers instrumentation data to their consumers, it uses the synchronized embedded time-stamps, the current time, an estimated clock difference between the ISM and the most-ahead EXS, and a user-specified time frame T to delay the processing of each instrumentation data record for T time units after the record creation. If the ISM detects that two successive records from different external sensors have been extracted from the heap out of order, it increases the time frame; then, it exponentially decreases the time frame to reduce

the latency of event processing and the amount of instrumentation data in the queues. This method of sorting results in a tradeoff between event ordering and latency.

Additionally, causally-related events are matched via a hash-table: if a consequence event record being processed does not match a reason event record with the same identifier in the hash-table, it is kept in a data structure until the corresponding reason event record is processed. When a just-arrived reason event record matches a waiting consequence event record whose time-stamp is smaller than its own, the latter's time-stamp is overridden by a larger value. Since in this case it is obvious that the clocks have not been synchronized, i.e., the time-stamps should reflect the causality, an extra round of the clock synchronization algorithm is invoked immediately. A causally-marked event of either type is kept in the data structure no longer than a specified timeout, because its peer may have been dropped.

The on-line sorting algorithm assumes that the EXS clocks are perfectly synchronized. Since this is not achievable in practice, tachyon occurrences are possible if the EXS clock value difference is greater than the time taken between the causally-related events, e.g., the time it takes to send a message between two nodes. (For example, when node A receives a message from node B, the event of receiving the message is causally-related to the event of sending the message. Node A's clock could be t_1 units behind node B's clock, but it also takes t_2, possibly smaller than t_1, time units for the message to arrive.) Tachyons can pass through the ISM if causally-related events in the target application are not marked using BRISK. In fact, instrumenting some causally-related events using BRISK may help BRISK maintain better EXS clock synchronization. (Note that users may define application-specific causally-related

events that are not necessarily send and receive events. However, there is danger that incorrect causality assumptions may deteriorate the BRISK clock synchronization.) The extra synchronization rounds would, in turn, reduce the probability of tachyon occurrences resulting from other causally-related events.

4.3 Evaluation of BRISK

Table 4.1 summarizes the relationships between (1) the design criteria in Section 4.1, and (2) the actual BRISK architecture and implementation presented in the previous section and experimental results presented in this section. The relevance of a result with respect to a criterion can be low (L), moderate (M), high (H) or none (-). Similarly, the success of a result in satisfying the corresponding criterion can be poor (P), fair (F), high/with special attention (H) or not applicable (-).

Table 4.1: Summary of BRISK evaluation

		Implementation results								
	Relevance/Success	LIS design	TP design	ISM design	NOTICE timing	EXS CPU usage	Event throughput	Event latency	Clock synchronization	On-line sorting
Design criteria	Performance flexibility	H/H	M/F	H/H	H/F	H/H	H/H	H/F	H/H	H/H
	Usage flexibility	H/H	M/F	H/H	M/F	L/F	M/H	M/F	H/H	H/F
	Portability	H/H	H/H	H/H	-/-	-/-	-/-	-/-	H/H	M/H
	Low intrusion	H/F	H/F	L/F	H/F	H/H	H/F	M/F	L/F	L/-
	Global clock reference	L/F	L/-	L/F	M/F	M/F	M/F	H/F	H/H	H/H
	Throughput and latency	M/F	H/H	H/H	L/F	M/F	H/H	H/F	L/-	L/-
	Transparent monitoring	H/H	L/-	M/H	L/-	L/-	L/-	L/-	L/-	L/-

The table entries containing at least one "H" follow from the preceding and following presentations. The "H/F" entries indicate aspects in which the basic implementation should be specialized in order to satisfy the criteria in extreme cases of target system behaviors and/or monitoring requirements. The "M/F" and "L/F" entries, based on available knowledge, may also imply rare use cases in which BRISK may need a design or implementation modification in order to satisfy a criterion which has become negatively affected by the basic implementation.

Experiments have been conducted with BRISK using two configurations. The first configuration consists of one external sensor; the range of several simple performance metrics were measured. The second configuration includes multiple external sensors on different nodes; the system scalability was measured, as well as the quality of the clock synchronization and dynamic on-line sorting. In both configurations, loop-based synthetic applications were used that were instrumented using NOTICE macros having six fields of type `integer`. Including the time-stamp and type information, each instrumentation data record requires 40 bytes in the XDR-based transfer protocol. The experiments were executed primarily on Sun Ultra-1 workstations running Solaris 2.5.1 within a 155 Mbps local ATM network. Eight nodes were available for measurements under the second configuration.

4.3.1 Local Performance

NOTICE macro (internal sensor) timing. The CPU time taken by a six-integer NOTICE macro to write to shared memory is given in Table 4.2 for three platforms.

The experiments were performed with 100,000 macro calls. The timings compare well to those of the simpler, low-intrusion JEWEL macros. Hence, with the options of NOTICE macro modifications for further reducing the degree of intrusion and the possibility of perturbation on the application, BRISK approaches an optimum that can be achieved with software, portable event generation and detection.

Table 4.2: CPU time per 6-integer NOTICE macro

Platform	μs
150 MHz SGI Onyx 4×R4400 / IRIX 6.2	18.6
195 MHz SGI Indigo2 R10000 / IRIX 6.2	8.7
140 MHz Sun Ultra-1 / Solaris 2.5.1	3.6

External sensor CPU use. Besides the internal sensors, the external sensor (EXS) is another component of BRISK that could interfere with a target distributed system. It is a separate process running on the application nodes, and it therefore competes with the application for machine resources such as CPU time. CPU utilization of the EXS was measured for six different event rates, with one message (batch) buffer of size 8 kB. The `top` tool was used to save CPU utilization to a file at 5-second intervals. The application was started approximately 10 seconds after starting `top`, and it ran for 60 seconds. Figure 4.6 shows the results.

The CPU utilization of the EXS is shown to be negligible at event rates of up to 38,000 per second. At 41,000 events per second, the EXS uses approximately 3%, which may still be insignificant for some applications. In practice, a few thousand events per second should be enough for fine-grainly instrumented real-time applications. At these rates, the EXS process could be assigned a lower priority (on a

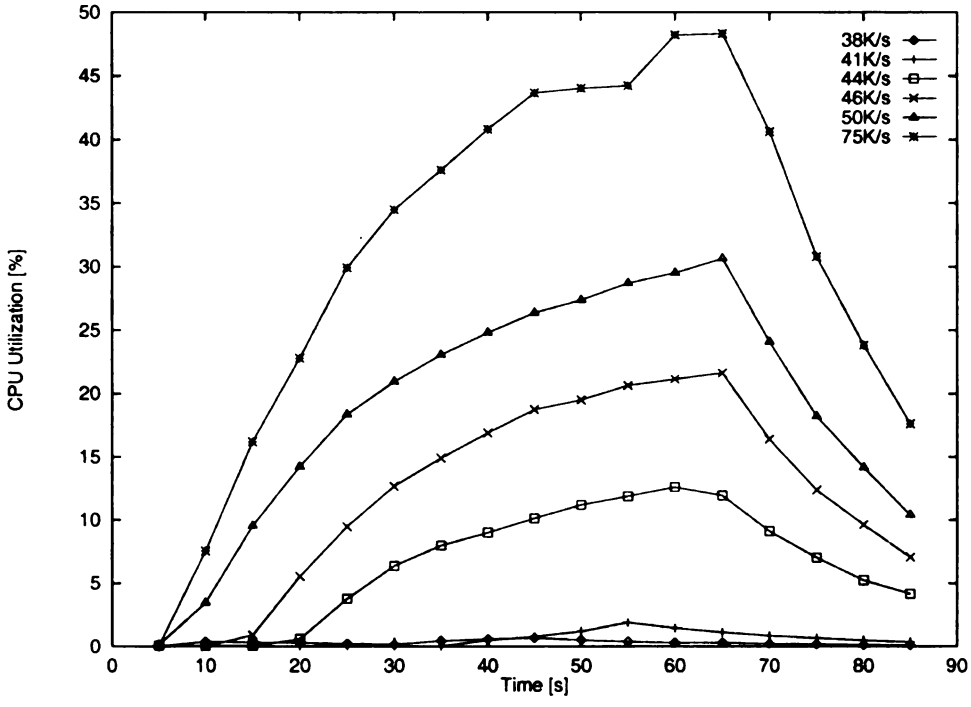

Figure 4.6: EXS CPU utilization for various event rates

conventional or real-time operating system) to further reduce the degree of intrusion and the chance for perturbation.

Event throughput. A synthetic application was used to generate event records at various fixed rates. In order to obtain the relatively high event rates desired, it was necessary to generate events in batches, pausing between the batches instead of between successive events. A **nanosleep** system call was used to create the pause between batches. The time resolution of this function is platform dependent; the shortest sleep time that could be achieved on the Sun workstations ranged from 10 to 20 milliseconds.

Throughput was measured for various values of two parameters: (1) the size of the internal/external sensor shared memory buffer, and (2) the maximum size of messages

that carry events from the EXS to the ISM. A single application process was used to generate event records. Each experiment was executed for a duration of one minute. To determine the maximum throughput, experiments were repeated using successively higher event rates, until the internal or external sensor began to drop events. In an experiment in which the maximum throughput was found to be around 90 thousand events per second, shared memory size of 2 MB was used with a message size of 65536 bytes, including 4 bytes for the header and 64 message buffers. The size of shared memory buffer was large enough to preclude it becoming a bottleneck. At the receiving end, the ISM is run with a command-line parameter that specifies its responsiveness to incoming event records. It sets the number of event records that can be processed/output without an interruption to receive incoming events. If this parameter is set sufficiently low, and the incoming event rate is sufficiently high, the ISM will receive incoming events faster than it outputs them. This is an unstable situation, since the ISM continues to allocate additional memory to hold the events waiting to be output. However, setting this parameter with an appropriate value may allow the ISM to handle higher peak rates of incoming events while allocating only a small amount of additional memory for events pending output.

To quickly adapt BRISK to a target application throughput and latency requirements, as well as to minimize its memory usage, the EXS and ISM provide several command-line parameters. These include all the above mentioned ones, and three more for the EXS: the maximum number of events to read from the shared memory in one turn; the approximate shared memory buffer polling period; and the approximate batch sending latency.

4.3.2 Distributed Performance

Event throughput. When multiple EXSes send instrumentation data to the same ISM, their maximum sustainable event rates are dependent on each other and also on event-generation/transfer patterns. Using approximately uniform event-generation/transfer patterns across the EXSes, the maximum per-EXS and aggregate event rates were measured to investigate potential bottlenecks.

In this experiment, the ISM's responsiveness command-line parameter was tuned with the increase of the number of EXSes in order to find a balance between the receiving and processing rates and to achieve an approximate maximum throughput of 90 thousand events per second. The on-line sorting was effectively turned off by setting the time frame T to 0, so that it does not affect the responsiveness. A significant factor here is the minimum blocking time of a `select` system call that separates the receiving and processing phases. To balance the flow of events through receiving, processing and blocking phases, the processing rate must be sufficiently high to keep up with all the instrumentation data that arrives during the phases.

The results show that the aggregate event rate is limited by the CPU capacity and dependent on the kernel clock quantization which affects the `select` system call. The rate remained almost constant as more EXSes were added. This low dependence on the number of EXSes is favorable toward future BRISK extensions that include scalable, hierarchical instrumentation data collection with local ISMs forwarding instrumentation data to a global ISM.

Clock synchronization. As explained in Section 4.2, the ISM polls EXSes in periodic rounds. At the end of each round, the clocks of the EXSes are updated with the time difference to the fastest EXS clock. Three metrics for evaluating the clock synchronization were selected:

1. the mean relative error, which is computed at the end of each round of polls, is simply the mean of the absolute values of differences (skews) between each EXS clock and the fastest one;
2. the time required for the mean relative error to converge, i.e., the number of rounds after which the mean relative error falls below a threshold value. This time also depends on the polling period;
3. the maximum relative error, which is also computed at the end of each round of polls, is found by taking the difference between the fastest and slowest clocks. It provides some indication of variance.

To perform a larger number of experiments, the polling period was reduced from a default of sixty seconds to five seconds. Because the system clocks have less chance to drift apart with five-second polling periods, the results obtained were expected to be better than what would be obtained with sixty-second periods. Figure 4.7 shows the performance of the BRISK clock synchronization on relatively lightly loaded processors and network. The mean and maximum relative error in microseconds are plotted over time after start of the algorithm, which is specified as number of polling periods or rounds.

Although the peaks in Figure 4.7 are relatively insignificant, the conditions were investigated under which the peaks occur. The algorithm is dependent on, among other things, the implementation of **gettimeofday**, i.e., the type and precision of the clock used, and network conditions.

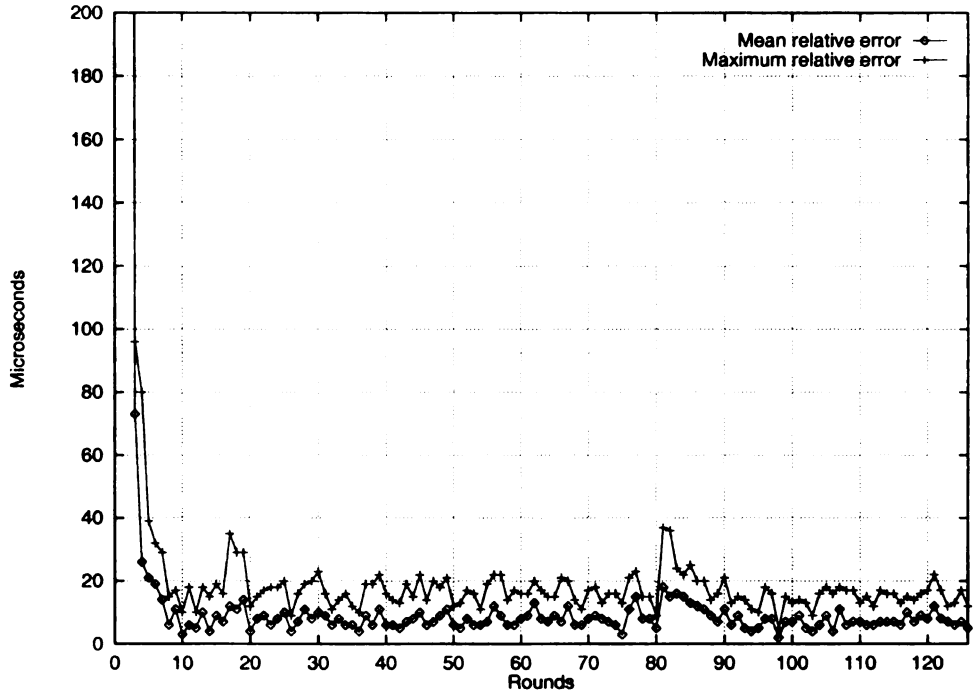

Figure 4.7: Measurements of the clock synchronization algorithm (8 EXS nodes, 5-second polling period, 10-minute experiment)

Improvements to the basic clock synchronization algorithm, including discarding of outliers, did not substantially reduce the occurrence of the peaks in the mean relative error. With further study, the first observation was that the peaks were caused by more than one node during an experiment. This observation eliminates the possibility of a single machine having an errant clock. Second, the clock errors occurred both above and below the expected value during a single round. These within-round variations imply that the Unix clock synchronization daemon is not the cause, since the daemon does not change the clock more than once within a few seconds. Finally, some correlation was observed between the occurrence of peaks and two other factors: (a) the time of day, and (2) other activity on the computers. For example, experiments were performed around the same time on separate trials.

Each time, the frequency of the peaks was greater at 11:30 PM than what had been measured just thirty minutes earlier. Thus, isolating the factor affecting the quality of clock synchronization, if needed, will require controlled experimentation.

The average synchronization of EXSes within a few tens of microseconds on a fast local network is likely to be an acceptable cheap substitute for synchronized hardware clocks for the majority of target applications. Similarly, the observed drift of EXS clocks relative to the ISM clock of up to a few tens of milliseconds per hour can often be tolerated. The command-line parameter for the clock synchronization period can be used to balance between the quality of the clock synchronization and the negative effect that messages sent by this distributed algorithm, which have the priority over instrumentation data messages, may have on the instrumentation data latency and throughput.

Dynamic on-line sorting. In order to test and evaluate the dynamic on-line sorting algorithm, which attempts to nullify variations in the latency of instrumentation data arriving from different nodes, code was added to BRISK to simulate delays in the network. This code passes event records for a random period of time. After this period, an event record is delayed for some other random time. The amount of time by which an event record is delayed is determined by subtracting a random number with a bounded, uniform distribution, from its time-stamp. An exponential distribution was used for the time between delayed event records, as it has the desirable "memoryless" characteristic for this type of simulation [61]. Event records are considered late when both of the following occur: (1) their delays are greater than

the time frame T, the on-line sorting delay described in Section 4.2.2, and (2) the ISM has already output an event record with a more recent time-stamp.

The sorting was evaluated with a set of experiments in which four parameters were varied: algorithm for time frame T; time frame half-life; maximum delay time; and mean time between delayed events. These parameters are described as follows:

1. An algorithm increases the time frame T upon the arrival of a late event record. The two options for this algorithm were either to double the current value of the time frame (doubling algorithm) or to set the time frame to the lateness of the event record (lateness algorithm).

2. The time frame half-life (`hl`) is the amount of waiting time before reducing the time frame when no event records are late. The two options for `hl` were either 2 s or 10 s.

3. While the minimum event record delay was fixed at 50 ms, the maximum event record delay time was variable. The two options for this parameter were either 200 ms or 500 ms.

4. The exponential distribution is parameterized by the mean time between delayed event records. The two options were either 1 s or 2 s.

The following performance metrics were used in these experiments to evaluate the on-line sorting algorithm: the frequency of increasing the time frame T and the peak value of the time frame T. Each experiment was performed twice. Eight EXSes were used, and the aggregate event rate was approximately 2,000 event records per second. The duration of each experiment was 125 seconds, resulting in 256,000 event records.

Table 4.3 shows the frequency by which the time frame T was increased due to late event records. Table 4.4 shows the peak magnitude of the time frame T, which correlates to the event latency. In both tables, the first column lists the simulation

parameters used: minimum event record delay (`min`), maximum event record delay (`max`), and mean time between delays (`MTB_delay`).

Table 4.3: Count of increases in time frame T

		doubling		lateness	
`min/max/MTB_delay`	Trial	`hl` = 2 s	`hl` = 10 s	`hl` = 2 s	`hl` = 10 s
50 ms/200 ms/1 s	1	48	8	16	9
	2	11	10	85	24
50 ms/200 ms/2 s	1	38	11	20	25
	2	14	4	61	22
50 ms/500 ms/1 s	1	44	13	33	29
	2	48	13	21	13
50 ms/500 ms/2 s	1	34	11	78	14
	2	47	12	56	14

Table 4.4: Peak time frame T in milliseconds

		doubling		lateness	
`min/max/MTB_delay`	Trial	`hl` = 2 s	`hl` = 10 s	`hl` = 2 s	`hl` = 10 s
50 ms/200 ms/1 s	1	800	800	429	268
	2	200	800	505	452
50 ms/200 ms/2 s	1	800	400	373	391
	2	400	400	374	391
50 ms/500 ms/1 s	1	800	3200	1589	1799
	2	3200	3200	1016	1524
50 ms/500 ms/2 s	1	1600	800	1194	1177
	2	1600	1600	808	1274

In Table 4.4, higher noise levels on the network, represented by higher `max` and lower `MTB_delay` values, resulted in higher latency. However, this elevated noise level did not result in a significantly higher number of late events, as shown in Table 4.3. These two observations suggest that the dynamic on-line sorting algorithm is adapting to the noise level on the network.

The non-parametric Wilcoxon signed rank test [16] has been applied to analyze the effect of the first two parameters on the performance metrics and numerically support qualitative analysis.

When comparing the results from the doubling algorithm to those from the lateness algorithm in Tables 4.3 and 4.4, the following observation was made. The lateness algorithm resulted in a higher number of late events, at a confidence level of about 90%, but smaller peak values of time frame T, at the 99.22% confidence level. This higher number suggests that the lateness algorithm is better suited for latency-critical applications, while the doubling algorithm is better for order-critical applications.

A comparison of the results from the 2 s half-life to those from the 10 s half-life shows that reducing the time frame T half-life increases the number of late events, at the 99.99% confidence level. No evidence was found for this to have recognizable effect on T's peak value. This independence of T's peak value on T's half-life implies that the half-life may be a large value for most applications, except those that are latency-critical.

The minimum value of the time frame T, the choice of the algorithm for its increasing, and its half-life are available as command-line parameters, and so are the maximum times for keeping reason and consequence events in the ISM hash tables.

Event latency

Extensive measurement of the event latency, which is affected more significantly by multiple EXSes than is throughput, is part of our future work on BRISK. Network noise, combined with dynamic on-line sorting, is one source of latency. Another source

is waiting `select` system calls, which are used in both the EXS's and ISM's main event loops to avoid busy-waiting. Namely, the duration of waiting `select` system calls depends on the resolution of the kernel clock, which has been found to be 10–20 ms. While the event processing time taken by EXS and ISM is in the range of tens of microseconds, two `select` calls can add up to about 40 ms latency in a worst case.

4.4 Summary

BRISK is a lightweight distributed instrumentation system developed with an emphasis on portability and flexibility. It exploits successful technical solutions of extant distributed ISes in order to provide a robust kernel for future distributed monitoring and control systems. So far, it has been ported to Sun Solaris, SGI IRIX, Linux and, using AT&T U/WIN [68] and Cygnus IPC library, to the MS Windows family.

Characteristics of BRISK that were evaluated included event throughput, clock synchronization, intrusion on the instrumented system, and dynamic on-line sorting. Measurements indicate that BRISK could meet the performance requirements of a wide range of distributed applications. Being a flexible and open IS, it should be relatively easy to adapt it to a specific environment and improve its performance in a specific direction.

Chapter 5

An On-Line Performance Visualization Technology

This chapter describes a framework for on-line performance analysis and visualization called PG^{RT} visual objects. It is object-oriented and easily distributable via middleware software such as CORBA [82] and DCOM [20]. Within it, a visual-object developer can integrate low- and high-level, application-specific PAV. Furthermore, it is based on two visual-object levels for portability and code reuse: a device-dependent low level, and a device-independent high-level. A goal was also to be able to integrate various sources of off- and on-line performance data. To achieve this flexibility, the visual objects consume performance data in the form of *event records* from an environment. To formalize the design of high-level visual objects, i.e., enforce a structured approach that is less error-prone, certain rules and a very high-level, component-based specification language, called Visual Object Markup Language (VOML) have been

defined. The language uses Standard Generalized Markup Language (SGML) markup for structuring visual objects, and Scheme scripts for defining PAV semantics.

The use of SGML enables development of a PAV information infrastructure for platform- and tool-independent development of visual objects. It may also facilitate automatic monitoring, analysis, and visualization of globally distributed applications via network-enabled SGML entity managers. The use of Scheme for visual object semantics enables both rapid prototyping of visual objects and customizing VOs for a wide range of platforms via, for example, Scheme-to-C and Scheme-to-Java VM bytecode compilers. That is, a single VOML specification may be used to generate automatically an X library-based visual object and one that runs within a WWW browser.

In Section 5.1, the visual-object framework is described in detail, and an example of successful use for PAV of a distributed multimedia real-time application are shown. A PAV architecture for high-level visual objects, the markup language based on it, and the development environment are presented in Section 5.2. An example of a VOML specification is given in Section 5.3.

5.1 Visual Object Architecture

The Visual Object (VO) architecture identifies two main software layers apparent in the majority of extant PAV tools, and represents them as two classes: the *high-level VO (HLVO) class* and the *low-level VO (LLVO) class.* In general, the responsibility of an HLVO class is to implement an application-specific semantics, while an LLVO

class is platform-dependent while providing a platform-independent interface to the HLVO class. When implementing a VO class, an HLVO class implementation is derived from an LLVO class implementation, as shown in Figure 5.1. (Vertical bars in a high-level method denote the presence of multiple peer components.) In the following subsections, the main characteristics of the LLVO and HLVO class, and an application to a heterogeneous system are shown.

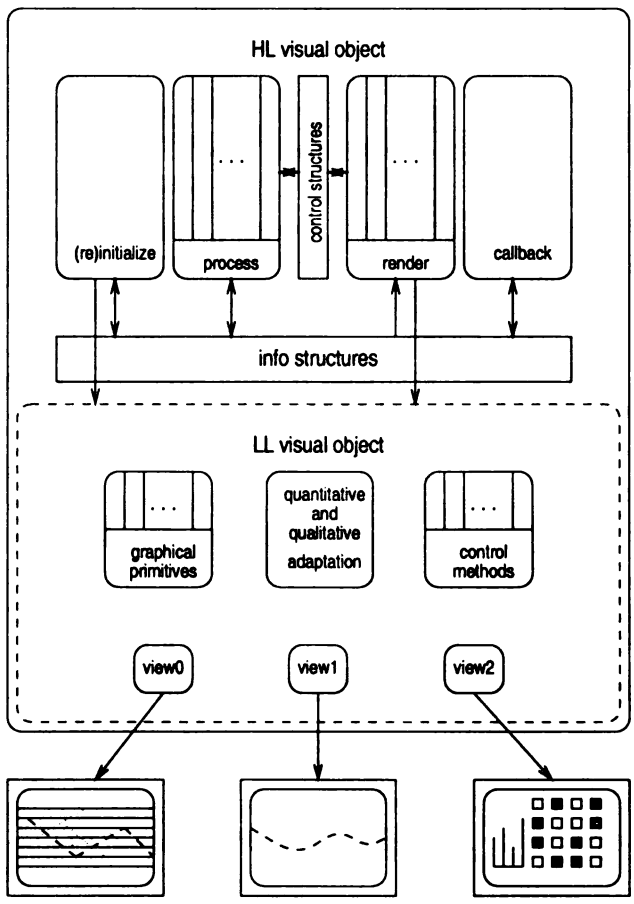

Figure 5.1: The design of a visual object

5.1.1 Low-level visual object

The responsibilities of an LLVO class described below illustrate the basic building block of this PAV technology. They have evolved by repeat of substantial experimentation with an X library-based two-dimensional LLVO class implemented in C++.

Multiple views. An LLVO maintains a number of display areas, referred to as *views*. In the Xlib-based implementation of the LLVO class, each display area is supported by a contained object that maintains the state of the corresponding X window.

Graphical primitives. The LLVO class provides methods for rendering simple graphical objects, text and figures in the views. The coordinate system used for the graphical objects' representative coordinates (as arguments to the methods) is a world coordinate system specified by the user at the moment of (re)initializing a view. That is, the user chooses a coordinate system in which the dimensions are closely related to the performance information. The transformation to the view coordinate system is then in most cases transparent to the user. (Major exceptions are labels written on the margins, whose coordinates must be given in the world coordinate system, while the view coordinate system is sometimes more intuitive. There exists a LLVO method that provides information which enables the use of the view coordinate system.)

Display area. A view consists of an internal area surrounded by *margins*, referred to as *scrollable area*. As a visualization progresses, the mapping from the world

coordinate system to the view coordinate system may change (either implicitly, as declared for quantitative adaptation purposes, or explicitly), at which point only the contents of the scrollable area may be translated or rescaled (zoomed) as a response.

Control methods. Methods such as *scroll*, *resize*, *rescale* and *snapshot* provide explicit control over each view. Combined with the graphical primitives, they allow an HLVO to control explicitly, among other things, what to be drawn and what to be visible at a *point in time*.

Quantitative adaptation. A relation between a view and calls to graphical primitives that draw in the view may be established that causes the view to adapt dynamically by translating or rescaling (zooming) the contents of the scrollable area, thus implicitly controlling what should be visible over an *interval of time*.

Qualitative adaptation. The LLVO class may be portable to multiple graphical platforms that differ at some extent (e.g., different X servers may use different color maps). At run time, it may adapt to the platform capabilities, as well as provide drawing optimization.

An LLVO class implementation may perform book-keeping about graphical objects being drawn and/or be based on vector graphics, in order to facilitate quantitative and/or qualitative adaptation. However, this is not mandatory and an HLVO class implementation can only assume that the underlying LLVO class is memoryless and raster-based.

The quantitative adaptation of a view in this implementation is initialized by specifying directions (from $\{x+, x-, y+, y-\}$) and types of adaptation (rescaling or scrolling) for these directions. On the other hand, one of the parameters of every graphical primitive is the *adaptation flag* that determines whether the view should adapt, e.g., rescale down if necessary, before the graphical object is drawn, in order for the graphical object to be visible. In the case of rescaling, the view may also adapt in the opposite way. For example, if the view had to rescale "down" (zoom out) in response to a peak in a temporal line plot (the contents of the scrollable area is scrolled to the left as the time progresses), it will rescale "up" (zoom in) once the peak has disappeared from the scrollable area. Another parameter for the initialization is the *level of quantitative adaptation.* It can be used to, for example, specify the maximum size of data structures (in this implementation, interconnected red-black trees [28]) used to remember extreme points of graphical objects that have been drawn with the adaptation flag set.

5.1.2 High-level visual object

Similarly as for the LLVO class in general, the implementors have freedom to define a precise framework for developing HLVOs. In this section, the HLVO implementation base is described, and in Section 5.2 an HLVO development framework is presented. The main components of an HLVO class are the four methods shown in Figure 5.1.

Event processing. The performance data passed to an HLVO via calls to the processing method are termed *events* (or data events). Based on the events, this

method (1) updates data structure holding performance information that are referred to as *info structures*, and (2) controls the rendering of this information by updating data structures referred to as *control structures*.

Information rendering. The rendering method may be called, to (read and) *map* a portion of the info structures' contents to the LLVO views, either immediately after processing an event (asynchronous rendering mode) or by a thread that may synchronize the rendering of multiple HLVOs (synchronous rendering mode). This method communicates with the processing method by *both reading and writing* the control structures.

Callback processing. An HLVO may also respond to changes in its run-time environment, as well as to the user's commands. This method may, for example, preprocess callback events coming from the LLVO (e.g., if a view has been automatically resized), the GUI of a VO, etc., and then forward them to the processing method as if they were data events.

(Re)initialization. In on-line performance visualization, it is desirable to be able to partially reinitialize or reconfigure an HLVO without interrupting the target application and/or instrumentation system that supplies performance data. The user may define which info structures should be reinitialized and how, and what should be drawn upon a mouse click on a "reinit" button or a similar event.

In order to allow for rapid prototyping of HLVOs and further PAV research, two frameworks have been developed based on two interpreters/compilers of the Scheme

language [26]. The first Scheme implementation is called GUILE [73] and has a Scheme-to-C compiler called Hobbit [108] associated with it. In this implementation, a generic HLVO class inherits the X library-based LLVO class. Both classes have some methods and data wrapped by Scheme procedures within a tool integration environment for instrumentation and performance visualization, called PG^{RT}-TIE [6, 7]. The GUI of a VO (in addition to LLVO callbacks) is implemented separately, using a GUILE interface to Tk [118]. A CORBA interface has been developed for this framework so that a PAV application may consist of VOs distributed over multiple nodes.

The second Scheme implementation is called Kawa [19], and includes an interpreter and a compiler to Java VM bytecodes. Similarly to the first implementation, a generic HLVO class inherits the Java AWT-based LLVO class. The LLVO class methods are wrapped by Scheme macros. The GUI of a VO is implemented in a way that minimizes necessary modifications of the VOML compiler described in Section 5.2.3: a layer of Scheme macros mimics the GUILE Tk interface on top of the Java Swing library. This framework allows for developing either standalone or WWW-centered VO prototypes.

It is possible in Scheme, as a dynamically-typed language, for the event processing method to receive any type of data structure as an event, which allows for easy integration of different performance data sources. Most importantly, this high-level algorithmic language is suitable for easy definition of complex info structures (e.g., association lists serving as micro-databases) and compact expression of updating and querying them (e.g., with the help of powerful macros) in the event processing and information rendering methods, respectively.

5.1.3 Application of visual objects to a heterogeneous system

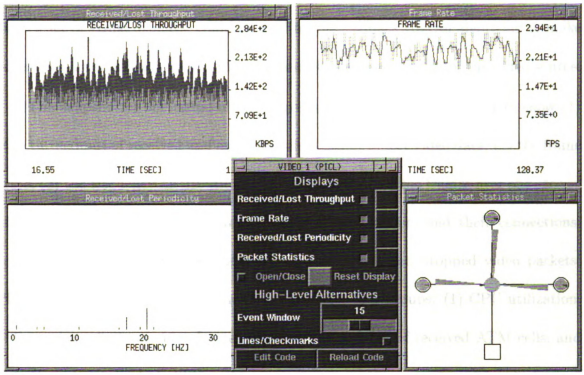

Figure 5.2: On-line performance visualization of the real-time multimedia application

As part of the PG^{RT} environment, two prototype visual objects have been applied to the study of a distributed multimedia real-time application. The target system consisted of a server and a number of heterogeneous receivers of multimedia data streams. The visual objects helped determine the processing demands required to playback different patterns of video frames and to handle different sizes of video frames, as well as the wasted computation due to receiving video frames that cannot be replayed due to time constraints (e.g., a new video frame arrives before an older video frame can be processed). Furthermore, they helped understand the operation of the application: variations in periodic behavior, and specific points in a network

where frame loss occurs, either due to network congestion or individual workstations loading conditions, were displayed. Immense amount of state information, condensed into a set of visual displays, could be used by the feedback control algorithm to make decisions automatically about target bandwidth being requested of the video source.

The snapshot of one visual object is given in Figure 5.2. Its four views show (1) the throughput of received and estimated throughput of lost video data, (2) the frame rate, (3) the frequency distributions of received and lost video frames over one-second intervals, and (4) a spatial, animated view of all receivers and their connections, depicting the relative volumes of received, lost, used, and dropped video packets. The other visual object has 16 views, divided into four groups: (1) CPU utilization, (2) the periodicity of video frames received, the number of received ATM cells, and (4) the number of lost ATM cells. In each group, there are four related views of the corresponding metric: (1) minimum-average-maximum, (2) sample deviation, (3) aggregate, and (4) per-receiver histogram.

5.2 Visual Object Markup Language (VOML)

In this section, a framework is described for semi-automatic design and prototyping of HLVOs. First, a generic architecture for event processing and performance information rendering is defined that is orthogonal to the VO architecture described in Section 5.1. (In this context, where the two architectures coexist, "orthogonal" means that either architecture can be extended without essentially affecting the other, while they are combined to obtain a working HLVO.) Next, salient characteristics of a very

high-level language based on this architecture, called Visual Object Markup Language (VOML), and its compiler are presented. The full VOML document type definition (DTD) is given in Appendix B. The VOML system allows a performance visualization developer to concentrate on application- and visualization-specific semantics and build HLVOs by combining reusable components.

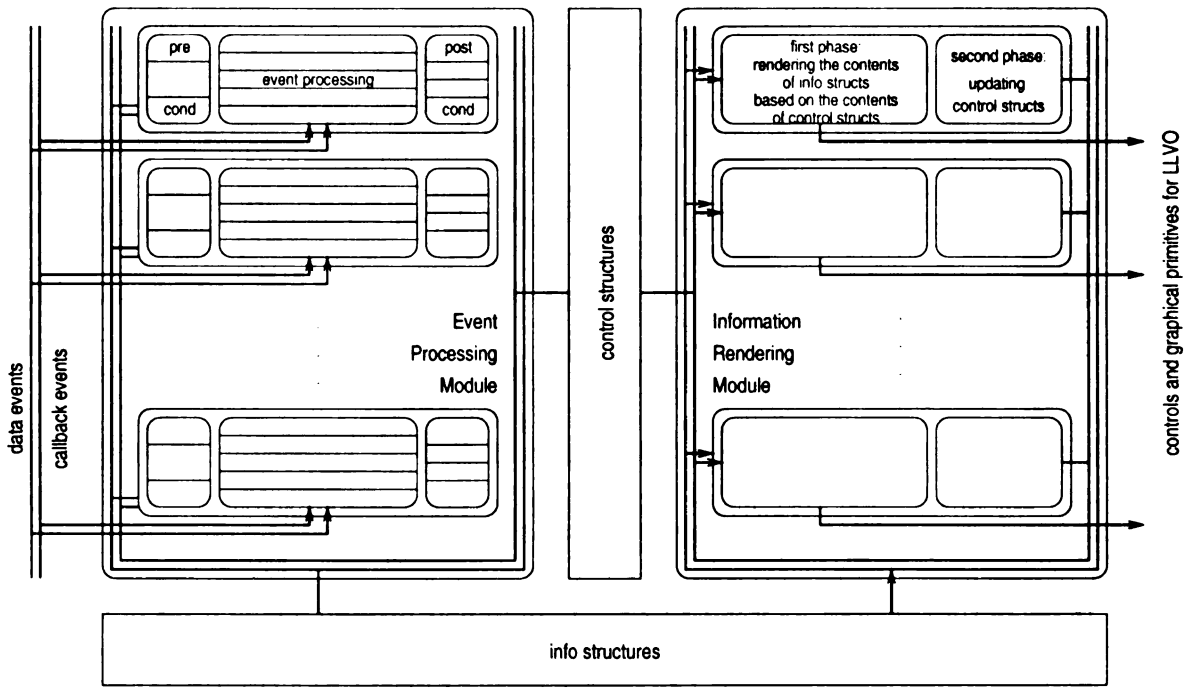

Figure 5.3: Event Processing and Information Rendering Architecture (EPIRA)

5.2.1 Event Processing and Information Rendering Architecture (EPIRA)

There are many possible patterns for development of complex HLVOs. For example, one could extend or modify the VO architecture in Figure 5.1 and build complex

HLVOs in a pure object-oriented style, by inheriting from simpler HLVOs. However, since a goal was to develop a framework that could be applicable to target languages that do not have strong support for object orientation (e.g., Scheme and C), a component-based approach has been taken.

Figure 5.3 shows the *Event Processing and Information Rendering Architecture (EPIRA)*. The architecture specifies the tentative parts of the HLVO architecture, shown in Figure 5.1, and focuses on the data-driven computation aspect. The two modules in the figure correspond to the event processing and information rendering methods. The events arrive (via method calls) from the two "busses" on the left: they carry the performance data and the changes in the run-time environment. Arrows are drawn to denote unidirectional data flows.

The event processing module may contain a number of *event processing (EP) components*. Each EP component in turn may contain a number of *parts* (separated by horizontal lines in the figure), belonging to one of three classes: (1) event-based ones, shown in the middle and executed upon arrival of a specific event, and condition-based ones, which can be executed (2) before or (3) after the event processing, provided that a specific condition tests true. (In VOML, this is referred to as a "condition event.") Since only one event can be received at a time, among all event-based parts (belonging to different EP components) only those for the received event are executed. The conditions corresponding to the condition-based parts are evaluated each time any event is received.

Similarly, the information rendering module contains a number of *information rendering (IR) components*. Each IR component in turn contains two parts, or phases,

in order to avoid potential write-after-read hazards involving the control structures. During the first phase, info and control structures are analyzed and the contents of the info structures are appropriately rendered in multiple views. Once that all first-phase parts of the IR components have been executed, the execution of second-phase parts begins, when the control structures may be safely updated. Since the HLVO assumes that the LLVO has no special rendering support (e.g., a depth buffer), some visualizations may depend on the relative execution order of the first-phase parts.

5.2.2 The VOML language

The SGML [42] has been chosen as the basis for a PAV information infrastructure to be built around the VO and EPIRA architectures. For a start, the VOML is an SGML document type definition (DTD) that encompasses the *structure* of HLVOs based on EPIRA. Some of its higher-level elements and example relations among the elements are given in Figures 5.4a and 5.4b.

As it can be seen from Figure 5.4b, VOML attributes are used both to specify certain characteristics of software components described by the elements and to create relations among them, some of which directly correspond to the connections shown in Figure 5.3. The others are not as "hardwired," and are described using Figures 5.4b and 5.5. Figure 5.5 defines an example IR component that is used in a visual object defined in Figure 5.4b.

Although SGML is a very suitable tool for writing structured specifications, it lacks the means for describing semantics of a specification. On the other hand, Scheme is

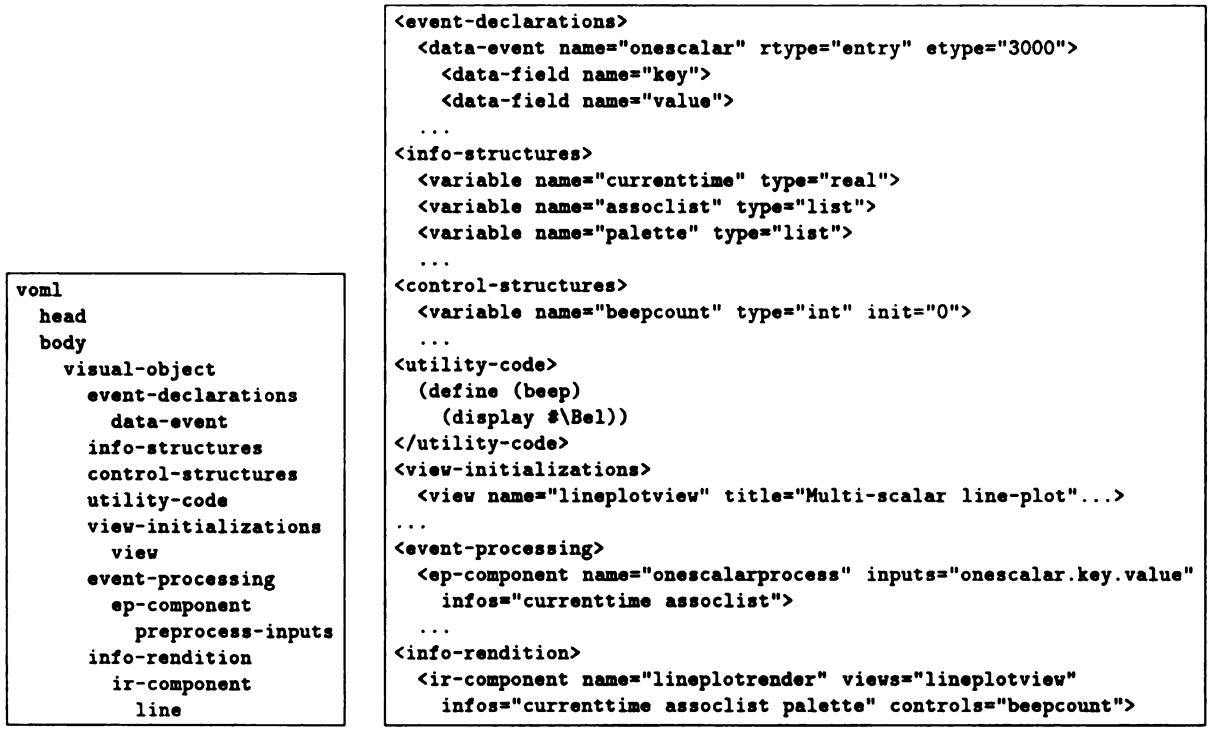

Figure 5.4: A brief description of VOML

a standardized language with simple syntax and clean semantics that is very suitable for describing the semantics of EP and IR components. Hence, a decision was to embed Scheme into VOML markup. Combining markup and a programming language, typically Java in WWW-related markup languages, is not a new idea. However, the integration of VOML and Scheme is tighter, as can be seen from the code example in Figure 5.5. Unlike script-augmented HTML files that are final documents to be "executed," VOML specifications are to be compiled.

Namely, within Scheme code defining the semantics of a component, there may exist references to "formal parameters:" info structures, control structures, events

```
<description>
  This IR component draws a line-plot of multiple scalars over time, in the supplied
  view (^0). Only lines with the last-update time equal to the current time are drawn.
  Once 10 lines have been drawn, a short sound (beep) is generated.

  The info structures consist of the current time ($0, non-negative real number)
  a multi-scalar association list ($1, indexed by non-negative integer keys),
  and a color palette ($2, a list of strings -- color names).
  Each value in the association list is a 4-element vector:
  #(old-time old-value new-time new-value).

  The key of each value in the multi-scalar association list is used to index the
  color. When all colors are exhausted, the line thickness is increased to distinguish
  between different scalars. A counter is used as a control structure (%0) for
  generating sounds.
</description>
  (let ((palette-len (length $2)))
    (alist-for-each
      (lambda (scalar-id scalar)
        (if (= $0 (vector-ref scalar 2))
          (begin
            (set! %0 (+ %0 1))
            (if (= %0 10)
              (begin
                (beep)
                (set! %0 0)))
            <line view="^0" from="(vector-ref scalar 0) (vector-ref scalar 1)"
              to="$0 (vector-ref scalar 3)" thick="(+ (quotient scalar-id palette-len) 1)"
              color="(nth (modulo scalar-id palette-len) $2)" adapt="yes" clip="margin">)))
      $1))
```

Figure 5.5: Code of the IR component used as **lineplotrender** in Figure 5.4b

(in EP components) or views (in IR components). The reference notation is `Tn[/m]`, where

- `T` is `$` for info structures, `%` for control structures, and `^` for events and views;
- `n` is the position of the formal parameter in the corresponding parameter list (e.g., `$0` corresponds to **currenttime** in the last line of Figure 5.4b, because it is the 0-th argument supplied via the **infos** attribute);
- optional `/m` is used for referencing individual fields of an event. (Currently, VOML only supports PICL [122] compliant events, i.e., lists with the first two elements being integers that determine the record and event type.) For example,

an occurrence of `^0/1` within code of EP component **onescalarprocess** in Figure 5.4b would reference field **value** of data event **onescalar**.

The info and control structures are translated into special global variables by the compiler. Effectively, they are "passed" to EP and IR components by reference when listed in the **infos** and **controls** attributes of the enclosing VOML element. In this way, a reusable component may be written, tested, and placed into a library. In an SGML system, such components may be kept as external SGML entities and used in VOML specifications of different HLVOs by simply referencing them by names.

EP components tend to be application-specific, as they process application-specific event records. To make them more reusable, the element **preprocess-inputs** is provided that allows for specifying "glue logic" (as Scheme expressions) for data events specific to a new application. Namely, before an existing EP component is referenced (i.e., used), any fields of the data events it processes may be arbitrarily preprocessed. For example, an EP component that updates info structures for a simple line-plot visualization (e.g., Scheme code just under **<ep-component name="onescalarprocess"...>** in Figure 5.4b, updating info structures to be rendered by the IR component code in Figure 5.5) can be used to visualize the frame rate of a multimedia application. The glue logic in this case could be a function that divides the number of frames received in a time interval by the length of the time interval, whose result would be assigned to the second field (named **value** in the case of the default data event **onescalar**). (Assume that the number of frames is contained in a data event field, and the time interval is kept in an info structure.)

Similarly, different library IR components that are parameterized may be combined in interesting ways over a number of views. Additionally, they may be given attributes to determine their higher-level behavior. (Currently, this behavior is supported in the HLVO implementation by auxiliary Scheme code; it might also be supported by a drawing-optimizing LLVO class.) One such attribute is named **refresh**, which currently can have any combination of values **resize**, **rescale** and **update**. If any of the first two values is used for an IR component, the component will *redraw* its contents if any of the views it draws to gets resized or rescaled. This is useful for raster-based LLVO class implementations, where resizing or rescaling an image is lossy. If **update** is used, the component will *undraw* what it drew last time, before proceeding to render the contents of the info structures again. Certain higher-level behavior, which would by default ignore any control structures (e.g., the effect of **update** does not depend on the actual IR component code), can also be controlled by an **enable** attribute that takes a Scheme expression evaluating to a Boolean value. When combining IR components, the HLVO developer may define their execution order.

5.2.3 The VOML compiler

The VOML compiler is built on top of an SGML transformation library called STIL [99] and consists of the following components.

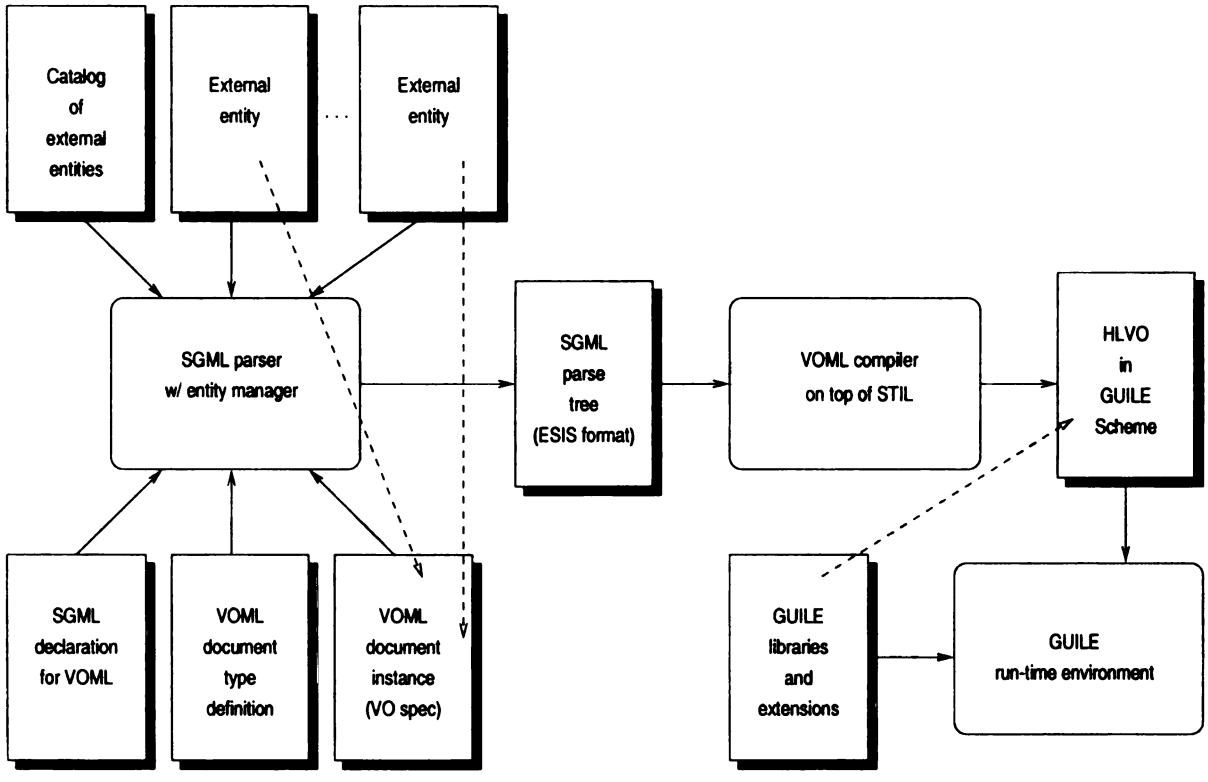

Figure 5.6: VOML compilation and execution process diagram

SGML parser. The `sgmls` parser [25] is used as the front-end that parses an SGML declaration, VOML DTD, and external entities used in a VOML specification of an HLVO.

STIL library. This library is written for the `clisp` [48] implementation of Common Lisp with CLOS. It allows traversing a parse tree created by the SGML parser, and defining "hooks" (semantic actions) that are called during the traversal.

VOML validating parser. One part of this component consists of the hooks called by the STIL library. The other part consists of CLOS objects that contain code and other information relevant to EPIRA components of the HLVO specification

being compiled. The hooks process VOML elements (including the contents in Scheme), their relations and attributes, and build the CLOS objects.

VOML code generator. This component "tangles" the plain, application- and visualization-specific Scheme code from a VOML specification with code that it generates for integration with the run-time environment. The latter includes a graphical user interface for accessing and modifying selected info and control structures, managing views, registering VOs with routines that supply data events, etc.

Figure 5.6 shows the compilation and execution process of a VOML specification in the GUILE-based environment. Processes are shown as oval rectangles, while input, intermediate and output files are shown as rectangles. Solid lines denote the process of file inclusion, while dashed lines denote references in VOML and Scheme files.

While an SGML parser uses an *entity manager* to find components of a document which are referenced as external entities—such as library EP and IR components—within its *virtual storage system*, the SGML standard itself does not specify how to implement one. A WWW-enabled entity manager would further enlarge the PAV information infrastructure and facilitate automated monitoring and PAV of globally distributed applications. Figure 5.7 shows an example of how component definitions could be fetched off a WWW site by the entity manager, to be included for compilation. In the example, the vendor of an imaginary software product, whose performance is to be visualized, keeps the latest implementations of an EP and IR

component for the product, ready to be used in a HLVO specification. (It is assumed that the information about the components' interfaces is available.)

```
<!DOCTYPE VOML PUBLIC "-//MSU-PGRT//DTD VOML 1.0//EN"
  [
     <!ENTITY SoftwareXYZep
       SYSTEM "http://vendor.com/voml/XYZep.voml">
     <!ENTITY SoftwareXYZir
       SYSTEM "http://vendor.com/voml/XYZir.voml">
  ]
>
<voml>
...
      <event-processing>
        <ep-component name="XYZep"
                      inputs="mydata1.f1.f2 ..." ...>
          &SoftwareXYZep;
        </ep-component>
...
      <info-rendition>
        <ir-component name="XYZir"
                      views="myview1 ..." ...>
          &SoftwareXYZir;
        </ir-component>
...
```

Figure 5.7: Sketch of a VOML specification that uses remote component definitions

5.3 The VOML Specification of a Simple Visual Object

In this section, the main parts of the VOML specification of a simple VO with a view similar to the last one of the VO shown in Figure 5.2 are presented and commented. The VO receives performance data events from a distributed application, generated whenever a node is (1) added or (2) removed, and periodically to carry profile data from each node. The event declaration section is shown in Figure 5.8. The record and event types of the first two events are taken from the PICL specification [122], while the profile event belongs to an extension of PICL. When some field are skipped (i.e.,

ignored), the **index** attributed is used to specify the position of the next declared field.

```
<event-declarations>
  <data-event name="addnode"
              rtype="pg-entry" etype="-901">
    <data-field name="ts" type="int">
    <data-field name="node-id" type="int">
  </data-event>
  <data-event name="rmnode"
              rtype="pg-exit" etype="-901">
    <data-field name="ts" type="real">
    <data-field name="node-id" type="int">
  </data-event>
  <data-event name="node-prf"
              rtype="entry" etype="3141">
    <data-field name="ts" type="real">
    <data-field name="node-id" index="3" type="int">
    <data-field name="node-type" index="5" type="int">
    <data-field name="rkbps" type="real">
    <data-field name="tb" type="real">
    <data-field name="used" type="real">
    <data-field name="fps" type="real">
    <data-field name="packets" type="int">
    <data-field name="pack-used" type="int">
  </data-event>
</event-declarations>
```

Figure 5.8: Event declarations

The info and control structure specifications are shown in Figure 5.9. The info variable **numofnodes** (although redundant) keeps the current number of communicating nodes; **nodes** is an association list that keeps the previous and current profile of each node; **nodeno** keeps the (non-negative) node id from the latest profile event. The control variable **nodechange** indicates whether the number of nodes has changed (meaning that the view has to be updated, as will be seen later).

The next is the utility code section, which is omitted for brevity. It contains function **getfontname** that returns a font name from a list of available fonts, given some hints. Besides, functions **id-get**, **id-put** and **id-rem**, which are used to manipulate

```
<info-structures>
  <variable name="numofnodes" type="int">
  <variable name="nodes" type="list">
  <variable name="nodeno" type="int" init="-1">
</info-structures>

<control-structures>
  <variable name="nodechange" type="boolean">
</control-structures>
```

Figure 5.9: Info and control structures

the **nodes** association list, may be defined in this section (in this case, they are defined and exported from another module, available in the run-time environment).

The view initialization section is shown in Figure 5.10. The (only) **BU-View** view is 700 by 700 pixels large, through which a rectangle in the world coordinate system from $(-10, -10)$ to $(110, 120)$ is visible. In this example, the view neither scrolls nor zooms. The control variable **nodechange** is set to true only to trigger the drawing of the switch in the beginning.

```
<view-initializations>
  <view name="BU-View" window="700 700"
    world="-10 110 -10 120" controls="nodechange">
    <description>Switch, nodes and bandwidth
      utilizations</description>
    (set! %0 #t)
  </view>
</view-initializations>
```

Figure 5.10: View initialization

There is one EP component for each event, although one could implement, for example, only one for all the three events (depending on the desired granularity when creating a component library). They are shown in Figure 5.11, as updating the info and control structures according to the event declarations. Fields of an event are listed using a notation in which the event name is followed by some of its fields'

```
<event-processing>
  <ep-component name="rmnode" inputs="rmnode.ts.node-id"
    infos="numofnodes nodes nodeno" controls="nodechange">
    <description>Remove a node</description>
    <input name="^0">
      (set! $1 (id-rem $1 ^0/1))
      (set! $0 (- $0 1))
      (set! $2 -1)
      (set! %0 #t)
    </input>
  </ep-component>
  <ep-component name="addnode" inputs="addnode.ts.node-id"
    infos="numofnodes nodes nodeno" controls="nodechange">
    <description>Add a node, reset the infos</description>
    <input name="^0">
     (set! $1 (id-put $1 ^0/1
           (cons (vector ^0/0 ^0/1 0 0 0 0 0 0 0 0)
                 (vector ^0/0 ^0/1 0 0 0 0 0 0 0 0))))
     (set! $0 (+ $0 1))
     (set! $2 -1)
     (set! %0 #t)
    </input>
  </ep-component>
  <ep-component name="nodeprofile"
    inputs="node-prf.ts.node-id.node-type.rkbps.tb.used.fps.packets.pack-used"
    infos="numofnodes nodes nodeno">
    <description>Update a node's infos</description>
    <input name="^0">
      (let ((old-info (cdr (id-get $1 ^0/1)))
            (new-info (vector ^0/0 ^0/1 ^0/2 ^0/3 ^0/4
                              ^0/5 ^0/6 ^0/7 ^0/8)))
        (set! $1 (id-put $1 ^0/1
                         (cons old-info new-info))))
      (set! $2 ^0/1)
    </input>
  </ep-component>
</event-processing>
```

Figure 5.11: Event processing components

names, delimited by periods. In the **nodeprofile** EP component, all the event fields declared above are used. It is not necessary to use them all and in the same order as declared; the **^m/n** notation uses the order(s) given in the **inputs** attribute. It can be seen that the field **node-id** is used as the key, and the value field in the **nodes** association list is a pair of vectors keeping the previous and current profile of a node. In this example, only the current profile will be used, but in a more complex VO both the previous and current one may be needed.

```
<ir-component name="nodes-ir" views="BU-View"
  infos="numofnodes" controls="nodechange"
  refresh="update resize" buffer="yes" enable="%0">
  <description>Switch, nodes, connections</description>
  (let* ((viewinfo <view-info view="^0">)
         (width    (list-ref viewinfo 5))
         (height   (list-ref viewinfo 6))
         (size     (inexact->exact
                     (max (/ width 40) (/ height 40))))
         (font     (getfontname "fonttable"
                                "courier" size "bold")))
    <text view="^0" coords="50 107" halign="center"
      font="font" fcolor='"black"'
      content='"Bandwidth Utilization"'>
    <figure view="^0" filename='"bggif/switch.gif"'
      orig-origin="0 0" orig-extents="0 0"
      world-origin="45 45" world-extents="10 10">
    (let* ((nodenum (- $0 1))
           (step    (/ (if (gt $0 0)
                           (/ 6.28 nodenum)
                           0)))
      (if (gt nodenum 0)
        (let loop ((num nodenum))
          (let* ((angle  (* num step))
                 (sine   (sin angle))
                 (cosine (cos angle)))
            <figure view="^0"
              filename='"bggif/node.gif"'
              orig-origin="0 0" orig-extents="0 0"
              world-origin="(+ 45 (* 45 sine))
                            (+ 45 (* 45 cosine))"
              world-extents="10 10">
            <line view="^0" from="(+ 50 (* 6 sine))
                                  (+ 50 (* 6 cosine))"
                            to="(+ 50 (* 39 sine))
                                (+ 50 (* 39 cosine))"
              color='"red"' thick="12">
            (if (gt num 0)
              (loop (- num 1))))))))
  <end-with>(set! %0 #f)</end-with>
</ir-component>
```

Figure 5.12: Template IR component

Finally, the information rendering section consists of two IR components. The **nodes-ir** IR component, which is shown in Figure 5.12 and will be executed first, writes text and draws the switch and as many "PG^{RT} globes" around it as there are active nodes, connected with the switch via thick red lines. (In the prototype implementation of the VOML compiler, the execution order of the IR components is opposite of the order they appear in a specification.) The **enable** attribute specifies that this IR component should be executed whenever the number of nodes has

```
<ir-component name="bu-ir" views="BU-View"
  infos="numofnodes nodes nodeno" refresh="resize">
  <description> Bandwidth utilization </description>
  (if (gt $2 -1)
    (let* ((angle    (* $2 (/ 6.28 $0)))
           (sine     (sin angle))
           (cosine   (cos angle))
           (new-info (cdr (id-get $1 $2)))
           (node-t   (vector-ref new-info 2))
           (newkbps  (vector-ref new-info 3))
           (newtotal (vector-ref new-info 4))
           (newused  (vector-ref new-info 5))
           (sent     (/ newkbps newtotal))
           (mag      (/ newused newkbps)))
      <line view="^0" from="(+ 50 (* 6 sine))
                            (+ 50 (* 6 cosine))"
                     to="(+ 50 (* 39 sine))
                         (+ 50 (* 39 cosine))"
        color='"red"' thick="12">
      (if (= node-t 2)
        <line view="^0" from="(+ 50 (* 6 sine))
                              (+ 50 (* 6 cosine))"
                       to="(+ 50 (* (+ 6 mag) sine))
                           (+ 50 (* (+ 6 mag) cosine))"
          color='"blue"' thick="10">
        <line view="^0" from="(+ 50 (* 6 sine))
                              (+ 50 (* 6 cosine))"
                       to="(+ 50 (* (+ 6 mag) sine))
                           (+ 50 (* (+ 6 mag) cosine))"
          color='"green"' thick="10">
    (set! $2 -1)))
</ir-component>
```

Figure 5.13: Active IR component

changed. The **refresh** attribute adds that everything the IR component drew last time should be redrawn when the view **BU-View** is resized. It also specifies that everything the IR component drew last time has to be undrawn before something new is drawn (whenever the IR component is enabled). The **buffer** attribute is used to make the IR component draw in-memory only until it is done, and then flush the contents of the memory to the screen. This is useful to make the rendering smoother and faster when there are many graphical objects to be drawn. This IR component resets the **nodechange** control variable in the second phase, so that other IR components may be added safely that depend on the value of this variable. In Scheme, the HLVO prototyping language embedded in VOML, this second phase is implemented using

the `delay` and `force` primitives. This allows for mixing the first- and second-phase IR code as if there were only one phase, and avoiding the write-after-read hazard at a high level. (Care must be taken when there are circular/mutual dependencies among control structure updates.)

The other IR component is executed each time an event is received, and it draws a blue thick line on top of a red thick line (drawn in the same place as the thick red line drawn by the previous IR component, so that it can effectively be undrawn), showing the relative bandwidth used by a node (i.e., a portion of the received bandwidth; the line is drawn after a profile event was received that resulted in setting the **nodeno** info variable to a non-negative value). If the node is the server, it draws a green thick line instead of a blue one, showing the relative bandwidth consumed by the server. The IR component code is shown in Figure 5.13. The refresh attribute treats the resizing of the view same as above.

A snapshot of the view is given in Figure 5.14. Note that not all of the VOML features have been shown in this example.

5.4 Summary

A novel PAV technology intended to satisfy growing needs of researchers and users of heterogeneous parallel and distributed systems has been presented. Salient characteristics of the technology include support for rapid prototyping and automated design of PAV tools, object orientation, distributability, portability, code reuse and

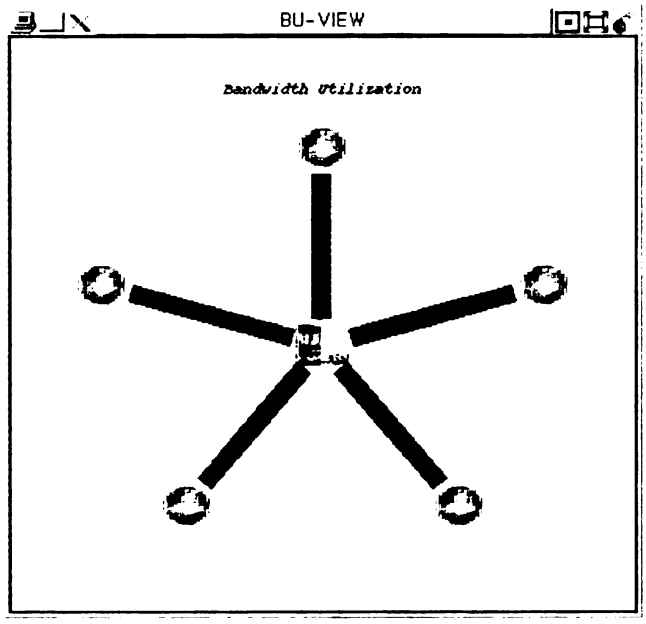

Figure 5.14: A snapshot of the view

flexibility. An example of high-level visual object specification in VOML has been presented that shows some of the features of the language.

Chapter 6

An Integrated Approach to Real-Time System Design and On-Line Performance Visualization with Steering

This chapter presents an approach to integration of the two technologies presented in chapters 3 and 5, which use BRISK, presented in Chapter 4, for instrumenting and steering a target complex real-time system. First, in Section 6.1, an example, real-world target distributed real-time system is described. Its design and engineering using the CLP approach/technology is described in Section 6.2. Next, technical details about the integration of the two technologies are explained in Section 6.3. Finally,

the operation of the new, integrated technology is shown on an example scenario in Section 6.4.

6.1 Target Real-Time System

The target real-time system used to demonstrate the approach is a modification of a real-time face-tracking system [11]. The original system runs on a single workstation equipped with an inexpensive "eye" video camera. It analyzes video frames and finds the workstation user's face and, once the face has been found, the eyes (as well as other features, such as the eyebrows and nose tip). By tracking the head movements and gaze direction in real time, the system can be used as a basis for handless application control, graphics cursor driver, etc.

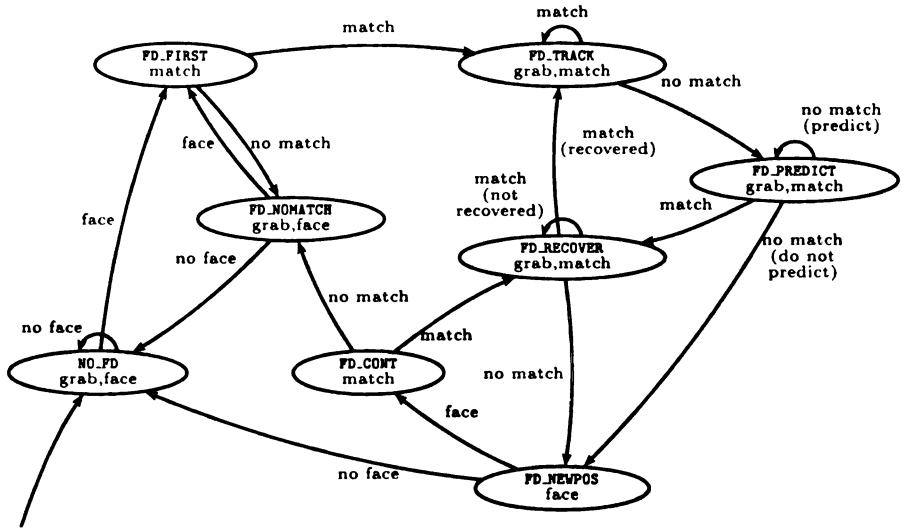

Figure 6.1: Original face-tracking system's state diagram

The state diagram of the original face-tracking system is shown in 6.1. The most relevant facts about it are summarized below.

- The task of finding the user's eyes ("match") is relatively lightweight, and is the main part of the system's "inner loop." It assumes that the user's face has been located, and is invoked as frequently as possible. It usually suffices to invoke this task at a frequency between 15 and 20 Hz in order to smoothly track the eyes. The eyes can be located even if the user's face moves a little, relatively to the assumed face location. In cases of short temporary problems with finding the eyes, the system uses a computationally simple Kalman filter to extrapolate the eyes coordinates ("predict").

- The most time-consuming task is that of finding the user's face ("face"). As the main part of the system's "outer loop," it is invoked after the user's face has moved significantly enough for that the eye-finding task to fail to locate the eyes. This usually happens after 5 to 20 successful eye findings, possibly helped by a few predictions.

Real-time characterization of the original system. It is a soft real-time system in which, besides a required minimum average tracking frequency that depends on the workstation user's head movements, important roles have the extrapolation and face-recovery mechanisms. The latter is the most critical task because it interrupts the tracking for a user-noticeable time interval. However, it is too expensive on a workstation to look for the current location of the user's face more often than only when necessary.

While the original system achieves required tracking frequencies on an average workstation, it does that at the cost of fully utilizing the CPU. Its modification p-

resented here is a transformation into a client-server system. The motivation comes from the facts that making the target real-time system distributed, (1) the approach to design and engineering of complex, distributed real-time systems from this dissertation can be applied on a non-trivial case, and (2) the CPUs of several workstations in a fast local-area network can be drastically off-loaded.

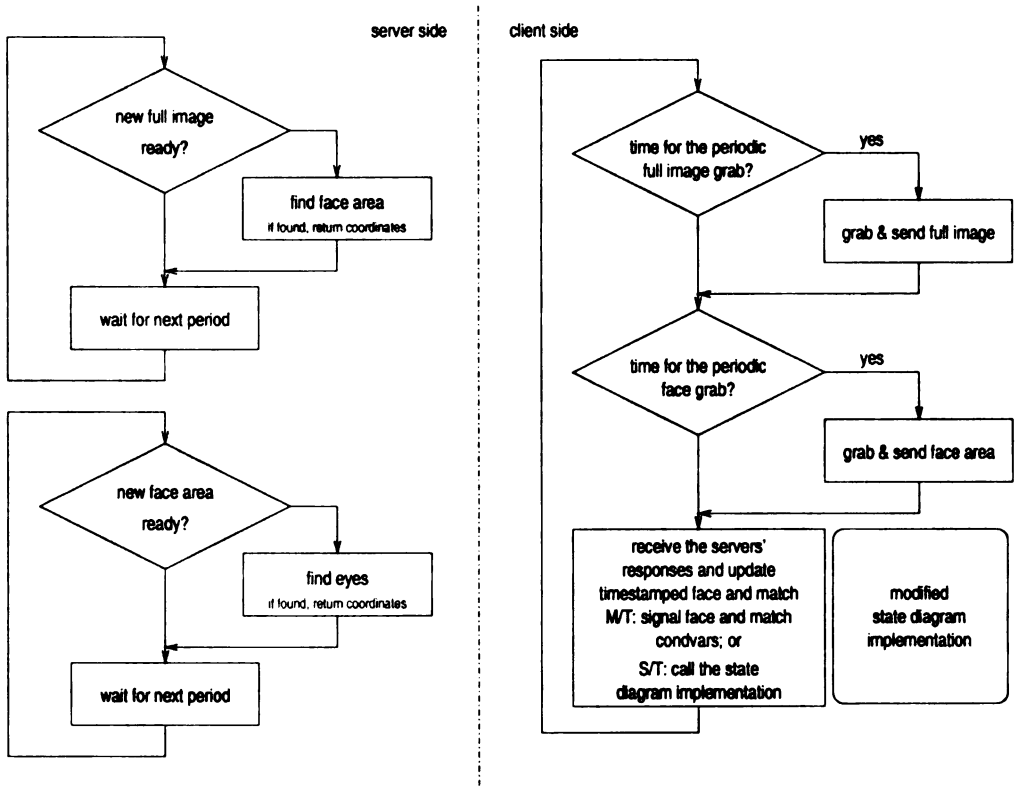

Figure 6.2: Distributed face-tracking system

The original state diagram has been modified, and the tasks of face and eye finding have been made largely decoupled periodic real-time tasks executing on the same or two different Real-Time Linux (RTL) [12, 78] server PCs. The new system is shown in Figure 6.2. A main modification of the state diagram consists of replacing all "grab, face/match" actions by returns (in the single-threaded case, where the

state diagram is a function) or waits on the corresponding signal variable (in the multi-threaded case, where the state diagram is a separate thread). The invocation frequencies of the real-time tasks are high enough to achieve a satisfiable face-tracking quality. At the same frequencies, they receive full and face-only raw images from the client workstations, and return the coordinates of the face and eyes, respectively. The client receives the coordinates, forwards new face coordinates to the eye-finding task, extrapolates eyes coordinates if necessary, and possibly passes them to an application.

Real-time characterization of the distributed system. It is a soft real-time system centered around a hard real-time subsystem running on the RTL server PCs. Behavior based on (fixed over an interval of time) periods had to replace the reactive one, because the delay and jitter introduced by two transfers over the network (to immediately send the image data and receive the coordinates) would stop the tracking for a moment and make it less smooth. The hard real-time subsystem acts as a clock of the whole system. The clients periodically send the image data into a pipeline and receive the coordinates with more delay but less jitter out of it; they can estimate the jitter and take advantage of that. Except for sudden head movements, the periodic face-finding task is less critical, because the face coordinates are kept more up-to-date. Even with a noticeable delay, the tracking can be useful and considered real-time if the delay is almost constant.

Besides these main design choices, there are implementation details to compensate partially the lack of real-time support in legacy workstations and networking hardware. The chosen target system should be viewed mainly as an example. There may be other real-time systems, similar in structure to this one, for which the motivation

for a similar transformation may be stronger. Section 6.2 describes how the target system has been engineered using the compiler-based approach from this dissertation.

6.2 RTSML Specification

In the transformed target real-time system, described in the previous section, up to eight pairs of CPU-demanding face and eye finding tasks execute on three RTL server PCs. They have been implemented as real-time tasks, which are scheduled using a rate-monotonic scheduler. The Linux kernel is also scheduled by RTL as a background task, which is allocated the CPU when no real-time tasks are ready. Figure 6.3 sketches the description of RTL-specific details (the arrows represent data and control flow).

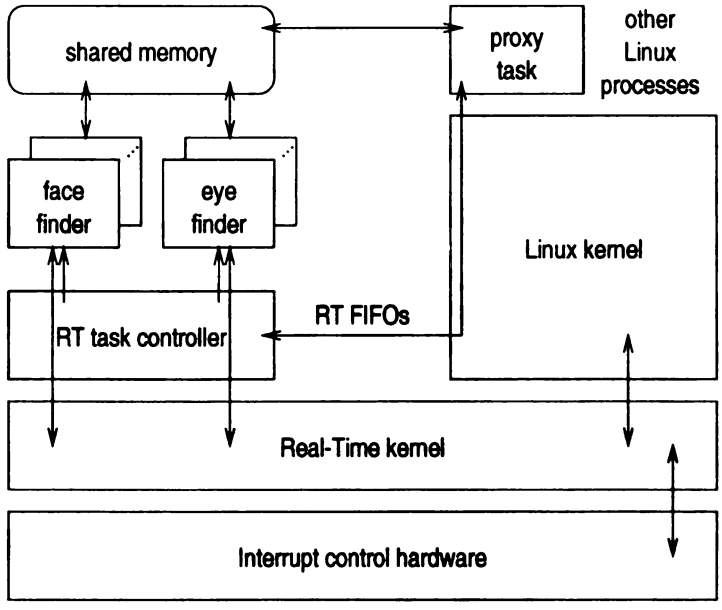

Figure 6.3: RTL-specific details of the target system

Due to the current lack of networking (TCP/IP) support in RTL, the inputs of the real-time tasks, messages with images from the corresponding clients' cameras, are received by a Linux network interface card (NIC) driver (see [95] for more information) and read by a proxy Linux process. The proxy process stores them to a (unmapped and hidden from the Linux kernel) shared memory area accessible to the real-time tasks. In order for the NIC driver not to lose packets of the incoming messages, causing their retransmission, it must be allowed to process them very frequently (1–2 kHz, depending on the NIC hardware). Since the driver cannot be isolated from the rest of Linux (and treated as a real-time task), the whole Linux kernel must be allocated the CPU very frequently and allowed enough time in each turn for the driver's interrupt handling routine, for which the RTL leaves NIC interrupts pending, can copy the incoming packets from the NIC. This required a modification of the default rate-monotonic scheduler, such that it (1) treats Linux similarly to a periodic real-time task, and also (2) allocates the CPU to it when no real-time tasks are ready. Unlike the real-time tasks, which return the control to the scheduler when they finish, the periodic invocations of Linux need to be stopped by the RTL's timer handler after their allowed time has expired.

Furthermore, a real-time analysis of the traffic coming into a RTL server from a fast Ethernet-based LAN is very limited. For example, it is not possible to guarantee that before each instance of a real-time task is allocated the CPU, a fresh input will be available in the shared memory. However, at least a soft real-time, average-case analysis can be beneficial that is applied on the TCP/IP packet assembly in the Linux kernel and storing of messages to the shared memory by the proxy task (the

proxy task receives the messages via stream sockets). Even though the Linux kernel is allocated the CPU very frequently, its allowed time in each turn is only supposed to be long enough to handle outstanding packets in the NIC buffers. Part of this time may actually be given by the Linux kernel to the TCP/IP subsystem and proxy task, but in a conservative analysis it is assumed negligible. Within an interval of time, the time needed by the proxy task to prepare the inputs for the real-time tasks is (conservatively) approximated as linearly proportional to the sum of the lengths of all the messages expected to arrive within that interval. A similar approximation can be done for the task of TCP/IP packet assembly, where small but possible bursts/jitter should be taken into account. Altogether, for this purpose a constant K_{cp} [ms/KB] can be defined whose value is a measured normalized input-preparation time on a given RTL server PC.

To reconcile the above RTL-specific requirements with the exact rate-monotonic schedulability analysis, a theorem due to Lehoczky, Sha and Ding [113] is presented for a start (T_i, C_i and D_i are period, time demand and deadline of task τ_i, respectively):

Theorem. Let a periodic task set $\tau_1, \tau_2, \ldots, \tau_n$ be given in priority order and scheduled by a fixed priority scheduling algorithm using those priorities. If $D_i \leq T_i$, then τ_i will meet all its deadlines under all task phasings if and only if

$$\min_{0 \leq t \leq D_i} \sum_{j=1}^{i} \frac{C_j}{t} \lceil \frac{t}{T_j} \rceil \leq 1. \tag{6.1}$$

The entire task set is schedulable under the worst case phasing if and only if

$$\max_{1 \le i \le n} \min_{0 \le t \le D_i} \sum_{j=1}^{i} \frac{C_j}{t} \lceil \frac{t}{T_j} \rceil \le 1. \tag{6.2}$$

Note: To find the minimum, one needs only to consider some of the values of t which are multiples of any of $T_1, T_2, \ldots, T_{i-1}$ and D_i. □

Assume that under a task-to-processor allocation, the tasks $\tau_1, \tau_2, \ldots, \tau_n$ are some face and eye finding tasks on a RTL server PC. Their priorities are, under the rate-monotonic scheduling, fixed based on their periods. First, the frequent allocation of the CPU to the Linux kernel may be added to the above equations as task τ_0, the highest-priority one. However, a simplification is possible because T_0 is two to three orders of magnitude smaller than the periods of the other tasks. Namely, in the summand in Equation 6.2, when t takes on the values of multiples of $T_j, j = 0, 1, \ldots, i-1$, the relative error of removing the ceiling function in the summand for $j = 0$ becomes negligible, and the summand approaches a constant utilization fraction C_0/T_0. It can be subtracted from both sides of the inequality for all $i = 1, 2, \ldots, n$. Furthermore, assuming that time demands $C_i, i = 1, 2, \ldots, n$, are inversely proportional to the CPU capacity (unlike C_0, which is the fixed allowed time), the effect of introducing the frequent CPU allocations to the Linux kernel is approximately equivalent to slowing down the CPU $1/(1 - C_0/T_0)$ times (the error is on the conservative side, because the time demands $C_i, i = 1, 2, \ldots, n$, on a real PC with a memory hierarchy will actually be increased by a smaller factor).

Second, consider "promoting" the Linux background task into a lowest-priority real-time task τ_{n+1} with $T_{n+1} = D_{n+1} > T_n$. The time demand within an interval of

time $t_{cp} \leq T_{n+1}$, for preparing the inputs from messages expected to arrive within the interval, under an assumption that t_{cp} is large enough for an average-case analysis, is given by the following formula:

$$C_{n+1}(t_{cp}) = t_{cp}\mathrm{K}_{cp}\sum_{i=1}^{n}\frac{L_i}{T_i}, \tag{6.3}$$

where L_i is the length of τ_i's input. Equation 6.1 has been chosen as the feasibility test for task τ_{n+1}, with $C_{n+1} = C_{n+1}(T_{n+1})$ and T_{n+1} slightly larger than the largest of the other periods. A stronger test could, for example, have multiple deadlines for partial preparations (so that each instance of a real-time task gets a fresh input before it is allocated the CPU):

$$\max_{1\leq i\leq n}\frac{C_{n+1}(T_i)}{T_i} + \sum_{j=1}^{n}\frac{C_j}{T_i}\lceil\frac{T_i}{T_j}\rceil \leq 1. \tag{6.4}$$

However, such a stronger and more complex test approaches a worst-case analysis as t_{cp} becomes small for the average-case analysis. Since the incoming network traffic is non-deterministic, a worst-case analysis is not worth pursuing. On the other hand, allowing t_{cp} to become too large would make the chosen feasibility test degenerate into a simple CPU utilization check, which is unnecessarily weak. Finally, the chosen feasibility test requires only a minimal extension to the original rate-monotonic schedulability analysis.

The soft real-time analysis of the network traffic is simple. The whole LAN is viewed as a single communication channel. While the message latency cannot be bounded in a non-real-time LAN, one can hope that it will be relatively low if the

LAN is not very loaded. By observation, e.g., looking at local FTP throughput rates, an upper bound for the allowed bandwidth B can be set. Then, the following formula is used to check if the messages will likely arrive from all the clients to all the servers with low latency:

$$B \geq \sum_{i=1}^{n} \frac{L_i}{T_i}. \tag{6.5}$$

The client programs execute on various non-real-time operating systems, one client per computer. They are now computationally light, with the average CPU time needed per video frame depending also on the application built on top of the face-tracking task. In addition, the CPU time needed by the OS for copying and sending messages to the server(s) is modeled similarly as in the case of the Linux background task on an RTL server PC. The total CPU utilization by the client can be limited using the following formula:

$$S_i \geq \frac{C_{norm,cl,i}}{T_{cl,i}} + \frac{C_{norm,cp}}{1000} \tag{6.6}$$

where S_i is the allowed dedicated CPU capacity of the computer on which the client i executes (the CPU capacity in idealized MIPS is used instead of utilization, because the RTSML compiler modules expect it as one of the parameters). $C_{norm,cl,i}$ is the normalized (at 1 MIPS) average CPU time demand of the client within an interval of time equal to the period of the corresponding, more frequent, eye-finding task, $T_{cl,i}$ (conservatively estimated). $C_{norm,cp}$ [ms] is the normalized message copying demand time within a unit of time (1000 ms), obtained using a formula similar to Equation 6.3.

Here ends the modeling of the distributed real-time face-tracking system. Two RTSML compiler modules have been used to compile parts of the system's specification:

1. default module `common`, for simple periodic tasks and messages, non-real-time channels and processors with utilization checks only (using generalizations of Equation 6.5 and Equation 6.6), and static routing. The client tasks, which are not trivially periodic, have been approximated by periodic ones; this explains the form of Equation 6.6.

2. module `rtlrms`, for RTL-augmented processors with rate-monotonic scheduling (using a generalization of Equation 6.2). Both the `rtlrms` and `common` modules internally create for each processor a `common` task to approximately model the CPU time demand for unaccounted message processing on that processor.

The above-mentioned generalizations assume all possible task-to-processor and message-to-route allocations, variable CPU capacities, message multicasting, etc. The `common` and `rtsml` modules generate CLP code in which satisfactory resource allocations are to be found by applying possibly multiple problem-solving approaches. In general, other modules may search for satisfactory values of some real-time parameters, such as periods. In the context of CLP and problem solving, one distinguishes between input *parameters* (e.g., tasks' periods) and output *solutions* (e.g., task-to-processor allocations). Figure 6.4 shows excerpts from the RTSML specification of the target system.

```
<!DOCTYPE RTSML PUBLIC "-//MSU-PGRT//DTD RTSML 2.0//EN">
<rtsml>
  <head>
    <title>Distributed Face Tracking Application</title>
  </head>
  <body>
    <system id="ft" type="common">
      <processor-group id="pg0" type="common">
        <processor id="p0" type="rtlrms (mips (initial 450) (domain (450)))
                                        (norm-copy-time 5)"></processor>
        <processor id="p1" type="rtlrms (mips (initial 450) (domain (450)))
                                        (norm-copy-time 5)"></processor>
        <processor id="p2" type="rtlrms (mips (initial 200) (domain (200)))
                                        (norm-copy-time 5)"></processor>
      </processor-group>
      <channel-group id="cg0" type="common">
        <channel id="c0" type="common (mbps (initial 4)
                                            (domain ((range 2 5))))"></channel>
      </channel-group>
      <processor-group id="pg1" type="common">
        <processor id="ws0" type="common (mips (initial 100) (domain (50 100)))
                                        (norm-copy-time 6)"></processor>
        ...
        <processor id="ws7" type="common (mips (initial 100) (domain (50 100)))
                                        (norm-copy-time 6)"></processor>
      </processor-group>
      <task-group id="servers" type="common">
        <task id="stf0" type="common
              (period (initial 590) (domain ((range 500 1000))))
              (deadline (initial 590) (domain ((range 100 1000))))
              (time@1mips (initial 20250) (domain (0 (range 18000 22500))))"
              processors="pg0"></task>
        <task id="ste0" type="common
              (period (initial 59) (domain ((range 50 100))))
              (deadline (initial 59) (domain ((range 50 100))))
              (time@1mips (initial 4050) (domain (0 (range 3600 4500))))"
              processors="pg0"></task>
        ... <!-- through stf7 and ste7 -->
      </task-group>
      <task-group id="clients" type="common">
        <task id="ct0" type="common
              (period (inherit-from ste0))
              (deadline (inherit-from ste0))
              (time@1mips (initial 500) (domain (0 500)))"
              processors="ws0"></task>
        ... <!-- through ct7 -->
      </task-group>
      <message-group id="mg0" type="common">
        <message id="mf0" type="common
                 (period (inherit-from stf0))
                 (deadline (inherit-from stf0))
                 (length (initial 225) (domain (0 225)))"
                 source="ct0" destinations="stf0"></message>
        <message id="me0" type="common
                 (period (inherit-from ste0))
                 (deadline (inherit-from ste0))
                 (length (initial 7) (domain (0 (range 4 14))))"
                 source="ct0" destinations="ste0"></message>
        ... <!-- through mf7 and me7 -->
      </message-group>
      <routing-table id="rt0" type="common">
        <route id="r00" type="common" source="ws0" destination="p0"
                channels="c0">
        ...
      </routing-table>
    </system>
  </body>
</rtsml>
```

Figure 6.4: RTSML specification excerpts

In line 7 of Figure 6.4, `common` means that there is no special relationship between the subsystems of the system with identifier `ft`. Similar holds for the subsystems themselves (processor, channel, task and message groups), and routing. The processors in group `pg0`, whose descriptions start in line 8, are the ones running RTL with the modified rate-monotonic scheduler, and the real-time model that they obey is identified by the name of the corresponding compiler module, `rtlrms`. After this identifier, follows a list of model parameters (given using S-expressions) of the processor being described: its capacity in estimated idealized MIPS (initial value and the domain, which may contain single integer values and ranges or integers) and the normalized message-copying demand time (K_{cp}) at the unit CPU capacity (1 MIPS). Parameters which have domains of possible values can be changed at run time (in the case of the MIPS parameter, the changing means increasing/decreasing the CPU utilization share allocated to the tasks of the target system). In line 17, channel `c0` models the 100 Mbps LAN that we have used for the clients and servers: it can be loaded with 2–5 MBps, depending on other users' usage. Eight client workstations are described in lines 20–26 as `common`, i.e., having a non-real-time OS. Eight pairs of face- and eye-finding real-time tasks are described in lines 27–39. The ranges of their periods correspond to the ranges of usual frequencies of the face- and eye-finding tasks in the original system. There are no special deadline constraints, so that they can be equal to the periods. The `time@1mips` parameter is the normalized CPU time demand in milliseconds, and its value is the *maximum* measured time demand in typical face-tracking experiments, multiplied by the CPU's specified capacity in idealized MIPS. Depending on the actual workstation user's head/eye movements,

this maximum value can vary within a range. The value of 0 means that the task is actually not running at all. The `processors` attribute states that the real-time tasks can only be allocated the processors in the group `pg0`. Eight client tasks are described in lines 40–47. Their periods and deadlines are inherited from the corresponding eye-finding server tasks for the purpose of modeling, but are also enforced so by a proxy task at run time, as will be described in Section 6.4. The `inherit-from` specifier is a convention provided by the module `common` for use with tasks and messages. In lines 48–60, the messages carrying full and face-only raw images are described. The length of a full 24-bit video frame needed for the face finding is 225 kB; the maximum length of a message carrying a typical 8-bit face area needed for the eye finding is 7 kB, but it can vary as the user moves closer/farther from the camera. An implicit convention of the `common` module is that the length of a message is 0 if its source task's time demand is 0, which means that the message is not sent at all. Finally, in lines 61–65, routes (in general, series of channels) are statically specified for each source-destination pair of processors.

6.3 RTSML Compiler Extension

This section describes the coupling of the RTSML/CLP and VOML-based on-line PAV technology, and extensions of the RTSML compiler for this purpose. In the coupling, the PG^{RT}TIE environment with visual objects also acts as an intermediary between the target system, a human operator and the CLP engine, as shown in Figure 6.5.

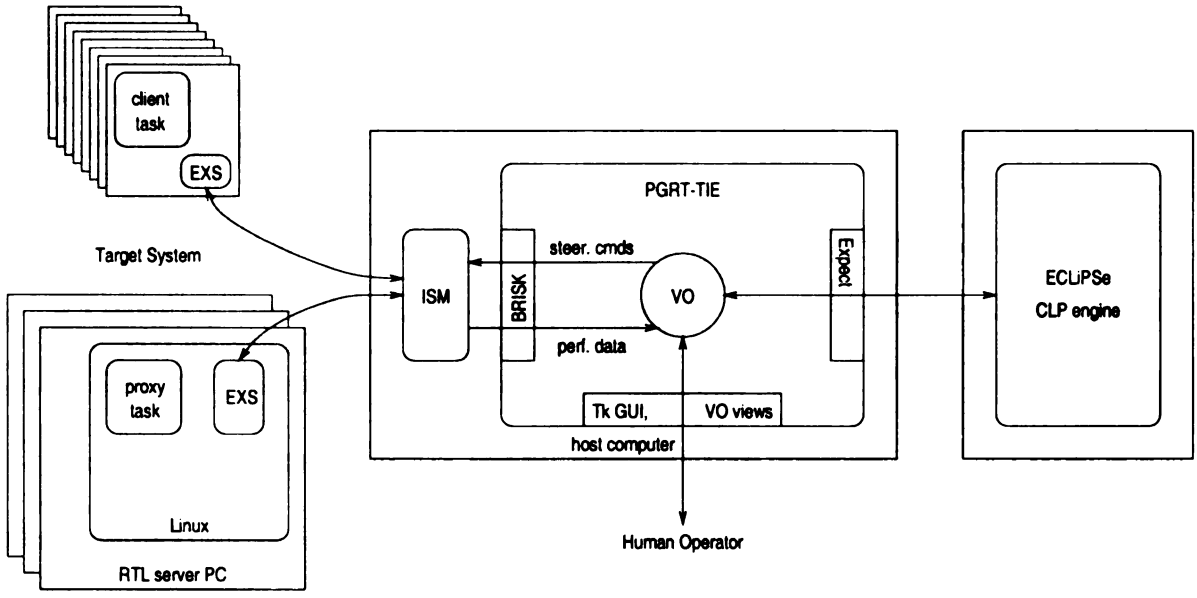

Figure 6.5: Integrated visualization, repair and steering

The performance data arrive to the environment and steering commands are sent to the target system via BRISK. Either the human operator or a visual object alone may trigger a repair as a response to, for example, deteriorated performance of the target system.

The core of compiler and three currently available modules have been extended to support the creation of VOML specifications, as well. From an RTSML specification, the compiler generates a VOML specification with one visual object for each **system** element. (That is, a target system may actually consist of multiple, disjoint or related, systems as by the RTSML notation.) This is achieved by

- adding slots to the classes of the initial objects that contain (1) data needed for graphical rendering of a system's structure and model-specific performance information of its subsystems and components, and (2) chunks of model-specific VOML/Scheme code for processing events and rendering performance informa-

tion, as well as preparing the parameter inputs and possibly automatic repair triggering;

- modifying the second compilation phase to both update some VOML data slots and augment the added methods to update some VOML data in the CLP code generation phase;
- additional cross-linking of the VOML data in the third compilation phase; and
- extending the fourth compilation phase with the splicing of all the VOML/Scheme data and code together into a VOML file.

The correspondence between the CLP and VOML/Scheme data and code is not always direct, because the latter must in some cases produce less abstract information suitable to the graphical rendering. Some VOML data, which are stored in the info and control structures, are compile-time transformations of the information supplied about the real-time parameters; Scheme code, embedded in VOML elements, is also needed in cases when similar transformations need be performed on-line by a created visual object. For example, while in the CLP domain messages are thought of as being sent via routes, a graphical view of a distributed system's physical resources should depict channels instead of the routes. Or, while different tasks may have different ranges of values for their periods, a view with normalized ranges and current values may be easier to comprehend.

In addition to the module-specific VOML/Scheme data and code, there is a convention that enables the creation of several standard views for any modeled system. Namely, each subsystem has to provide common VOML data: the name of the real-time model obeyed; identifier; and a list of its components. Similarly, each component has to provide: the name of the real-time model obeyed; identifier; subsystem identifier; a list of allocated resources or activities; and the minimum, maximum and

initial/current goodness value. After these common data, any module may or may not provide additional data and accompanying code. The standard views for an RTSML system include: a resource (processors and channels) and activity (tasks and messages) graphs, generated using a graph-drawing algorithm [34]; and one view for each of the subsystem types: processor, channel, task and message group, graphically presenting contained components and their goodness values. All the views are interlinked: the user can, for example, click on a task in the activity graph, and the task view will pop up; the user can then click on the task view to open another view of the corresponding task group, etc.

Note, on the other hand, that using this scheme, an RTSML compiler module could be used for implementing a specific performance visualization aspect only. For example, instead of using **common** as the value of the **type** attribute of a **task-group** element, one could use the name of a "real-time model" that sets no additional constraints over its tasks, but only creates a view in which specific performance aspects, such as the tasks' collective timeliness, are visualized on-line. (It is assumed that a compiler module handling the **task-group** element would "know how" to extract relevant data from the task objects based on the task class.)

This section concludes with the description of an example VOML extension of the RTSML compiler for supporting **common** tasks (to save space, the complete code is omitted). The list of modifications is as follows:

- the module-specific class **i-task-common**, which inherits from the initial object class **i-task**, specifies shared (per-class) slots containing the following VOML/Scheme code:
  - event declarations: a callback event **common-task-click** for clicking on the **common-task** view; and a data event **common-task-instance** that carries the information about a task instance's identifier, start time, response time, and time demand;
  - control structures: a Boolean (distinguishing between *false* and any other value) variable **current-common-task** for remembering which task's information is rendered in the **common-task** view; and a Boolean variable **common-task-updated** considered for enabling an IR component **common-task-updated-ir**, which updates graphical presentation of the **current-common-task**'s performance information;
  - info structures: an association list **common-tasks** that contains module-specific, on-line processed information about all **common** tasks;
  - view initializations: the title, size and span of the **common-task** view;
  - event processing: an EP component **common-task-click-ep** for responding to the user's click on "task group" or (allocated) "processor" in the **common-task** view, by updating one of predefined control structures **selected-task-group** and **selected-processor**; an EP component **common-task-instance-ep** for processing the **common-task-instance** events and updating the **common-tasks** association list with new mini-

mum, moving average and maximum values of the period, response time and time demand; and an EP component `common-repair` for responding to a pre-condition event `repair-needed` by setting the new values of some of the parameters, as needed for the CLP repair described above, based on all the `common` tasks' moving average period and time demand; and

  - information rendering: an IR component `common-task-ir` for rendering in the `common-task` view the expected performance information as previously computed by the CLP engine; and the IR component `common-task-updated-ir`, mentioned above, for rendering the current performance information that can be compared against the expected ones;

- one of the individual (per-instance) CLP code generation methods, which outputs lists of CLP variables, is augmented to update mappings between the CLP parameter and solution variables, and data (period, deadline, demand and processor allocation) kept in the VOML info structures, so that the expected performance data and resource allocation computed by the CLP engine can be loaded into the info structures after an initial solution has been found or after a repair (note that immediately after a repair, the expected performance information is consistent with previously measured ones);

- the module-specific method used in the second compilation phase, as described at the beginning of this section, is augmented to update task-related data, kept in a predefined slot of the class `i-system` that will be stored in VOML info

structures; and to collect all the VOML/Scheme code from the `i-task-common` class' shared slots into predefined slots of the class `i-system`; and

- the data and code from the above-mentioned slots of the `i-system` class are spliced into a VOML file in the final compilation phase.

6.4 Example Run-Time Session

This section completes the presentation of the integration by giving some more detail about the target system and the integrated approach that has been adopted for experimentation, and by showing an excerpt from the operation of the former being steered by the latter.

A middleware has been developed that consists of three Linux proxy tasks, described in Section 6.2, one of them acting as the main proxy for the face-tracking clients. The proxy tasks are located in a dedicated LAN that is connected to larger LAN via a switch. All the proxies and clients are instrumented using BRISK. The proxies also receive steering commands via BRISK and control the real-time tasks' periods and RTL server PC allocations, which they forward to the clients. The following is a sequence of steps that initiates a run-time session:

1. the PG^{RT}-TIE environment is started on a control host computer;

2. on the host computer, outside PG^{RT}-TIE, the human operator starts the BRISK Instrumentation System Manager (ISM);

3. on all the RTL server PCs and computers where the face-tracking clients will be running, the BRISK External Sensor (EXS) is started; on the RTL server PCs, the EXSes are given unique, known identifiers;

4. the proxies are started and given the same identifiers as those of the EXSes running on the same RTL server PCs;

5. in PG^{RT}-TIE, the human operator starts BRISK as a so-called data source, from which the visual object will receive instrumentation data records (as performance data events);

6. in PG^{RT}-TIE, the human operator loads a Scheme file generated by the VOML compiler, containing the visual object according to the target system's RTSML specification;

7. by opening, for example, the resource graph view of the visual object, and by clicking on a processor, the first (callback) event is generated; before the processor view is opened, the visual object starts the ECL^iPS^e engine on another computer via the Guile implementation of the Expect library [72], and asks it to find a feasible resource allocation according to the initial real-time parameters in the RTSML specification;

8. after the solution received from ECL^iPS^e is loaded into the info structures, a special, manually added event-processing component passes the task-to-processor allocation mapping and periods for the real-time tasks, via the BRISK interface of PG^{RT}-TIE and the ISM, to the EXSes running on the RTL server PCs;

the proxies read the allocation and periods from a memory area shared between them and the EXSes, start/stop the real-time tasks and forward these information to the clients, too;

9. the face-tracking clients are started, and they connect to the main proxy task, from which they receive unique client identifiers and the addresses of the RTL server PCs which will run the corresponding face- and eye-finding tasks, according to the initial resource allocation; subsequently, the clients establish connections with the proxies running on the assigned RTL server PCs;

10. the visual object starts receiving performance data from the proxy tasks (about the real-time tasks, such as the `common-task-instance` event mentioned in Section 6.3) and clients.

The above-mentioned special event-processing component has knowledge about the correspondence of the proxy and real-time task identifiers on one hand, and the identifiers of the processors belonging to the group `pg0` and tasks belonging to the group `servers` in the RTSML specification, on the other. Since the steering is integrated with the performance visualization, and only indirectly with the design and engineering approach, the RTSML compiler has no knowledge about a particular steering mechanism. Note, again, the disparity among the various variables involved: (1) the period and time demand of a task are system parameters, and can be monitored; (2) the deadline is also a system parameter, but is not supposed to be monitored; (3) the goodness of a task is defined as its laxity (deadline minus response, one of the fields of the `common-task-instance` event), which can be monitored but

is not a system parameter; etc. Probably the most of related knowledge that can be embedded in an RTSML compiler module is about what variables are monitorable, regardless of whether they belong to the parameters or solution. This would help the code reuse and separate design/engineering from steering concerns.

Once that the run-time session has been initiated, the visual object allows the human operator to compare visually the expected and actual system performance. The `common-task` view, mentioned in Section 6.3, is shown in Figure 6.6, together with a few other views and the GUI.

In order to receive the coordinates of the new face area and the face features within the current face area at the requested frequencies, the clients must send the messages to the real-time tasks in a pipelined fashion, with up to a predefined maximum number of outstanding messages in the LAN and servers' buffers. On the other hand, if a message does not arrive in time to be processed by a real-time task, the real-time task immediately returns the CPU to the RTL scheduler and waits for the next period. In this case, the sending of a `common-task-instance` event via BRISK is skipped. Too frequent skips can be noticed by the moving-average task period that becomes significantly larger than the projected period. Similarly, if it becomes harder to find a workstation user's face or eyes, the time demand by the corresponding real-time task(s) may become larger than the expected maximum. The clients also may send data about their perceived quality of service, which, for example, may drop if the face-tracking needs higher face- or eye-finding frequencies; these data may be processed and visualized by another visual object, possibly on another computer. In a situation like these, the human operator may decide to trigger a repair, by pressing the "Trigger

Figure 6.6: A snapshot from the RTSML-based visual object

Repair" button in the GUI, which will result in passing of the measured performance information as new parameters to ECL^{i}PS^e. With a more elaborate GUI, generated indirectly by the RTSML compiler modules, the human operator could be allowed to manually override the new parameters. If a solution is found by ECL^{i}PS^e, it is enforced upon the target system in a way similar to the steps 8–9 in the above sequence. Depending on the target system model's complexity and the values of the parameters, the repair approach's solving time may vary from about the same as for solving from scratch, to about an order of magnitude faster than for solving from scratch, as discussed in Section 3.3.2.

6.5 Summary

A novel technology was presented that integrates multiple other approaches: the design and engineering of complex real-time systems; support for their dynamic reconfiguration; on-line PAV with steering/reconfiguration. The presentation included an example distributed real-time system and details about how the involved technologies are used in an integrated manner to help system developers and users achieve better system performance.

Chapter 7

Conclusions and Future Work

This dissertation describes the work in four areas related to heterogeneous, parallel and distributed real-time systems. The main focus is on a compiler-based approach to the design and engineering of complex real-time systems, which is augmented by instrumentation, performance analysis & visualization, and dynamic system reconfiguration.

7.1 Research Contributions

Based on the analogy with non-linear circuits in other areas, such as the control theory and digital systems, the design and engineering approach for complex real-time systems presented in this dissertation views a complex real-time system as a network of integrated and interoperating subsystems and components. It attempts to help developing distributed, heterogeneous real-time systems that can operate in

more realistic scenarios by integrating specialized parts that abide extant models from real-time theory.

The observation that real-time requirements are best termed as time and resource constraints expressed via equations and inequalities of the real-time models, led to the application of CLP as a promising engineering tool for specifying through integration and solving real-time design problems.

An SGML-based specification language (RTSML) has been developed that supports integration of heterogeneous components and subsystems into complex real-time systems. Its compiler generates CLP code and can be extended with new modules to support other real-time models. An especially interesting and important aspect of the CLP-based problem solving is the repair-based approach. The experiments showed that it is promising for coping with moderate dynamic changes in the system parameters. Portions of this work were published in [8].

In the area of distributed instrumentation systems, the concept of a portable and flexible IS kernel was proven by the development and evaluation of BRISK. It is expected to be beneficial for instrumenting heterogeneous distributed real-time systems due to its ability to adapt and tune so as to minimize the intrusion on a real-time target system. Portions of this work were published in [10].

A novel on-line performance analysis and visualization technology for heterogeneous parallel and distributed systems was developed. It supports rapid PAV prototyping, automated design of PAV tools, object orientation, distributability, portability, code reuse and flexibility. A high-level visual object specification language

(VOML) has been developed. Its compiler generates visual object code that receives instrumentation data from BRISK. Portions of this work were published in [9].

The RTSML compiler was extended with support for the developed on-line PAV technology. The BRISK IS kernel is also utilized in a novel, mainly automated, integrated technology that includes the design and engineering of complex real-time systems, system-specific PAV, and PAV-driven and model-based dynamic reconfiguration.

7.2 Future Work

Work in the complex real-time system design and engineering based on the approach presented in this dissertation may include:

- addition of new compiler modules for other real-time models, with an emphasis on stochastic ones,
- static (e.g., introduction of redundant constraints) and dynamic (e.g., support for user/module-specified labeling priorities) CLP search/repair optimization, and
- potential extensions to RTSML (e.g., broader support for end-to-end constraints; finer-than-task granularity of real-time software; support for formal methods).

The basic BRISK implementation may be extended in several ways. First of all, hierarchical instrumentation data collection, preprocessing and reduction—via the notion of local ISM—would make it usable for large-scale parallel/distributed systems. For programmable, run-time control of BRISK's components, it may be extended with a portable interpreter [73] over the existing communication infrastructure

for BRISK control. (It is assumed that resources needed for the IS control are insignificant compared to those used for the main IS functions under average operating conditions.) A system type may be added to event records to allow for low-latency, out-of-band events. The NOTICE macros could be made multithread-safe. The support for causally-related events may be extended with one-to-many and many-to-one dependencies, such as those described in [67]. As needed, more interfaces to consumer tools would be added, e.g., SDDF trace files and DCOM [20]. More evaluations ought to be done, especially of latency and causally-related events.

Besides potential extensions to the VOML language and enhancements of the underlying VO implementations, the most promising aspect of the on-line PAV technology is its being a base for the development of advanced on-line PAV tools. The EPIRA and the declarative style of composing an HLVO from reusable components allow for easy automated generation of domain-specific on-line PAV tools, analogous to the one presented in Chapter 6. On the other hand, the VOML compiler could be extended into a more powerful tool, such as an expert system for, e.g., pattern-based, on-line PAV composition.

The application of the PAV-driven dynamic reconfiguration of a complex real-time system could be extended. For example, there could be multiple, distributed and cooperating visual objects and CLP tools that would exploit the repair-based dynamic system reconfiguration for different purposes, such as gradual performance improvement; or simultaneously trying and deciding from multiple, different repairs (with different assumptions and/or repair heuristics), etc.

Appendices

Appendix A

RTSML Document Type Definition (excerpt)

This appendix provides the precise definition of RTSML, which is subject to changes and extensions in the future. The header part of the DTD is omitted to save space. Element attributes marked #IMPLIED are inferred from the context by the compiler, unless specified by the user.

```
...
<!--================ Document Body ====================================-->

<!ELEMENT PROCESSOR - - (TASK*)>
<!ATTLIST PROCESSOR
        id              ID      #REQUIRED
        sys             IDREF   #IMPLIED
        type            CDATA   #REQUIRED
        group           IDREF   #IMPLIED
        >

<!ELEMENT PROCESSOR-GROUP - - (PROCESSOR*)>
<!ATTLIST PROCESSOR-GROUP
        id              ID      #REQUIRED
```

```
        sys             IDREF   #IMPLIED
        type            CDATA   #REQUIRED
        >

<!ELEMENT CHANNEL - - (#PCDATA)>
<!ATTLIST CHANNEL
        id              ID      #REQUIRED
        sys             IDREF   #IMPLIED
        type            CDATA   #REQUIRED
        group           IDREF   #IMPLIED
        >

<!ELEMENT CHANNEL-GROUP - - (CHANNEL*)>
<!ATTLIST CHANNEL-GROUP
        id              ID      #REQUIRED
        sys             IDREF   #IMPLIED
        type            CDATA   #REQUIRED
        >

<!ELEMENT ROUTE - O EMPTY>
<!ATTLIST ROUTE
        id              ID      #REQUIRED
        sys             IDREF   #IMPLIED
        type            CDATA   #REQUIRED
        table           IDREF   #IMPLIED
        source          IDREF   #REQUIRED
        destination     IDREF   #REQUIRED
        channels        IDREFS  #REQUIRED
        >

<!ELEMENT ROUTING-TABLE - - (ROUTE*)>
<!ATTLIST ROUTING-TABLE
        id              ID      #REQUIRED
        sys             IDREF   #IMPLIED
        type            CDATA   #REQUIRED
        >

<!ELEMENT TASK - - (#PCDATA)>
<!ATTLIST TASK
        id              ID      #REQUIRED
        sys             IDREF   #IMPLIED
        type            CDATA   #REQUIRED
        group           IDREF   #IMPLIED
        processors      IDREFS  #IMPLIED
        >

<!ELEMENT TASK-GROUP - - (TASK*)>
```

```
<!ATTLIST TASK-GROUP
        id              ID      #REQUIRED
        sys             IDREF   #IMPLIED
        type            CDATA   #REQUIRED
        >

<!ELEMENT MESSAGE - - (#PCDATA)>
<!ATTLIST MESSAGE
        id              ID      #REQUIRED
        sys             IDREF   #IMPLIED
        type            CDATA   #REQUIRED
        group           IDREF   #IMPLIED
        source          IDREF   #REQUIRED
        destinations IDREFS  #REQUIRED
        >

<!ELEMENT MESSAGE-GROUP - - (MESSAGE*)>
<!ATTLIST MESSAGE-GROUP
        id              ID      #REQUIRED
        sys             IDREF   #IMPLIED
        type            CDATA   #REQUIRED
        >

<!ELEMENT SYSTEM - - ((PROCESSOR-GROUP |
                       CHANNEL-GROUP |
                       ROUTING-TABLE |
                       TASK-GROUP |
                       MESSAGE-GROUP)*)>
<!ATTLIST SYSTEM
        id              ID      #REQUIRED
        type            CDATA   #REQUIRED
        >

<!ELEMENT BODY O O (SYSTEM)*>

<!--================ Document Structure ==============================-->

<!ENTITY % version.attr "VERSION CDATA #FIXED '%RTSML.Version;'">

<!ELEMENT RTSML O O (HEAD, BODY)>
<!ATTLIST RTSML
        %version.attr;
        >
```

Appendix B

VOML Document Type Definition (excerpt)

This appendix provides the precise definition of VOML, which is subject to changes and extensions in the future. The header part of the DTD is omitted to save space. Element attributes marked **#IMPLIED** are inferred from the context by the compiler, unless specified by the user.

```
...
<!--================ Document Body ===================================-->

<!ENTITY % datatype     "(int|real|string|boolean|list|vector)">
<!ENTITY % guitype      "(checkbutton|textbox|slidebar)">
<!ENTITY % cbacktype    "(click|keystroke|resize|rescale|gui)">
<!ENTITY % recordtype   "(entry|exit|atomic|pg-entry|pg-exit|pg-atomic)">
<!ENTITY % inputtype    "INPUT|PRECONDITION|POSTCONDITION">
<!ENTITY % graphtype    "POINT|LINE|RECTANGLE|POLYGON|ARC|ELLIPSE|TEXT|FIGURE">
<!ENTITY % viewcontrol "OPEN-VIEW|CLOSE-VIEW|SCROLL|RESIZE|RESCALE|VIEW-INFO|
                        SNAPSHOT|TEXT-EXTENTS">
<!ENTITY % scrolldir    "(left|right|up|down)">
<!ENTITY % svaltype     "(endval|displacement)">
<!ENTITY % boolval      "(yes|no)">
```

```
<!ENTITY % clipval     "(edge|margin)">

<!ELEMENT DATA-FIELD - O EMPTY>
<!ATTLIST DATA-FIELD
        name    NAME        #REQUIRED
        index   NUMBER      #IMPLIED
        type    %datatype   #IMPLIED
        >

<!ELEMENT DATA-EVENT - - (DATA-FIELD|#PCDATA)*>
<!ATTLIST DATA-EVENT
        name       ID           #REQUIRED
        rtype      %recordtype  #REQUIRED
        etype      CDATA        #REQUIRED
        >

<!ELEMENT CALLBACK-EVENT - O EMPTY>
<!ATTLIST CALLBACK-EVENT
        name       ID           #REQUIRED
        ctype      %cbacktype   #REQUIRED
        view       IDREF        #IMPLIED
        var        IDREF        #IMPLIED
        >

<!ELEMENT CONDITION-EVENT - - (#PCDATA)>
<!ATTLIST CONDITION-EVENT
        name       ID       #REQUIRED
        infos      IDREFS   #IMPLIED
        controls   IDREFS   #IMPLIED
        >

<!ELEMENT EVENT-DECLARATIONS - - (DATA-EVENT|CALLBACK-EVENT|CONDITION-EVENT)*>

<!ELEMENT VARIABLE - O EMPTY>
<!ATTLIST VARIABLE
        name     ID          #REQUIRED
        title    CDATA       #IMPLIED
        type     %datatype   #REQUIRED
        init     CDATA       #IMPLIED
        gui      %guitype    #IMPLIED
        gparam   CDATA       #IMPLIED
        >

<!ELEMENT INFO-STRUCTURES - - (VARIABLE)*>
<!ELEMENT CONTROL-STRUCTURES - - (VARIABLE)*>

<!ELEMENT (%inputtype) - - (#PCDATA)>
```

```
<!ATTLIST (%inputtype)
        name  CDATA  #REQUIRED
        >
<!ELEMENT (%viewcontrol) - O EMPTY>

<!ATTLIST OPEN-VIEW
        view        CDATA        #IMPLIED>

<!ATTLIST CLOSE-VIEW
        view        CDATA        #IMPLIED>

<!ATTLIST SCROLL
        view        CDATA        #IMPLIED
        direction   %scrolldir   #REQUIRED
        valtype     %svaltype    #REQUIRED
        value       CDATA        #REQUIRED
        >

<!ATTLIST RESIZE
        view        CDATA        #IMPLIED
        window      NUMBERS      #REQUIRED
        >

<!ATTLIST RESCALE
        view        CDATA        #IMPLIED
        factors     CDATA        #REQUIRED
        >

<!ATTLIST VIEW-INFO
        view        CDATA        #IMPLIED
        >

<!ATTLIST SNAPSHOT
        view        CDATA        #IMPLIED
        filename    CDATA        #IMPLIED>


<!ATTLIST TEXT-EXTENTS
        content     CDATA        #IMPLIED
        font        CDATA        #IMPLIED
        halign      CDATA        #IMPLIED
        valign      CDATA        #IMPLIED>

<!ELEMENT (%graphtype) - O EMPTY>

<!ATTLIST POINT
        view    CDATA     #IMPLIED
```

```
        coords  CDATA     #REQUIRED
        color   CDATA     #IMPLIED
        adapt   %boolval  #IMPLIED
        clip    %clipval  #IMPLIED
        >

<!ATTLIST LINE
        view    CDATA     #IMPLIED
        from    CDATA     #REQUIRED
        to      CDATA     #REQUIRED
        thick   CDATA     #IMPLIED
        color   CDATA     #IMPLIED
        adapt   %boolval  #IMPLIED
        clip    %clipval  #IMPLIED
        >

<!ATTLIST RECTANGLE
        view    CDATA     #IMPLIED
        from    CDATA     #REQUIRED
        to      CDATA     #REQUIRED
        color   CDATA     #IMPLIED
        fillc   CDATA     #IMPLIED
        adapt   %boolval  #IMPLIED
        clip    %clipval  #IMPLIED
        >

<!ATTLIST POLYGON
        view    CDATA     #IMPLIED
        points  CDATA     #REQUIRED
        color   CDATA     #IMPLIED
        fillc   CDATA     #IMPLIED
        adapt   %boolval  #IMPLIED
        clip    %clipval  #IMPLIED
        >

<!ATTLIST ARC
        view    CDATA     #IMPLIED
        center  CDATA     #REQUIRED
        radii   CDATA     #REQUIRED
        angles  CDATA     #REQUIRED
        color   CDATA     #IMPLIED
        fillc   CDATA     #IMPLIED
        adapt   %boolval  #IMPLIED
        clip    %clipval  #IMPLIED
        >

<!ATTLIST ELLIPSE
```

```
        view    CDATA     #IMPLIED
        center  CDATA     #REQUIRED
        radii   CDATA     #REQUIRED
        angle   CDATA     #IMPLIED
        color   CDATA     #IMPLIED
        fillc   CDATA     #IMPLIED
        adapt   %boolval  #IMPLIED
        clip    %clipval  #IMPLIED
        >

<!ATTLIST TEXT
        view     CDATA      #IMPLIED
        content  CDATA      #REQUIRED
        coords   CDATA      #REQUIRED
        halign   CDATA      #IMPLIED
        valign   CDATA      #IMPLIED
        fcolor   CDATA      #IMPLIED
        bcolor   CDATA      #IMPLIED
        font     CDATA      #IMPLIED
        adapt    %boolval   #IMPLIED
        clip     %clipval   #IMPLIED
        >

<!ATTLIST FIGURE
        view           CDATA      #IMPLIED
        filename       CDATA      #REQUIRED
        orig-origin    NUMBERS    #IMPLIED
        orig-extents   NUMBERS    #IMPLIED
        world-origin   CDATA      #IMPLIED
        world-extents  CDATA      #IMPLIED
        adapt          %boolval   #IMPLIED
        clip           %clipval   #IMPLIED
        >
<!ELEMENT OVERRIDE-FIELD - O EMPTY>
<!ATTLIST OVERRIDE-FIELD
        name      CDATA   #REQUIRED
        new-value CDATA   #REQUIRED
        >

<!ELEMENT PREPROCESS-INPUTS - - (OVERRIDE-FIELD)*>

<!ELEMENT DESCRIPTION - - (#PCDATA)>

<!ELEMENT EP-COMPONENT - - ((PREPROCESS-INPUTS)*, DESCRIPTION, (%inputtype)*)>
<!ATTLIST EP-COMPONENT
        name      NAME    #REQUIRED
        inputs    NAMES   #IMPLIED
```

```
        infos     IDREFS  #IMPLIED
        controls  IDREFS  #IMPLIED
        >

<!ELEMENT END-WITH - - ((%viewcontrol|%graphtype|#PCDATA)*)>

<!ELEMENT IR-COMPONENT - - (#PCDATA, DESCRIPTION,
                            (%viewcontrol|%graphtype|#PCDATA|END-WITH)*)>
<!ATTLIST IR-COMPONENT
        name      NAME     #REQUIRED
        views     IDREFS   #IMPLIED
        infos     IDREFS   #IMPLIED
        controls  IDREFS   #IMPLIED
        refresh   NAMES    #IMPLIED
        enable    CDATA    #IMPLIED
        buffer    %boolval #IMPLIED
        bcolor    CDATA    #IMPLIED
        >

<!ELEMENT EVENT-PROCESSING - - (EP-COMPONENT)*>

<!ELEMENT INFO-RENDITION - - (IR-COMPONENT)*>

<!ELEMENT VIEW - - (#PCDATA, DESCRIPTION, (%viewcontrol|%graphtype|#PCDATA)*)>
<!ATTLIST VIEW
        name     ID       #REQUIRED
        title    CDATA    #IMPLIED
        infos    IDREFS   #IMPLIED
        controls IDREFS   #IMPLIED
        window   NUMBERS  #REQUIRED
        world    CDATA    #REQUIRED
        margins  CDATA    #IMPLIED
        fcolor   CDATA    #IMPLIED
        bcolor   CDATA    #IMPLIED
        adirs    NAMES    #IMPLIED
        >

<!ELEMENT UTILITY-CODE - - (#PCDATA)>
<!ATTLIST UTILITY-CODE
        infos    IDREFS   #IMPLIED
        controls IDREFS   #IMPLIED
        >

<!ELEMENT VIEW-INITIALIZATIONS - - (VIEW)+>

<!ELEMENT VISUAL-OBJECT - -
        (EVENT-DECLARATIONS,
```

```
          INFO-STRUCTURES,
          CONTROL-STRUCTURES,
          (UTILITY-CODE)*,
          VIEW-INITIALIZATIONS,
          EVENT-PROCESSING,
          INFO-RENDITION)
         >
<!ATTLIST VISUAL-OBJECT
         name  ID            #REQUIRED
         type  (async|sync)  #REQUIRED
         aqual NAME          #IMPLIED
         >

<!ELEMENT BODY O O (VISUAL-OBJECT)*>

<!--================ Document Structure ================================-->

<!ENTITY % version.attr "VERSION CDATA #FIXED '%VOML.Version;'">

<!ELEMENT VOML O O (HEAD, BODY)>
<!ATTLIST VOML %version.attr;>
```

Bibliography

Bibliography

[1] Gul A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[2] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, April 25 1994.

[3] Corinne Ancourt et al. Automatic data mapping of signal processing applications. In *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors (ASAP'97)*, July 14–16 1997.

[4] Ruth A. Aydt. The Pablo self-defining data format. Technical report, Department of Computer Science, University of Illinois, Urbana, Illinois 61801, April 11 1995.

[5] R. Bahgat and R. Yang. A distributed repair-based technique for constraint satisfaction. In *Proceedings of the 7th International Conference on Intelligent Systems*, July 1–3 1998.

[6] Aleksandar Bakić, Matt W. Mutka, and Diane T. Rover. Real-time system performance visualization and analysis using distributed visual objects. In *Proceedings of the IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, December 2 1997.

[7] Aleksandar Bakić, Matt W. Mutka, and Diane T. Rover. Performance optimization of distributed applications in an extensible, adaptive environment. Accepted for publication in Elsevier Science Future Generation Computer Systems, special issue on performance data mining., 1999.

[8] Aleksandar M. Bakić and Matt W. Mutka. A compiler-based approach to design and engineering of complex real-time systems. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, May 31–June 5 1999.

[9] Aleksandar M. Bakić, Matt W. Mutka, and Diane T. Rover. BRISK: A portable and flexible distributed instrumentation system. In *Proceedings of the IEEE 2nd Merged Symposium IPPS/SPDP 1999 — 13-th International Parallel Processing Symposium & 10-th Symposium on Parallel and Distributed Processing*, April 12–16 1999.

[10] Aleksandar M. Bakić, Matt W. Mutka, and Diane T. Rover. An on-line performance visualization technology. In *Proceedings of the IEEE Heterogeneous Computing Workshop, in conjunction with the 2-nd Merged Symposium IPPS/SPDP 1999 — 13-th International Parallel Processing Symposium & 10-th Symposium on Parallel and Distributed Processing*, April 12–16 1999.

[11] Vera Bakić and George Stockman. Menu selection by facial aspect. In *Proceedings of the 12th Conference on Vision Interface '99*. Canadian Image Processing and Pattern Recognition Society, May 1999.

[12] Michael Barabanov. A Linux-based real-time operating system. Master's thesis, New Mexico Tech, 1997. [Online] Available `http://www.rtlinux.org/`.

[13] Renate Beckmann, Ulrich Bieker, and Ingolf Markhof. Application of constraint logic programming for VLSI CAD tools. In *Proceedings of the Conference Constraints in Computational Logics*, September 1994.

[14] S. Bennett. *Real-Time Computer Control: An Introduction.* Prentice Hall, 1994.

[15] G. Bernat and A. Burns. Combining (n m)-hard deadlines with dual priority scheduling. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, December 1997.

[16] Peter J. Bickel and Kjell A. Doksum. *Mathematical Statistics—Basic Ideas and Selected Topics.* Prentice Hall, 1977.

[17] D. Bobrow, L. DiMichiel, R.P. Gabriel, S. Keene, G. Kiczales, and D. Moon. A Common LISP object system specification: X3J13 document 88–002R. *SIGPLAN Notices*, 23, September 1988.

[18] Robert Bosch, Chris Stolte, Diane Tang, John Gerth, Mendel Rosenblum, and Pat Hanrahan. Rivet: A flexible environment for computer systems visualization. *Computer Graphics*, 34(1), February 2000.

[19] Per Bothner. Kawa - compiling dynamic languages to Java VM. In *Proceedings of the 1998 USENIX Annual Technical Conference, New Orleans*, 1999. [Online] Available http://www.gnu.org/software/kawa/.

[20] Don Box. *Essential COM.* Addison-Wesley, January 1998.

[21] A. Burns and A.J. Wellings. HRT-HOOD: A structured design method for hard real-time Ada systems. *Real-Time Safety Critical Systems*, 3, 1995.

[22] Saurav Chatterjee and Jay Strosnider. Distributed pipeline scheduling: A framework for distributed, heterogeneous real-time system design. *The Computer Journal*, 38(4):271–285, 1995.

[23] Sheng-Tzong Cheng. *Scheduling and Allocation in Multiprocessor Systems.* PhD thesis, Department of Computer Science, University of Maryland, 1995.

[24] S. E. Chodrow, F. Jahanian, and M. Donner. Run-time monitoring of real-time systems. In *Proceedings of the Real-Time Systems Symposium*, pages 74–83, December 4–6 1991.

[25] James Clark. `sgmls`: A validating SGML parser, 1993. [Online] Available `ftp://ftp.jclark.com/pub/sgmls/`.

[26] William Clinger and Jonathan Rees, editors. *Revised(4) Report on the Algorithmic Language Scheme.* IEEE, November 2 1991.

[27] Object Management Group Technical Committee. Realtime CORBA 1.0 RFP, 1999. [Online] Available `http://www.omg.org/techprocess/meetings/schedule/Realtime_CORBA_1.0_RFP.html`.

[28] Corman, Lieserson, and Rivest. *Introduction to Algorithms.* McGraw-Hill, 1990.

[29] F. Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3:146–158, 1989.

[30] Paul S. Dodd and Chinya V. Ravishankar. Monitoring and debugging distributed real-time programs. *Software—Practice and Experience*, 22(10):863–877, October 1992.

[31] Jack Dongarra et al. MPI: A message-passing interface standard. Technical report, Oak Ridge National Laboratory, June 1995.

[32] G. Giest et al. *PVM 3.0 User's Guide and Reference Manual.* ORNL/TM–12187, February 1993.

[33] Laurent Fribourg. A closed-form evaluation for extended timed automata. Technical Report LSV–98–2, Laboratoire Spécification et Vérification, Ecole Normale Supérieure de Cachan, 61, avenue du Président Wilson, 94235 Cachan, Cedex, France, March 1998.

[34] Mehldau Frick, Ludwig. A fast adaptive layout algorithm for unidirected graphs. In *Proceedings of Graph Drawing '94.* Springer-Verlag, 1994. [Online] Available `http://i44www.info.uni-karlsruhe.de/~frick/gd/gd94p.ps.gz` and `ftp://ftp.uni-passau.de:/pub/local/graphed`.

[35] Markus P.J. Fromherz and John Conley. Issues in reactive constraint solving. In *Proceedings of the Workshop on Concurrent Constraint Programming for Time Critical Applications - COTIC 97*, November 1997.

[36] Thom Frühwirth, Alexander Herold, Volker Kĉhenhoff, Thierry Le Provost, Pierre Lim, Eric Monfroy, and Mark Wallace. Constraint logic programming—an informal introduction. Technical report, European Computer-Industry Research Center, Arabellastr. 17, D-8000 München 81, Germany, 1992.

[37] Thom Früwirth. Temporal annotated constraint logic programming. *Journal of Symbolic Computation, Special issue on Executable Temporal Logics*, 22:555–583, 1996.

[38] FWEB. A literate programming tool. The FTP server of the Plasma Physics Laboratory, Princeton University (PPPL), 1995. [Online] Available `ftp://ftp.pppl.gov/pub/fweb/`.

[39] R. Gerber, S. Hong, D. Kang, and M. Saksena. *Formal Methods in Real-Time Computing*, chapter End-to-End Design of Real-Time Systems, pages 237–265. John Wiley & Sons, 1996.

[40] Martin Gergeleit, Jörg Kaiser, and Hermann Streich. Checking timing constraints in distributed object-oriented programs. *OOPS Messenger*, 7(1):51–58, January 1996.

[41] David E. Goldberg. *Genetic Algorithms*, chapter Computer Implementation of a Genetic Algorithm, pages 85–86. Addison-Wesley Publishing Company, Inc., 1989.

[42] Charles F. Goldfarb. *The SGML Handbook*. Oxford University Press, 1990.

[43] Weiming Gu, Grep Eisenhauer, Eileen Kramer, Karsten Schwan, John Stasko, and Jeffrey Vetter. Falcon: On-line monitoring and steering of large-scale parallel programs. Technical Report GIT-CC-94–21, Georgia Institute of Technology, 1994.

[44] Weiming Gu, Jeffrey Vetter, and Karsten Schwan. An annotated bibliography of interactive program steering. Technical report, College of Computing, Georgia Institute of Technology, November 1 1994.

[45] Christophe Guettier. *Global Optimisation of DSP Application Mapping onto Parallel Architectures using Constraint Logic Programming*. PhD thesis, Ecole des Mines de Paris, Boulevard Saint Michel, Paris, France, December 12 1997.

[46] G. Gupta and E. Pontelli. A constraint-based approach for specification and verification of real-time systems. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, December 2–5 1997.

[47] Dieter Haban and Dieter Wybranietz. A hybrid monitor for behavior and performance analysis of distributed systems. *IEEE Transactions on Software Engineering*, 16(2):197–211, February 1990.

[48] Bruno Haible. CLISP, a Common LISP implementation, 1997. [Online] Available `http://clisp.cons.org/~haible/clisp.html`.

[49] Michael T. Heath and Jennifer Etheridge Finger. ParaGraph: A tool for visualizing performance of parallel programs. Technical report, Oak Ridge National Laboratory, Oak Ridge, TN, 1994.

[50] Harold H. Hersey, Steven T. Hackstadt, Lars T. Hansen, and Allen D. Malony. Viz: A visualization programming system. Technical Report CIS-TR-96-05, Department of Computer and Information Science, University of Oregon, Eugene, OR 97403-1202, April 1996.

[51] T. Hogg. Phase transitions in constraint satisfaction search, 1997. [Online] Available `http://www.parc.xerox.com/spl/groups/dynamics/www/-constraints.html`.

[52] Christian Holzbaur. OFAI clp(Q,R) manual. Technical report, Austrian Research Institute for Artificial Intelligence (OFAI), Schottengasse 3, A-1010 Vienna, Austria, December 1995.

[53] E. Hyvönen. Constraint reasoning based on interval arithmetic. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, 1989.

[54] Sun Microsystems Inc., editor. *Solaris 2.4 Software Developer Answerbook.* Sun Microsystems, Inc., 1995.

[55] ISR. What is systems engineering? Technical report, The Institute for Systems Research, University of Maryland, 1999. [Online] Available `http://www.-isr.umd.edu/`.

[56] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[57] F. Jahanian and A. Goyal. A formalism for monitoring real-time constraints at run-time. In *Proceedings of the IEEE Fault-Tolerant Computing Symposium*, pages 148–155, June 1990.

[58] F. Jahanian, R. Rajkumar, and S. Raju. Runtime monitoring of timing constraints in distributed real-time systems. *Real-Time Systems*, 7(3):247–274, November 1994.

[59] Farnam Jahanian and Aloysius Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, SE-12(9):890–904, September 1986.

[60] Farnam Jahanian and Aloysius K. Mok. A graph-theoretic approach for timing analysis and its implementation. *IEEE Transactions on Computers*, C-36(8), August 1987.

[61] Raj Jain. *The Art of Computer Systems Performance Analysis.* John Wiley & Sons, Inc., 1991.

[62] E. Douglas Jensen. Some of my perspectives on real-time computer systems, 1995. [Online] Available `http://www.realtime-os.com/realtime.html/`.

[63] E. Douglas Jensen. A real-time manifesto, 1996. [Online] Available `http://www.realtime-os.com/rtmanifesto/rtmani_0.html`.

[64] D. Kandlur, K.G. Shin, and D. Ferrari. Real-time communication in multi-hop networks. *IEEE Transactions on Parallel and Distributed Systems*, pages 1044–1056, October 1994.

[65] P.C. Kanellakis and D.Q. Goldin. Constraint programming and database query languages. In *Proceedings of the Symposium on Theoretical Aspects of Computer Software*, number 789 in Lecture Notes in Computer Science, pages 96–120, April 1994.

[66] Dong-In Kang, Richard Gerber, and Manas Saksena. Performance-based design of distributed real-time systems. In *Proceedings of the Third IEEE Real-Time Technology and Applications Symposium*, 1997.

[67] Doug Kimelman and Dror Zernik. On-the-fly topological sort—a basis for interactive debugging and live visualization of parallel programs. In *Summary of the ONR/ACM Workshop on Parallel and Distributed Debugging*, May 17–18 1993.

[68] David G. Korn. UWIN—UNIX for Windows. Technical report, AT&T Research, 1998. [Online] Available `http://www.research.att.com/sw/tools/-uwin/`.

[69] S. Lakmazaheri and W. Rasdorf. Constraint logic programming for the analysis and partial synthesis of truss structures. *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing*, 3(3):157–173, 1989.

[70] Frank Lange, Reinhold Kroeger, and Martin Gergeleit. JEWEL: Design and implementation of a distributed measurement system. Technical report, German National Research Center for Computer Science (GMD), P. O. Box 1316, Schloss Birlinghoven, D–5205 St. Augusting 1, Federal Republic of Germany, April 1992.

[71] J. L. Lassez and J. Jaffar. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages (POPL)*, pages 111–119, January 1987.

[72] Don Libes. *Exploring Expect.* O'Reilly & Associates ISBN 1–56592–090–2, 1994.

[73] Thomas Lord. An anatomy of Guile/the interface to Tcl/Tk. In *Proceedings of Usenix Tcl/Tk Workshop '95*, 1995. [Online] Available `http://www.red--bean.com/guile/`.

[74] ObjecTime Ltd. Embedded software solutions for the connected world, 1999. [Online] Available `http://www.objectime.com/`.

[75] Shikharesh Majumdar. Application of relational arithmetic in performance analysis of computing systems. In *Workshop on Interval Constraints (International Logic Programming Symposium ILPS'95)*, December 1995.

[76] Allen D. Malony, Daniel A. Reed, and Harry A.G. Wijshoff. Performance measurement intrusion and perturbation analysis. *IEEE Transactions on Parallel and Distributed Systems*, 3(4):433–450, July 1992.

[77] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent System's Specification.* Springer-Verlag, 1992.

[78] Paolo Mantegazza. DIAPM-RTAI for Linux: WHYs, WHATs and HOWs. In *Proceedings of the Real-Time Linux Workshop*, November 1999. [Online] Available `http://www.thinkingnerds.com/projects/-rtl-ws/presentations.html`.

[79] Barton P. Miller, Jonathan M. Cargille, R. Bruce Irvin, Krishna Kunchithapadam, Mark D. Callagha, Jeffrey K. Hollingsworth, Karen L. Karavanic, and Tia Newhall. The Paradyn parallel performance measurement tools. *IEEE Computer*, November 1995.

[80] Barton P. Miller, Cathryn Macrander, and Stuart Sechrest. A distributed programs monitor for Berkeley UNIX. *Software—Practice and Experience*, 16(2):183–200, February 1986.

[81] K. Mok and F. Jahanian. Modecharts: A specification language for real-time systems. *IEEE Transactions on Software Engineering*, 20(12):933–947, December 1994.

[82] Thomas J. Mowbray and Ron Zahavi. *The Essential CORBA—System Integration Using Distributed Objects.* The Object Management Group ISBN 0–471–10611–9, 1997.

[83] Peter Murray-Rust. Chemical markup language, 1997. [Online] Available `http://www.venus.co.uk/omf/cml/`.

[84] Matt W. Mutka and Jong-Pyng Li. A tool for allocating periodic real-time tasks to a set of processors. *Journal of Systems and Software, Vol. 24*, pages 135–148, 1995.

[85] Brian Nielsen, Shangping Ren, and Gul Agha. Specification of real-time interaction constraints. In *Proceedings of the First International Symposium on Object-Oriented Real-Time Computing*, 1998.

[86] Christor H. Papadimitriou. *Computational Complexity*, chapter Reductions and Completeness, page 159. Addison-Wesley Publishing Company, 1994.

[87] D. T. Peng, K. G. Shin, and T. F. Abdelzaher. Assignment and scheduling of communicating periodic tasks in distributed real-time systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(12), December 1997.

[88] Amir Pnueli. The temporal logic of programs. In *The Proceedings of the 18th Annual Symposium on Foundations of Computer Science*. IEEE, 1977.

[89] D. Priddin and A. Burns. Integrating real-time structured design and formal techniques. In *Proceedings of the Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, number 1486 in Lecture Notes in Computer Science, pages 92–102, September 14–18 1998.

[90] D. Reed, K. Shields, W. Scullin, L. Tavera, and C. Elford. Virtual reality and parallel systems performance analysis. *IEEE Computer*, 28(11):55–67, November 1995.

[91] Daniel A. Reed, Ruth A. Aydt, Tara M. Madhyastha, Roger J. Noe, Keith A. Shields, and Bradley W. Schwartz. An overview of the Pablo performance analysis environment. Technical report, Department of Computer Science, University of Illinois, Urbana, Illinois 61801, November 7 1992.

[92] Daniel A. Reed, Ruth A. Aydt, Tara M. Madhyastha, Roger J. Noe, Keith A. Shields, and Bradley W. Schwartz. An overview of the Pablo performance analysis environment. Technical report, Department of Computer Science, University of Illinois, Urbana, Illinois 61801, November 7 1992.

[93] Shangping Ren, Gul A. Agha, and Masahiko Saito. A modular approach for programming distributed real-time systems. *Journal of Parallel and Distributed Computing, Special Issue on Object-Oriented Real-Time Systems*, 36(1):4–12, July 10 1996.

[94] Kay Römer and Arno Puder. MICO: CORBA 2.0 implementation. Technical report, Computer Science Department, University of Frankfurt, Germany, 1997. [Online] Available `http://diamant-atm.vsb.cs.uni-frankfurt.de/~mico/`.

[95] Alessandro Rubini. *Linux Device Drivers*. O'Reilly & Associates, 1998.

[96] Hani El Sakkout and Stefano Novello. Repair through conflict detection in ECL^{i}PS^e. Technical report, IC-Parc, University of London, UK, June 1997.

[97] Manas Saksena and Seongsoo Hong. An engineering approach to decomposing end-to-end delays on a distributed real-time system. In *Proceedings of the IEEE Workshop on Parallel and Distributed Real-Time Systems*, pages 244–251, 1996.

[98] Klaus Schild and Jörg Würtz. Off-line scheduling of a real-time system. In *Proceedings of the Symposium on Applied Computing*, 1998.

[99] Joachim Schrod. SGML transformations in LISP. Technical report, Computer Science Department, Technical University of Darmstadt, Kranichweg 1, D-63322 Roedermark, FR Germany, 1995.

[100] Bran Selic. Turning clockwise: Using UML in the real-time domain. *Communications of the ACM*, 42(10):46–54, October 1999.

[101] Tri-Pacific Software. PERTS: The art of modeling real-time systems, 1998. [Online] Available `http://www.tripac.com/perts/`.

[102] Marco Spuri and John A. Stankovic. How to integrate precedence constraints and shared resources in real-time scheduling. *IEEE Transactions on Computers*, 43(12):1407–1412, 1994.

[103] Jack Stankovic et al. Real-time computing: A critical enabling technology. [Online] Available `ftp://ftp.cs.umass.edu/pub/ccs/spring/critenable.ps`.

[104] John A. Stankovic. Distributed real-time computing: The next generation. *Journal of the Society of Instrument and Control Engineers of Japan*, 1992.

[105] Susan Stepney. *High Integrity Compilation—A Case Study*, chapter Introduction, page 4. Prentice Hall International (UK) Ltd., 1993.

[106] Douglas A. Stuart, Aloysius K. Mok, and Farnam Jahanian. A methodology and support tools for analysis of real-time specifications. Submitted for publication.

[107] D. Subramanian and C-S Wang. Kinematic synthesis with configuration spaces. In *Proceedings of Qualitative Reasoning 93*, pages 228–239, 1993.

[108] Tanel Tammet. Hobbit: A Scheme-to-C compiler. Technical report, Department of Computing Science, Chalmers University of Technology, University of Göteborg, Sweden, 1995. [Online] Available `http://www-swiss.ai.mit.edu/~jaffer/Hobbit.html`.

[109] The VRML Consortium Inc. The virtual reality modeling language, 1997. [Online] Available `http://www.vrml.org/`.

[110] Oliver Theel and Felix C. Gärtner. An exercise in proving convergence through transfer functions. In *Proceedings of the IEEE ICDCS Workshop on Self-Stabilizing Systems*, pages 41–47, June 5 1999.

[111] T.-S. Tia, Z. Deng, M. Shankar, M. Storch, J. Sun, L.-C. Wu, and J.W.-S. Liu. Probabilistic performance guarantee for real-time tasks with varying computation times. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, 1995.

[112] Jeffrey J. P. Tsai and Steve J. H. Yang, editors. *Monitoring and Debugging of Distributed Real-Time Systems.* IEEE Computer Society Press, 1995.

[113] André M. van Tilborg and Gary M. Koob, editors. *Foundations of Real-Time Computing: Scheduling and Resource Management*, chapter Fixed Priority Scheduling Theory for Hard Real-Time Systems, page 7. Kluwer Academic Publishers, 1991.

[114] Flavio M. Varejao, Markus P.J. Fromherz, Ana C. Bicharra Garcia, and Clarisse S. de Souza. An integrated framework for the specification and design of reprographic machines. In *Proceedings of the 13rd International Conference on Applications of Artificial Intelligence in Engineering (AIENG'98)*, July 1998.

[115] Visual Insights, Lucent Technologies. Software components, 1998. [Online] Available `http://www.visualinsights.com/components/`.

[116] Ioannis Vlahavas, Panagiotis Tsarchopoulos, and Ilias Sakellariou. *Parallel and Constraint Logic Programming: An Introduction to Logic, Parallelism and Constraints.* Kluwer Academic Publishers, 1998.

[117] Abdul Waheed and Diane T. Rover. A structured approach to instrumentation system development and evaluation. *Proceedings of Supercomputing '95*, December 4–8 1995. To appear.

[118] Brent Welch. *Practical Programming in Tcl and Tk.* Prentice Hall ISBN 0–13–182007–9, 1995.

[119] Lonnie R. Welch, Michael W. Masters, and Robert D. Harrison. Toward a 21st century shipboard computing infrastructure. Technical report, Naval Surface Warfare Center, Dahlgren, Virginia, January 1996.

[120] Lonnie R. Welch, Binoy Ravindran, Behrooz A. Shirazi, and Carl Bruggeman. Specification and modeling of dynamic, distributed real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, December 2–4 1998.

[121] David Wilner. WindView: A tool for understanding real-time embedded software through system vizualization. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, & Tools for Real-Time Systems*, pages 117–123, July 21–22 1995.

[122] Patrick H. Worley. A new PICL trace file format. Technical Report ORNL/TM-12125, Oak Ridge National Laboratory, Mathematical Sciences Section, P.O. Box 2008, Bldg. 6012, Oak Ridge, TN 37831-6367, September 1992.

[123] Jerry Yan. Performance tuning with AIMS—an automated instrumentation and monitoring system for multicomputers. In *Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*, Hawaii, January 1994.

[124] Darko Zupanič. Optimal solution for a textile production unit. In *Proceedings of the Second International Conference on the Practical Application of Constraint Technology*, April 1996.

MICHIGAN STATE UNIVERSITY LIBRARIES
3 1293 02092 5198