

FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

**Knowledge Specification for
Automatic Performance Analysis
APART Technical Report
Revised Version**

*Thomas Fahringer, Michael Gerndt, Bernd Mohr,
Felix Wolf, Graham Riley, Jesper Larsson Träff*

FZJ-ZAM-IB-2001-08

August 2001

(letzte Änderung: 22.08.2001)

Knowledge Specification for Automatic Performance Analysis
APART Technical Report
Revised Version ¹

<http://www.fz-juelich.de/apart>

Workpackage 2
Identification and Formalization of Knowledge

Thomas Fahringer
Institute for Software Science
University of Vienna
tf@par.univie.ac.at

Michael Gerndt
Institut für Informatik, LRR
Technische Universität München
gerndt@in.tum.de

Bernd Mohr, Felix Wolf
Central Institute for Applied Mathematics
Research Centre Juelich
b.mohr@fz-juelich.de
f.wolf@fz-juelich.de

Graham Riley
Department of Computer Science
University of Manchester
griley@cs.man.ac.uk

Jesper Larsson Träff
C&C Research Laboratories
NEC Europe Ltd.
traff@ccrl-nece.technopark.gmd.de

August 22, 2001

¹The ESPRIT IV Working Group on *Automatic Performance Analysis: Resources and Tools* is funded under Contract No. 29488

Abstract

The lack of a useful and accurate software infrastructure for measuring, modeling, and analyzing the performance of a wide variety of programming paradigms and architecture platforms is a critical issue for performance-oriented program development. Commonly, a cyclic process is employed to tune the performance of programs which includes the gathering of performance data through measurement and prediction and the analysis of the data collected on-the-fly or during a postmortem session to yield summary statistics and histories of program behavior. Usually, this process also involves comparison of the performance data with that of previous program versions. So far most approaches require the programmer to control this tedious, time-consuming, and error-prone process which is typically driven by some informal hypotheses about potential performance problems. Moreover, many tools are platform and language dependent and cannot correlate performance data gathered at lower levels (for example, from hardware counters) with higher-level programming paradigms. Further, they tend to focus only on specific program and machine behavior, and do not provide sufficient support to infer important performance properties.

In this report we describe a novel approach to the formalization of performance bottlenecks and the data required to detect them with the aim of supporting automatic performance analysis for a large variety of programming paradigms and architectures. We present the APART Specification Language (ASL) developed as part of the APART Esprit IV Working Group on *Automatic Performance Analysis: Resources and Tools*. This language allows the description of performance-related data through the provision of an object-oriented specification model and supports definition of performance properties in a novel formal notation. Performance-related data can either be static (gathered at compile-time, e.g. code regions, control and data flow information, predicted performance data, etc.) or dynamic (gathered at run-time, e.g. timing events, performance summaries, etc.) and is used as a basis for describing performance properties. A performance property (e.g. load imbalance, communication, cache misses, etc.) characterizes a specific type of performance behavior which may be present in a program. Checks for which properties are present in (the execution of) a program are given by a set of conditions defined over the performance-related data. Conditions have an associated confidence level which indicates the degree of certainty in the diagnosis of the presence of the performance property. Performance properties also have an associated severity measure (usually an expression), the magnitude of which specifies the importance of the property in terms of its contribution to limiting the performance of the program. The severity can be used to focus effort on the important performance issues during the (manual or automatic) performance tuning process.

In the second year of APART, ASL has been extended by new concepts that include templates and metaproperties. Templates are used to describe similar performance properties with a generic representation. Commonly only the cost function in a property is used to parameterize a template which has the advantage of very compact and easy to understand performance properties. Moreover, in order to support the specification of performance properties based on other performance properties, we invented metaproperties. By incorporating this concept we can formulate properties based on performance-related data and components (condition, confidence, and severity) of existing performance properties. ASL has been further extended by patterns to describe compound events that indicated the existence of performance properties which can only be detected in event traces.

Our approach is very general and can be efficiently employed to describe many performance properties for a large variety of programming paradigms and architectures. We illustrate our approach by applying it to some of the most important programming paradigms for performance-oriented scientific computing including MPI, OpenMP, and HPF.

Contents

1	Introduction	1
1.1	Overview	1
1.2	Overall Design	2
1.3	Related Work	3
2	Performance Related Data Specification	7
2.1	Specification Language	7
2.2	Standard Class Library	7
2.3	Paradigm Specific Data Models	9
2.3.1	MPI Classes	9
2.3.2	HPF Classes	12
2.3.3	OpenMP Classes	14
3	Performance Property Specification	19
3.1	Specification Language	19
3.2	Property Templates and Metaproperties	23
3.3	Properties Based on Pattern Matching in Traces	24
3.3.1	Abstractions	24
3.3.2	Example: MPI Related Abstractions	25
3.3.3	Specifying Compound Events	25
3.3.4	Compound Events in ASL	26
3.3.5	Pattern Matches	26
3.3.6	Example: MPI Related Compound Events	27
3.3.7	Using Patterns in Property Definitions	28
3.3.8	Summary	28
3.4	Paradigm Related Property Specification	28
3.4.1	MPI	28
3.4.1.1	MPI global definitions	30
3.4.1.2	TotalCost	30
3.4.1.3	CommCost, SyncCost, IoCost	31
3.4.1.4	CacheCost	31
3.4.1.5	DominatingCommFunction	32
3.4.1.6	DominatingCommCall	32
3.4.1.7	FrequentCommunication	33
3.4.1.8	BigMessages	33
3.4.1.9	LateSender	33
3.4.1.10	LateReceiver	34
3.4.1.11	UnevenMpDistribution	34

3.4.1.12	LoadImbalanceAtBarrier	34
3.4.1.13	SlowSlaves	35
3.4.1.14	OverloadedMaster	35
3.4.1.15	WrongOrder	36
3.4.1.16	FrequentCommInProgress	37
3.4.1.17	FrequentCommOnAllProcesses	37
3.4.1.18	CacheMissesPerInstance	38
3.4.1.19	Property for all instances of a region	39
3.4.1.20	Property template for some instances	40
3.4.1.21	FrequentCommSomeProcesses	41
3.4.2	HPF	41
3.4.2.1	HPF global definitions and templates	41
3.4.2.2	template costs	42
3.4.2.3	costs	42
3.4.2.4	communication_costs	42
3.4.2.5	forall_synchronization_costs	43
3.4.2.6	io_costs	43
3.4.2.7	parallel_organization_costs	43
3.4.2.8	redist_costs	44
3.4.2.9	serialization_costs	45
3.4.2.10	uneven_work_distribution	46
3.4.2.11	inspector_costs	46
3.4.3	OpenMP	46
3.4.3.1	OpenMP global definitions	47
3.4.3.2	Template costs	47
3.4.3.3	template rel_costs	48
3.4.3.4	costs	48
3.4.3.5	measurable_costs	48
3.4.3.6	unmeasurable_costs	49
3.4.3.7	non_parallelized_code	49
3.4.3.8	synchronization	49
3.4.3.9	irregular_sync_across_instances	50
3.4.3.10	load_imbalance	50
3.4.3.11	remote_accesses	50
3.4.3.12	remote_access_to_variable	51
3.4.3.13	multiple_transfer_of_same_data	51
3.4.3.14	wrong_page_distribution_for_variable	52
3.4.3.15	parallel_organization	52
4	Conclusions and Future Work	54
A	Unified Modeling Language Class Diagrams	57
B	APART Base Class Library	59
C	MPI Property Specification	61
D	HPF Property Specification	71
E	OpenMP Property Specification	77

Chapter 1

Introduction

1.1 Overview

Performance-oriented program development can be a daunting task. In order to achieve high or at least respectable performance on today's multiprocessor systems, careful attention to a plethora of system and programming paradigm details is required. Commonly, programmers go through many costly and time consuming cycles of experimentation involving the gathering and analysis (a-priori and post-mortem) of performance data, detection of performance problems, and code refinements. Clearly, the programmer must be intimately familiar with many aspects related to this experimental process. Although there exist a large number of tools to assist the programmer in performance experimentation, the responsibility for taking the majority of strategic decisions still lies with the programmer. It is particularly distressing that many performance tools remain platform and language dependent, cannot correlate performance data gathered at a lower level with higher-level programming paradigms, focus only on specific program and machine behavior, and do not provide sufficient support to infer important performance properties.

In this report we describe a novel approach to the task of formalizing the description of performance bottlenecks and the data required to detect them with the aim of supporting automatic performance analysis for a large variety of programming paradigms and architectures. This research has been performed as part of APART Esprit IV Working Group on *Automatic Performance Analysis: Resources and Tools* (APART, <http://www.fz-juelich.de/apart>).

In the remainder of this report we use the following terminology:

Performance-Related Data: Performance-related data defines information that can be used to describe performance properties of a program. There are two classes of performance related data. First, static data specifies information that can be determined without executing a program on a target machine. Static data is useful in order to specify dynamic performance related data and to formalize performance properties. Examples include code versions, program regions, source files, control and data flow information, loop scheduling information, predicted performance data, and information on the programming paradigm (e.g. master-slave, divide-and-conquer, bulk-synchronous, etc.). Second, dynamic performance related data describes the dynamic behavior of a program during execution on a target machine. This includes timing events, performance summaries and metrics, and communication patterns that are statically undetectable, etc.

Performance Property: A performance property (e.g. load imbalance, communication, cache misses, redundant computations, etc.) characterizes a specific performance behavior of a program and can be checked by a set of *conditions*. Conditions are associated with a *confidence value* (between 0 and 1) indicating the degree of confidence about the existence of a performance property. In addition, for every performance property a *severity measure* is provided the magnitude of which specifies the importance of the property. The severity can be used to focus effort on the important performance issues during the (manual or automatic) performance tuning process. Performance properties, confidence and severity are defined over performance-related data.

Performance Problem: A performance property is a performance problem, iff its severity is greater than a user- or tool-defined threshold.

Performance Bottleneck: A program has a unique performance bottleneck which is its most severe performance property. If this bottleneck is not a performance problem, then the program's performance is acceptable and does not need any further tuning.

For example, during performance analysis, a specific code region may be examined to determine the existence of a performance property denoted *communication*. The condition for this property holds if any process executing the region invokes communication (that is, if communication time for the region is greater than zero). The confidence value is 1, since measured communication time represents a proof for the presence of this property. The severity of the property may be calculated as the percentage of the communication time in the region relative to the execution time of the entire program. If the severity is above a user- or tool-defined threshold, then the communication performance property defines a performance problem. If this performance problem is the most severe of all the performance problems in the program, then it is the performance bottleneck. Commonly, a programmer may try to eliminate, or at least to alleviate, this bottleneck before examining any other performance problems.

This report introduces the APART Specification Language (ASL) which allows the description of performance-related data through the provision of an object-oriented specification model and which supports the definition of performance properties in a novel formal notation. Our object-oriented specification model is used to declare – without the need to compute – performance information. It is similar to Java, uses only single inheritance and does not require methods. A novel syntax has been introduced to specify performance properties. Moreover, ASL has been extended in the second year of APART by new concepts that include templates and metaproperties. Templates are used to describe similar performance properties with a generic representation. Commonly only the cost function in a property is used to parameterize a template which has the advantage of very compact and easy to understand performance properties. Moreover, in order to support the specification of performance properties based on other performance properties, we invented metaproperties. By incorporating this concept we can formulate properties based on performance-related data and components (condition, confidence, and severity) of existing performance properties. ASL has been further upgraded by the ability to describe patterns. Compound events can be specified which indicate the existence of performance properties that can only be found in event traces.

The organization of this report is as follows. We continue this chapter with the presentation of an overall design of an automatic performance analysis environment that incorporates the specification of performance properties. Related work is discussed in Section 1.3. In the next chapter, Chapter 2, we describe our object oriented specification model for performance-related data by using UML (unified modeling language) class diagrams. We apply this model to some of the most important programming paradigms in current use including MPI, OpenMP, and HPF. In Chapter 3 we describe our syntax for performance property specification including the syntax for templates, metaproperties, and patterns. Again, paradigm related property specifications are presented for MPI, OpenMP, and HPF. Conclusions and Future work are discussed in Chapter 4. In the Appendix we give an overview of UML, and summarize performance related data and performance properties.

1.2 Overall Design

Performance property specification as described in this report can be considered as part of a possible design for an automatic performance analysis environment. This environment comprises three components (see Figure 1.1):

Performance Property Specification defines information about performance properties for the current programming paradigm and machine, in combination with proof conditions and severity data.

Performance Process Specification reflects the knowledge applied in tuning the performance of programs including, for example, how many hypotheses about performance problems are evaluated. This evaluation can be based on stepwise refinement, i.e. the process specification determines which hypotheses are evaluated first before more precise hypotheses are examined. For example, it may be useful to prove that message passing is significant in a subroutine before examining individual MPI calls. More detailed analysis may require considerably more performance-related data.

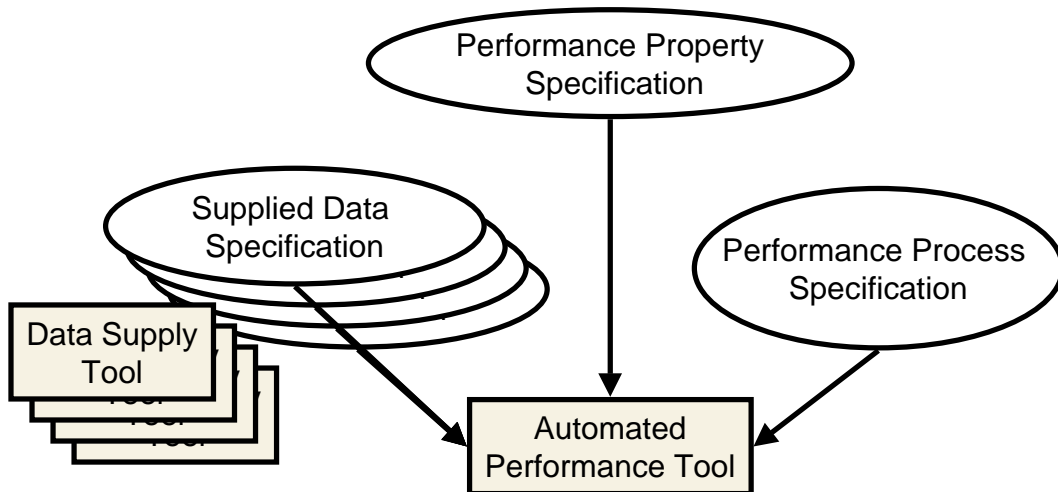


Figure 1.1: Design of an integrated automatic performance analysis environment.

Supplied Data Specification describes, for a particular tool, which of the performance-related data required for performance property specification can be obtained from that tool. Moreover, query commands to access this data can be indicated. Examples for such tools are PARADYN [MCCHI 95], SCALA [FaScPa 99], TAU [MoBrMa 94], and VAMPIR [NaArWeHoSo 96], etc. Based on the supplied data specifications, an automated performance analysis environment can use existing tools to access relevant data in the search for performance problems and bottlenecks.

An integrated system combining all three components should substantially alleviate the task of re-targeting performance tools to new architectures and programming paradigms, facilitate the development of new performance tools and also enable the enhancement of existing tools by providing access to a wealth of performance information and analysis capabilities.

1.3 Related Work

The specification of performance problems as presented in this report is a novel approach. Relatively few existing performance analysis tools and related projects apply specification languages in the context of performance analysis.

The most well-known automatic performance analysis tool is Paradyne [MCCHI 95]. Paradyne performs an automatic online analysis using dynamic instrumentation for monitoring. The *Performance Consultant* (PC) searches for performance bottlenecks according to the W^3 Search Model: each potential bottleneck is expressed in terms of *why* there is a problem, *where* in the application the problem is found (i.e., a focus), and *when* the problem occurs, i.e., in which phase(s) of the execution.

Hypotheses are conditions of the form $metric, focus > threshold$, where metric is a time-varying function that characterizes some aspect of a parallel program performance, such as CPU utilization or number of synchronization operations, and focus is the program location where the metric is measured. While metrics can be defined via the *Metric Description Language* (MDL) [Paradyne 98] the set of bottleneck hypotheses is currently predefined. It includes *CPUbound*, *Excessive Sync Waiting Time*, *ExcessiveIOBlockingTime*, and *TooManySmallIOOps*. The metric description specifies among other things the measurement basis (counter or timer), the aggregate operator (average, sum, minimum, or maximum) and the instrumentation actions.

The Autopilot [Re 98] project integrates dynamic performance instrumentation and on-the-fly performance data reduction with configurable, malleable resource management algorithms, and a real-time adaptive control mechanism that automatically chooses and configures resource management algorithms based on application request patterns and observed system performance. The goal is the creation of an infrastructure for building resilient, distributed and parallel applications. The Autopilot infrastructure

is built on the lessons and software from the successful Pablo performance environment [Re 93] and the experiences with adaptive file systems.

A rule-based specification of performance bottlenecks and the analysis process was developed within the context of the SVM-Fortran project [BeGeKr 96]. The performance analysis tool OPAL [GeKrOz 95] supports post-mortem analysis and used in combination with the monitoring system SAM, it realizes an incremental performance analysis process. New measurements can be requested based on the user's insight in the performance behavior. The measurements are executed during the next program run without having to recompile the code. Based on experience gained in applying this tool to real applications, a rule-based design for the automation of OPAL was developed [GeKr 97]. The rule base consists of a defined set of parameterized hypothesis with proof rules and refinement rules. The proof rules determine whether a hypothesis is valid based on the measured performance data. The refinement rules specify which new hypotheses are generated from a proven hypothesis. Therefore, the refinement rules specify the analysis process while the proof rules specify the knowledge about bottlenecks.

Another approach is to define a performance bottleneck as an event pattern or compound event which may occur during execution of a parallel program. Such patterns have to be detected in an event trace provided by tools like PAT [GaMo 98] after program termination. The compound event is built from primitive events such as those associated with entering a program region or sending a message.

EDL [BaWi 83] allows the definition of compound events based on extended regular expressions. Primitive events are clustered to higher-level events by certain formation operators. Relational expressions over the attributes of the constituent events place additional constraints on valid event sequences obtained from the regular expression. Abstraction mechanisms allow the reuse of already defined event patterns to form custom hierarchies of events.

EARL [WoMo 98] describes event patterns in a more procedural fashion as scripts in a high-level event-trace-analysis language, which is implemented as an extension of common scripting languages like Tcl, Perl or Python. Frequently used higher-level events like region instances or message transfers are represented by links among their constituent events, which can be easily traversed by a script. In addition, EARL supports navigation through function-call stacks and message queues of a chosen execution state of the parallel program, enabling compact specification and efficient detection of the requested compound event. Besides these tools and specification concepts for performance problems, a few others tools have been developed supporting automatic performance analysis.

KAPPA-PI [EsMaLu 98] is an automatic performance analyzer for PVM-programs developed at the Universitat Autònoma de Barcelona. It is a post-execution tool, implemented in PERL, that evaluates traces generated by the Tape/PVM monitoring library or by the VAMPIR MPI trace library. Based on a predefined list of performance bottlenecks, it searches for performance problems and their causes. In addition to trace data, it analyzes the source code using pattern matching.

One additional design, POIROT, was published by Robert Helm and Allen Malony [HeMa 95]. The design is based on the concept of heuristic classification. The main properties of a program run are extracted from trace data by a process called abstraction. These properties are matched against a database of possible performance bottlenecks and the selected bottleneck is refined to fit additional properties of the program run. Performance data are gathered via an environment interface that makes POIROT independent of intricate details of the programming environment, e.g., how to instrument a program.

FINESSE [Mu 99, Mu 00] is a prototype environment designed to support rapid development of parallel programs for single-address-space computers by both expert and non-expert programmers. The environment provides semi-automatic support for systematic, feedback-guided identification and reduction of the various classes of overhead associated with parallel execution. FINESSE automatically identifies code regions with significant overhead, classifies and quantifies this overhead and ranks regions according to their execution time. FINESSE also suggests possible improvements which should lead to improved implementations.

P^3T [Fa 95, Fa 96] is a static performance estimator for data parallel programs which guides the selection of efficient data distribution strategies and profitable code transformations. This tool tries to answer three fundamental questions: 1. What performance bottlenecks exist? 2. Where are these these performance bottlenecks in the input program? 3. What must be done in order to gain performance? P^3T automatically computes a variety of performance parameters including work distribution, number of data transfers, amount of data transferred, transfer times, network contention, number of cache misses, and computation times. Through a graphical user interface the programmer can optionally specify for each of

these parameters a specific threshold which implies a bottleneck. The default option is that P^3T visualizes every computed performance parameter relative to the worst-case value found in the entire program. Color-coded performance visualization directly guides the user to all bottlenecks of a program found by P^3T . A list of code transformations is suggested to eliminate or alleviate each specific performance bottleneck.

Chapter 2

Performance Related Data Specification

A necessary prerequisite for automatic performance analysis is the availability of sufficient information to allow the examination of the performance behavior of an application on a given architecture.

Performance-related data can be obtained either statically, by using, for instance, program analysis tools, or dynamically with the aid of monitoring tools. Dynamic information collection requires the application to be executed on a target machine, whereas accessing static information does not. Examples of static data include, code versions, program regions, source files, control and data flow information, predicted performance data, and information on the programming paradigm (e.g. master-slave, divide-and-conquer, bulk-synchronous, etc.). Timing events, performance summaries and metrics, and communication patterns that are statically undetectable represent dynamic information.

A key issue in our work on automatic performance analysis is to find a well-suited representation of static and dynamic information such that we can exploit performance-related data obtained from many different sources (e.g. performance tools, program analysis tools, databases, user provided data, etc.). Moreover, it is of paramount importance to be able to relate performance information derived from different sources. These abilities will alleviate current difficulties encountered in the specification of performance properties and support the task of automatically searching for performance bottlenecks.

In this section, we introduce a specification language for describing performance-related data. We first present class libraries, in UML notation (see Appendix A), for performance-related data that is programming paradigm independent. Thereafter, specialized class libraries and examples are presented for the following programming paradigms: MPI, HPF, and OpenMP.

2.1 Specification Language

Figure 2.1 shows the syntax for specifying both static and dynamic performance-related data in Backus Naur form. Performance-related data is described by a set of classes following an object-oriented style with single-inheritance. Among others, class members can be of type `FLOAT` (e.g., for timing measurements), `BOOLEAN` (e.g., for flags), `INT` (e.g., for counting events), `STRING` (e.g., for naming applications or files), `DATETIME` (time at which some event occurs), and *reference* (e.g. for named enum types and class names). An identifier is described by *ident*. *SETOF* and *ENUM* enable set and enumeration notations.

Syntax variables in the syntax diagrams ending with “-list” identify a colon-separated list of one or more elements. For example, *string-list* represents a list of character constants such as “DO, FORALL, WHILE”.

2.2 Standard Class Library

In this section we describe a library of classes that represent static and dynamic information for performance property analysis. We distinguish two sets of classes. First, the set of base classes which

```

class-def    is CLASS ident [EXTENDS ident ] '{ member-def * }' ';'
member-def  is  type ident ';'
type        is  FLOAT
            or  BOOLEAN
            or  INT
            or  STRING
            or  DATETIME
            or  set-type
            or  enum-type
            or  reference
set-type    is  SETOF type
enum-type   is  ENUM ident '{ string-list }'

```

Figure 2.1: Syntax for describing performance-related data.

is independent of any programming paradigm, and secondly, programming paradigm-dependent classes comprising shared memory, data parallel, and message passing paradigms. The programming paradigm dependent classes are shown for HPF (High Performance Fortran [HPF 93]), OpenMP (shared memory paradigm [DaMe 98]), and MPI (Message Passing Interface [SnOtHUVaDo 98]) which are implementations of the shared memory, data parallel and message passing paradigm, respectively.

Note that we expect that most data models described with this language will have a similar overall structure. This similarity was captured in the design of the base classes. Future data models can build specialized classes in form of subclasses.

The data models presented for MPI, HPF, and OpenMP are intended to be examples. They cover typical program, machine, and performance data available in programming environments for those paradigms. Since they have been defined without a concrete programming environment with performance tools providing performance-related data in mind, it is possible, that those models do not fit the readers favorite programming environment.

Note that we do not claim that our class library is complete. Our classes, and in particular their attributes, can be extended to include other static and dynamic information to model relevant performance aspects of a large variety of programming paradigms.

Figure 2.2 shows the UML representation of the base classes which are programming paradigm independent. Initially, there is an application for which performance analysis has to be done. Every application has a name and may possibly have a number of implementations, each with a unique version number. Versions may differ with respect to their source files and experiments. Every source file (the contents of which are stored in a generic string) has one or several static code regions each of which is uniquely specified by *start_pos* (position where region begins in the source file) and *end_pos* (position where region ends in the source file). A position in a region is defined by a line and column number with respect to the given source file. Among other things, regions can be continuous sequences (e.g., entire programs or loops) or parts (e.g., expressions or even data references) of source-code lines. Regions can have sub-regions. For instance, we can represent all procedures or loops of a region by using its *sub_regions* attribute.

Experiments – denoting the second attribute of a version – are described by the time (*start_time*) when the experiment started and the number of processors (*nr_processors*) that were available to execute the version. Furthermore, an experiment is also associated with a static description of the machine (e.g. number of processors available) that is used for the experiment. Every experiment includes also dynamic data, i.e. a set of region summaries (*profile*) and a set of events (*trace*). The class *RegionSummary* describes performance information across all processes employed for the experiment. Region summaries are associated with the appropriate region. The class *Events* represents information about individual events occurring at runtime, such as sending a message to another process. Each event has a *time_stamp* attribute determining when the event occurred and a *process* attribute determining in which process the event occurred.

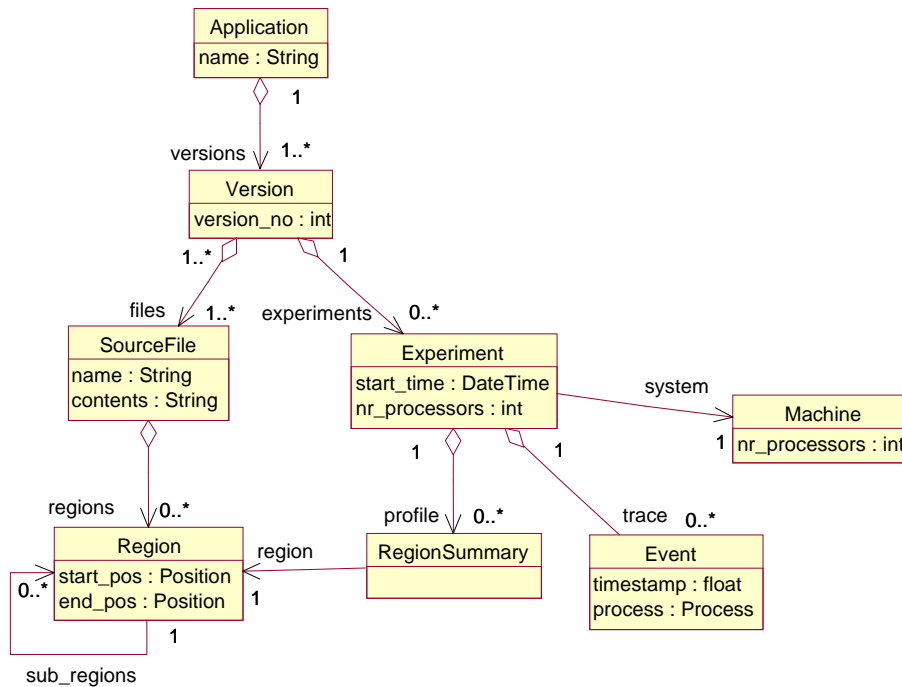


Figure 2.2: Base classes of performance-related data models.

2.3 Paradigm Specific Data Models

2.3.1 MPI Classes

In this section we describe static and dynamic information for MPI which is an implementation of the message passing paradigm. Figure 2.3 outlines the classes for describing static information. A new class *MPIApplication* is derived from *Application*. It provides an additional attribute *Paradigm* which relates to the paradigm implemented, e.g. master-slave, divide-and-conquer, and bulk-synchronous.

Regions in an MPI program are modeled by two classes *Function* and *FuncCall*. *Function* has two attribute specifying the number of call sites in the function and the function name. *FuncCall* has three attributes identifying the called function, the caller, and the role in the paradigm, e.g. master/slave send/receive operation.

MPIFunction is a more specialized class derived from *Function*. Special types of MPI functions are represented by *MPICommFunc*, *MPISyncFunc*, and *MPIioFunc*. The set of MPI communication functions is further subdivided into *MPIp2pFunc* and *MPICollFunc*. Although *MPIBarrier* is also a collective function it is represented by *MPISyncFunc* and not by *MPICollFunc*.

MPIp2pFunc provides more specific information about the point-to-point communication. An attribute *type* determines whether the underlying communication is based on a send or receive operation. The communication mode (e.g. buffered, synchronous, ready) is denoted by attribute *mode*. Blocking or nonblocking communication can be defined by attribute *semantics*.

MPICollFunc refers to various collective operations. This class comprises an attribute *type* for the type of collective operation (e.g. reduction or broadcast). The precise semantics of the above mentioned MPI communication modes and types can be found in [SnOtHUVaDo 98].

Figures 2.4 and 2.5, respectively, describe summary and event information which reflects the dynamic behavior of MPI. Class *MPISummary* in Figure 2.4 extends the base class *RegionSummary* (see Figure 2.2) and determines profile information. An object of this class can either determine information for a single process or information for all processes. In the latter case, the *Proc* attribute is set to ALLPROCESSES. The information provided in *MPISummary* is:

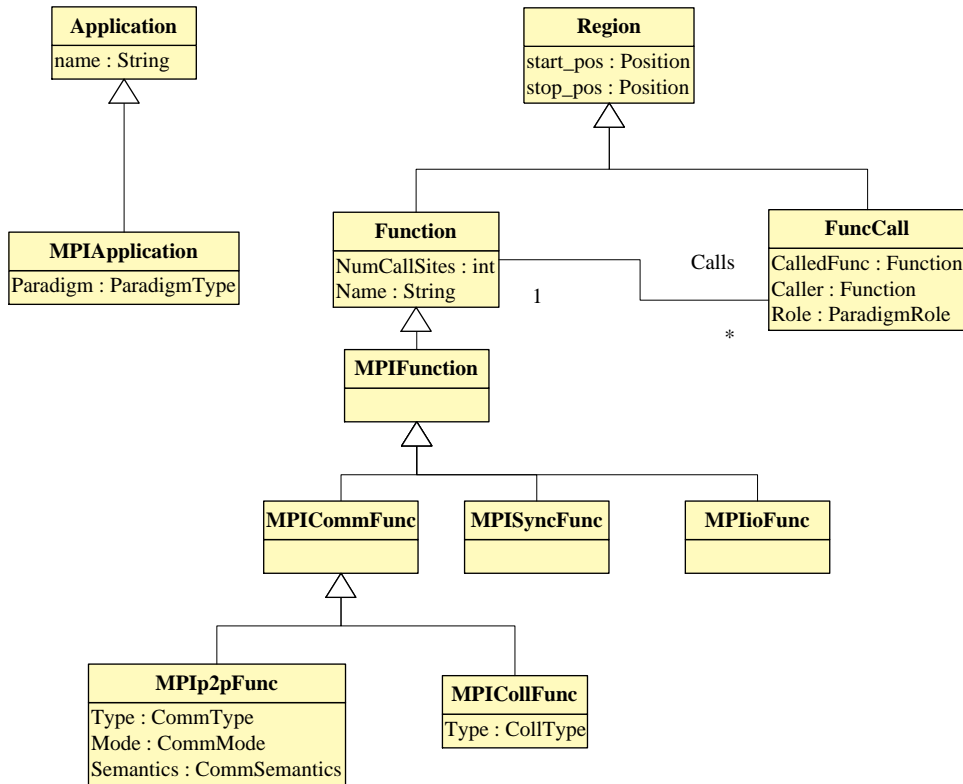


Figure 2.3: Regions in the MPI data model.

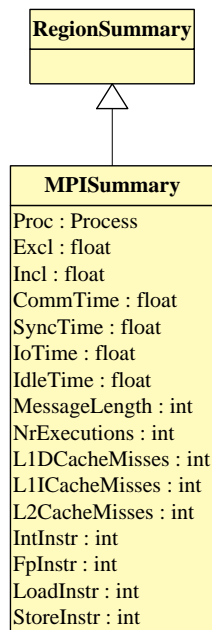


Figure 2.4: Summaries in the MPI data model

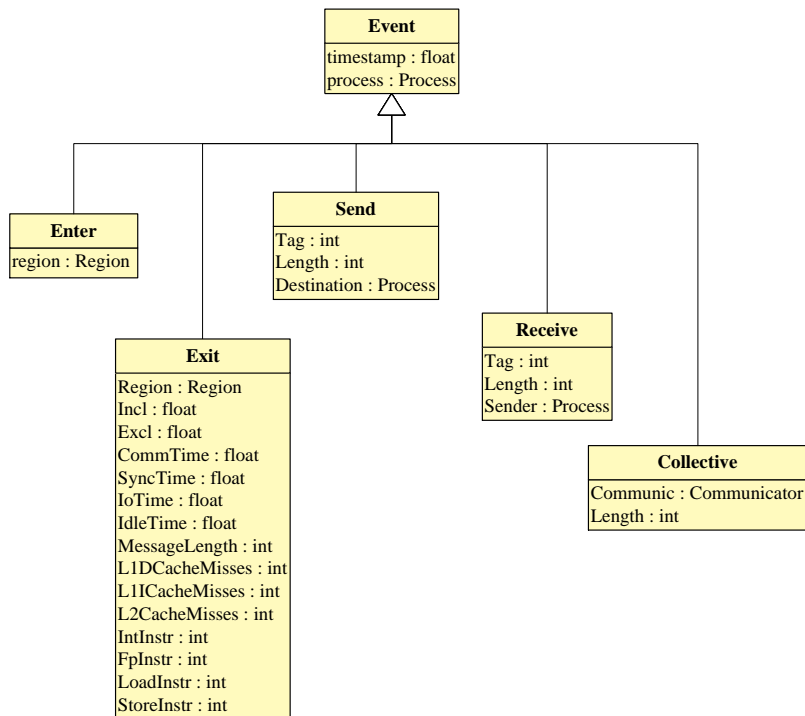


Figure 2.5: Events in the MPI data model.

- *Excl*: execution time excluding time spent in called routines
- *Incl*: execution time
- *CommTime*: communication time
- *SyncTime*: barrier synchronization time
- *IoTime*: input/output time
- *IdleTime*: idle time
- *MessageLength*: sum of the length of all messages sent
- *NrExecutions*: number of executions of a given region
- *L1DCacheMisses*: number of L1 data cache misses
- *L1ICacheMisses*: number of L1 instruction cache misses
- *L2CacheMisses*: number of L2 cache misses
- *IntInstr*: number of executed integer operations
- *FpInstr*: number of executed floating point operations
- *LoadInstr*: number of executed load operations
- *StoreInstr*: number of executed store operations

Figure 2.5 describes dynamic events for MPI programs as they are generated by the VAMPIRtrace library provided by Pallas. Five specialized subclasses of the *event* base class are introduced for MPI: *Enter*, *Exit*, *Send*, *Receive* and *Collective*.

The *Enter* and *Exit* events are generated when the execution of a region, e.g. function or loop, is started or terminated, respectively. The events provide a reference to that region. The exit event provides the same performance data as the region summary information. Therefore, individual executions or instances of regions can be checked for the same behavior. If a send or receive operation is executed, a *send* or *receive* event is generated that specifies additional information, i.e. the tag and length of the message as well as the sender or destination respectively. An additional event is also generated during the execution of collective operations. This event specifies the communicator which identifies the partners in the collective operation, and determines the sum of the lengths of all messages sent by the executing process.

The event data model for MPI is used for the specification of pattern of events in Section 3.3.

2.3.2 HPF Classes

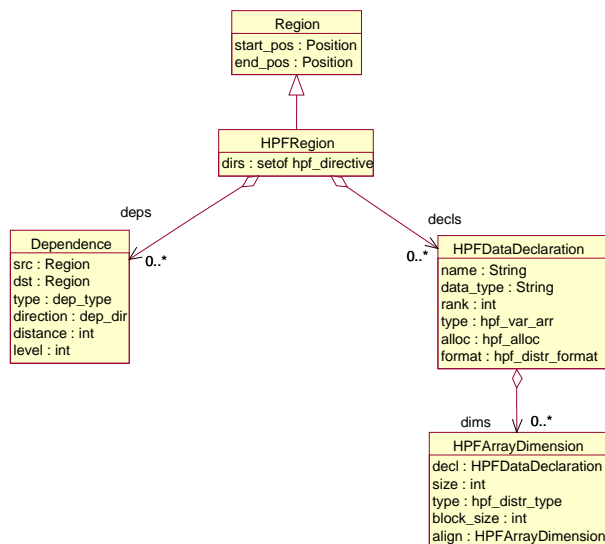


Figure 2.6: Static performance-related information for *HPFRegions*.

In this section we describe the class libraries for HPF which is an implementation of the data parallel programming paradigm. Class *HPFRegion* extends *Region* (see Figure 2.2) and comprises the following attributes (Figure 2.7) representing static performance-related information:

- *dirs* describes HPF directives such as PROCESSORS, DISTRIBUTED, ALIGN, RESHAPE, INDEPENDENT, etc.
- *deps* specifies data dependences implied by code regions.
- *decls* specifies HPF data declarations for scalars and arrays. Attribute *alloc* denotes whether data has been declared DYNAMIC or STATIC. For arrays there is additional data provided for every dimension including size of dimension, distribution and alignment information.

Figure 2.7 displays several subclasses which extend *HPFRegion*:

- *HPFProcedure*:
- *HPFLoop*:
- *HPFIfBlock*:

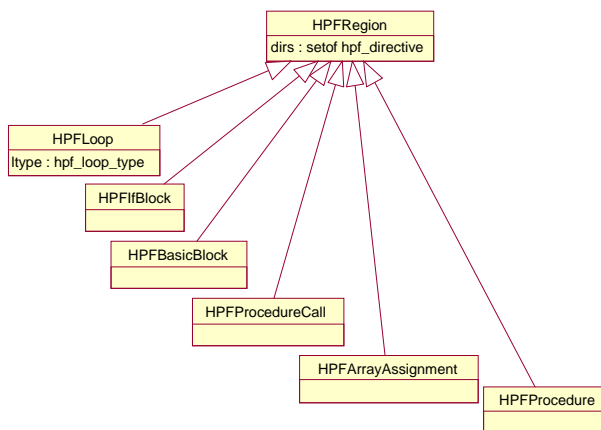


Figure 2.7: Subclasses of *HPFRegion*.

- *HPFBasicBlock*:
- *HPFProcedureCall*:
- *HPFArrayAssignment*:

Class *HPFLoop* can be further specified by attribute *ltype* (e.g. DO, INDEPENDENT, and FORALL).

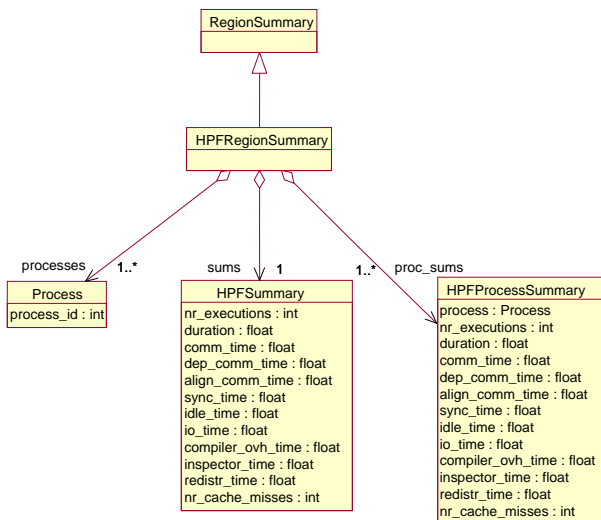


Figure 2.8: Dynamic performance-related information (summaries) in the HPF data model.

Dynamic performance-related data is described by class *HPFRegionSummary* in Figure 2.8 with the following attributes:

- *processes* specifies the set of processes executing a region
- *sums* reflects performance summary information across all processes executing the region
- *proc_sums* indicates performance summary information for a region with respect to individual processes.

Class *HPFSummary* contains several performance attributes which are average values across all processes with respect to a specific region:

- *nr_executions*: number of times the region has been executed
- *duration*: time spent in executing the region
- *comm_time*: communication time
- *dep_comm_time*: communication time caused by data dependences
- *align_comm_time*: communication time caused by data alignment
- *comm_time*: communication time
- *sync_time*: synchronization time
- *idle_time*: idle time
- *io_time*: input/output time
- *compiler_ovh_time*: compiler overhead time
- *inspector_time*: time spent in inspector/executor phase (compiler inserted code to handle irregular problems)
- *redistr_time*: time spent in redistribution of arrays
- *nr_cache_misses*: number of cache misses.

Class *HPFProcessSummary* contains all attributes of class *HPFSummary* restricted to average values for a specific process.

2.3.3 OpenMP Classes

Figure 2.9 shows the classes that model static information for OpenMP programs. Class *SmRegion* is a subclass of *Region* (see Figure 2.2) and contains an attribute with data dependence information about the modeled region. *SmRegion* is then further refined by two subclasses *ParallelRegion* and *SequentialRegion* which, respectively, describe parallel and sequential regions. Note that in OpenMP a master thread is responsible for the execution of sequential regions and is also responsible for engaging other threads in the execution of parallel regions. Typically, for efficiency reasons, threads will sit in some form of idle pool while the master executes sequential regions, rather than threads being continually created and destroyed by the master in an explicit fork/join model. Currently we have defined four sequential regions including:

- *Function*
- *IfBlock*
- *BasicBlock*, and
- *FunctionCall*

which respectively, refer to a function, IF-THEN-ELSE construct, basic block (single-entry-exit code regions – [AhSeUl 88]), and a function call in the OpenMP program. When a master thread encounters a parallel region it releases a set of threads from an idle pool (they typically are waiting on a region entry barrier) in order to execute the region in parallel. Parallel regions include a boolean variable *no_wait_exit* which denotes whether or not the region is terminated by an explicit exit barrier operation. A specific execution of a region corresponds to a region instance. The following parallel regions are modeled:

- *PRegion* correspond to OpenMP’s parallel region which is a block of code whose instances are executed by all threads in a replicated mode.
- *PSection* refer to OpenMP’s parallel sections each of which is executed by a specific thread in parallel.

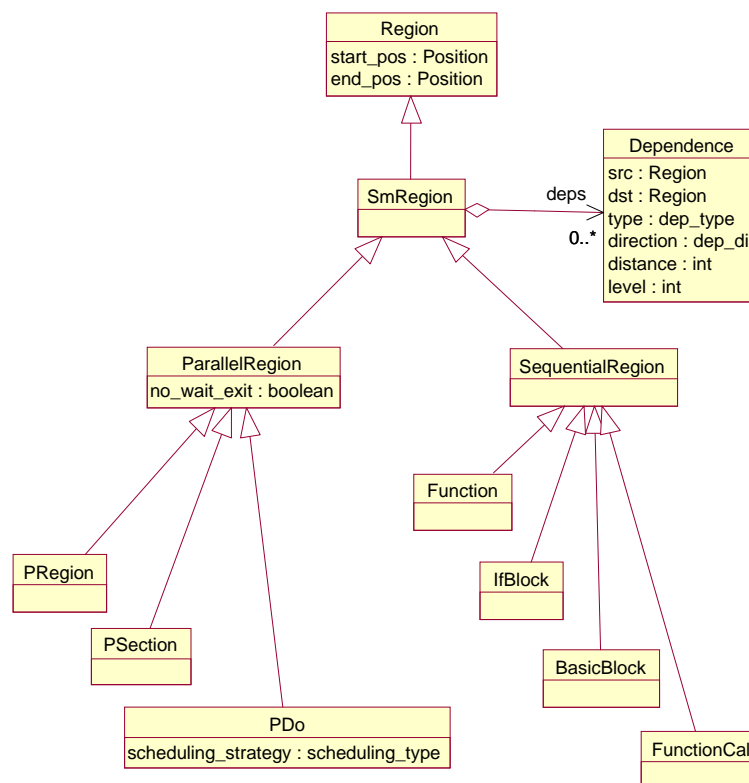


Figure 2.9: OpenMP classes for static information

- *PDo* relates to OpenMP’s parallel DO construct whose iterations are executed by a set of threads in parallel. The DO loop’s iterations can be distributed in various ways including STATIC(CHUNK), DYNAMIC(CHUNK), and GUIDED(CHUNK) onto the set of threads (as defined in the OpenMP standard). The distribution is specified in class *PDo*. STATIC(CHUNK) distribution means that the set of iterations are consecutively distributed onto the threads in blocks of CHUNK size (resulting in block and cyclic distributions). DYNAMIC(CHUNK) distribution implies that iterations are distributed in blocks of CHUNK size to threads on a first-come-first-served basis. GUIDED(CHUNK) means that blocks of exponentially decreasing size are assigned on a first-come-first-served basis. The size of the smallest block is determined by CHUNK size.

Figure 2.10 shows the OpenMP class library for dynamic information. Class *SmRegionSummary* extends class *RegionSummary* (see Figure 2.2) and comprises three attributes: *nr_executions* specifies the number of times a region has been executed by the master thread, *sums* describes summary information across all region instances, and *instance_sums* relates to summary information for a specific region instance. The attributes of class *SmSums* include:

- *duration*: time needed to execute region by master thread
- *non_parallelized_code*: time needed to execute non-parallelized code
- *seq_fraction*: $\frac{\text{non_parallelized_code}}{\text{duration}}$
- *nr_remote_accesses*: number of accesses to remote memory by load and store operations in ccNUMA machines [CuSiGu 99]
- *scheduling*: time needed for scheduling operations (e.g. scheduling of threads)
- *additional_calc*: time needed for additional computations in parallelized code (e.g. to enforce a specific distribution of loop iterations) or for additional computations (e.g. where it is cheaper for all threads to compute a value rather than communicate it, possibly with synchronization costs)

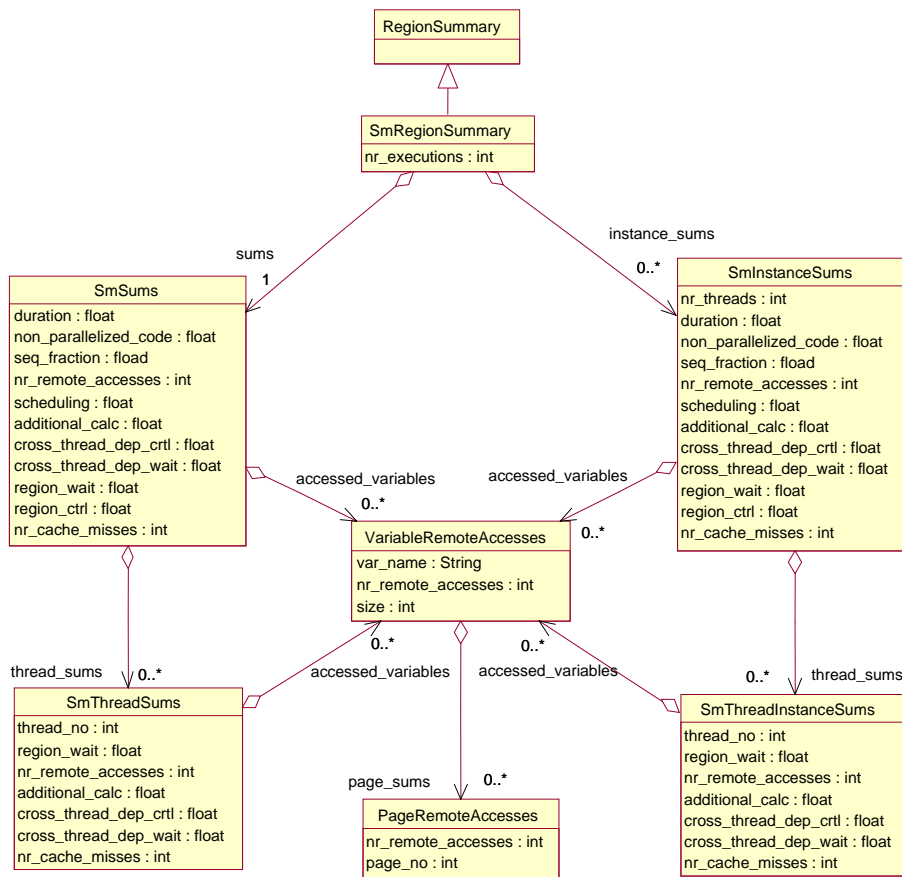


Figure 2.10: OpenMP classes for dynamic information

- *cross_thread_dep_ctrl*: synchronization time except for entry and exit barriers and waiting in locks
- *cross_thread_dep_wait*: synchronization waiting time except waiting in entry or exit barrier
- *region_wait*: waiting time in entry or exit barrier
- *region_ctrl*: time needed to execute region control instructions (e.g. controlling barriers)
- *nr_cache_misses*: number of cache misses
- *thread_sums*: summary data for every thread executing the region
- *accessed_variables*: set of remote access counts for individual variables referenced in that region

Note that attributes *duration* and *region_ctrl* are given with respect to the master thread, whereas all other attributes are average values across all threads that execute a region. Summary data (described by class *SmThreadSums*) for every thread executing a region is specified by *thread_sums* in *SmSums*. The attributes of *SmThreadSums* are a subset of class *SmSums* attributes and refer to summary information for specific threads identified by a unique thread number (*thread_no*).

In addition to the number of remote accesses in a region, the number of remote accesses is collected for individual variables that are referenced in that region. This information is modeled by the class *VariableRemoteAccesses* with the attributes *var_name*, *nr_remote_accesses*, and *size* which denotes the total size of the variable in bytes. This information can be measured if address range specific monitoring is supported, e.g. [KaLeObWa 98]. The last attribute of this class is *page_sums* which is a set of page-level remote access counters. For example, the remote access counters on SGI Origin 2000 provide such information [CuSiGu 99]. With the help of additional mapping information, i.e. mapping variables to addresses, this information can be related back to program variables. Each object of class *PageRemoteAccesses* determines the *page_no* and the number of remote accesses.

The second attribute of *SmRegionSummary* is given by *instance_sums* which is described by a class *SmInstanceSums*. This class specifies summary information for a specific region instance. *SmInstanceSums* contains all attributes of *SmSums* and the number of threads executing the region instance. Finally, class *SmThreadInstanceSums* describes summary information for a given region instance with respect to individual threads.

Chapter 3

Performance Property Specification

A performance property characterizes an aspect of the dynamic behavior of an application. In the context of automatic performance analysis we need to specify only performance properties describing inefficient behavior.

A performance property typically occurs in a specific context. This context can include the program region, a specific process, or a specific instance of that region in a specific process. For example, a property *message passing* can exist for specific regions, such as a subroutine and a message passing statement, or for a specific instance of a region, e.g. an instance of a subroutine in a process.

The existence of a property can be checked by evaluating appropriate conditions based on static and dynamic performance-related data. It is clearly possible that different conditions might exist at the same time, especially if conditions may only give an indication for the existence of that property. Whether a condition proves the existence or indicates the existence is determined by the confidence expression of the property (see below).

Which condition is evaluated might depend on the required information. For example, on shared virtual memory machines thrashing of pages is a major problem. A very good indication for thrashing is a big number of page faults. But, this is not a proof. To be able to prove its existence, individual page fault events have to be traced which may perturbate program execution much more than profiling and huge amounts of data can be generated.

As mentioned above, for each condition, a confidence value between 0 and 1 indicates the confidence in this check. A tool might use this information to first do a fast and simple check with a lower confidence based on already existing information before requesting more detailed information.

The last feature of a performance property is the severity expression. It returns a value indicating the importance of the property in relation to other performance properties.

Performance properties of parallel programs belong to different categories. For example, synchronization and message passing belong to the execution time category, while memory overhead belongs to the memory category. The severity expression of the properties in a single category can easily be normalized so that a global ranking of those properties is possible. For properties of different categories it is difficult or impossible to do that. We currently favor the concept of having global conversion functions between categories. Those conversion functions could easily be adapted to the programmer's preferences while the individual severity expressions need not be changed.

3.1 Specification Language

This section introduces the syntax constructs of the APART specification language for specifying performance properties. The syntax for the structure of the whole specification is shown in Figure 3.1. The specification consists of the performance-related data model specification followed by the performance property specifications. The latter section might include property templates and metaproperties which are defined in Section 3.2.

The performance property part consists of a set of global definitions followed by the property specifications. The definitions specify functions or constants that can be used in the individual property

```

performance-property-spec  is  PERFORMANCE DATA
                               class-def *
                               PERFORMANCE PROPERTIES
                               [LET
                               def *
                               IN]
                               ( property OR template ) *
                               END

```

Figure 3.1: Overall structure of the specification including the performance-related data model and the performance properties.

```

property  is  PROPERTY pp-name '(' arg-list ')' '{'
              [LET
              def *
              IN]
              pp-condition
              pp-confidence
              pp-severity
              '};'
or  PROPERTY template-name '<' ident-list '>' pp-name ';'

pp-condition  is  CONDITION ':' conditions ';'
conditions   is  condition
or  condition OR conditions
condition   is  ['(' cond-id ')'] bool-expr

pp-confidence is  CONFIDENCE ':' MAX '(' confidence-list ')' ';'
or  CONFIDENCE ':' confidence ';'
confidence   is  ['(' cond-id ')'] '->' ] arith-expr

pp-severity  is  SEVERITY ':' MAX '(' severity-list ')' ';'
or  SEVERITY ':' severity ';'
severity    is  ['(' cond-id ')'] '->' ] arith-expr

```

Figure 3.2: Property specification syntax.

<i>def</i>	is	<i>function-def</i>
	or	<i>const-def</i>
<i>function-def</i>	is	<i>type ident</i> '(' <i>arg-list</i> ')' '=' <i>expr</i> ';' ;
<i>const-def</i>	is	<i>type ident</i> '=' <i>expr</i> ';' ;
<i>expr</i>	is	<i>set-expr</i>
	or	<i>arith-expr</i>
	or	<i>bool-expr</i>
<i>arg</i>	is	<i>type ident</i>

Figure 3.3: Definition of functions and constants facilitating subsequent specifications.

specifications to make them more readable.

The syntax shown in Figure 3.2 defines two different ways for specifying properties either via a property specification or by instantiation of a property template. A property specification defines the name of the property, its context via a list of parameters, and the condition, confidence, and severity expressions. Each property specification can also include local definitions that are then available in subsequent specifications of the property.

The property specification is based on a set of parameters. These parameters specify the property's context and parameterize the expressions.

While the context of a property was defined above, the use of additional parameters will be shown by an example. The severity specification will typically be based on a parameter specifying the ranking basis (*rank_basis*). If, for example, a representative test run of the application has been monitored, the time spent in message passing should be compared to the total execution time. If, instead, a short test run is the basis for performance evaluation since the application has a cyclic behavior, the message passing overhead should be compared to the execution time of the shortened loop. Then the loop region is the appropriate *rank_basis*.

The condition specification consists of a list of conditions. A condition is a predicate that can be prefixed by a condition identifier (*cond-id*). The identifiers have to be unique with respect to the property since the confidence and severity specifications can refer to the conditions via those condition identifiers.

The confidence specification is an expression that computes the maximum of a list of confidence values. Each confidence value is computed via an arithmetic expression resulting in a value in the interval of 0 and 1. The value can be guarded by a condition identifier introduced in the condition specification. The condition identifier represents the value of the condition. This confidence value is computed only if the condition evaluates to TRUE.

The severity specification has the same structure as the confidence specification. It computes the maximum of the individual severity values of the conditions.

Figure 3.3 specifies the syntax of definitions that either can be local to a property or global to all properties. Two types of definitions are allowed: definitions of functions and constants. The right-hand side of both definitions are boolean, arithmetic, or set expressions.

Figure 3.4 defines the syntactical structure of boolean expression, also called predicates. Predicates can be built from other predicates with the standard boolean operations. An atomic predicate is either a reference to a boolean attribute or an external function returning a boolean value. Functions can be supplied by the environment. Information in the data model can be accessed via references.

Set expressions (Figure 3.5) can also be built with standard operations. Here, the terminal symbols '+', '/', and '-' denote set union, set intersection and subtraction, respectively. The language also supports the union and intersection operators for sets of sets, as well as the UNIQUE function. It selects a unique value from the set given as argument.

Figure 3.6 introduces the syntax of arithmetic expressions. In addition to the standard syntax, we allow arithmetic operations that work on elements of sets.

```

bool-expr      is bool-expr AND bool-expr
                or bool-expr OR bool-expr
                or NOT bool-expr
                or '(' bool-expr ')'
                or quantifier-list SUCH THAT bool-expr
                or function-name '(' expr-list ')'
                or reference
                or arith-expr relop arith-expr
quantifier    is FORALL bound-variable-list
                or EXISTS bound-variable-list
                or NEXISTS bound-variable-list
bound-variables is ident-list IN set-expr
relop         is '>'
                or '<'
                or '=='
                or '!='
                or '>='
                or '<='
reference     is ident
                or ident.'reference

```

Figure 3.4: Syntax for predicates.

```

set-expr      is set-expr '+' set-expr
                or set-expr '/' set-expr
                or set-expr '-' set-expr
                or '(' set-expr ')'
                or reference
                or function-name '(' parm-list ')'
                or '{' [set-expr WHERE ]ident IN set-expr [WITH bool-expr ]}'
                or set-op '(' set-expr ')'
set-op        is UNION
                or INTERSECTION
                or UNIQUE
parm         is expr

```

Figure 3.5: Syntax for set expressions.

```

arith-expr    is arith-expr '+' arith-expr
                or arith-expr '-' arith-expr
                or arith-expr '*' arith-expr
                or arith-expr '/' arith-expr
                or '(' arith-expr ')'
                or reference
                or function-name '(' parm-list ')'
                or arith-set-op '(' arith-expr WHERE ident IN set-expr ')'
arith-set-op is SUM
                or COUNT
                or MAX
                or MIN
                or STDEV

```

Figure 3.6: Syntax for arithmetic expressions.

```

template           is  PROPERTY TEMPLATE template-name
                    '<' template-parameter-list '>' '(' arg-list ')' '{'
                    [LET
                      def *
                    IN]
                    pp-condition
                    pp-confidence
                    pp-severity
                    '};'

template-parameter is  function-parameter
                        or  property-parameter

function-parameter is  type ident '(' type-list ')'
property-parameter is  PROPERTY pp-name '(' type-list ')'

```

Figure 3.7: Template specification syntax.

3.2 Property Templates and Metaproperties

Based on experience with ASL we observed many similar performance properties both within and across different programming paradigms. Frequently properties differ only in accessed performance data but are similar or identical with respect to condition, confidence, and severity. For this purpose we introduce the concept of property templates which are used to support generic descriptions of similar performance properties.

Figure 3.7 defines the syntax of property templates. Property templates are parameterized properties. The parameters are function parameters or property parameters. If a template has only function parameters, new properties can be defined by substituting the function parameters. If a template has additional property parameters, properties can be defined based on the template by substituting the function and property parameters. Those properties are called metaproperties to reflect that they are dependent on other properties.

In the following we show the example template *cost_per_process*. The template has one function parameter - enclosed by “<” and “>” - which reflects a special cost category. The cost category is specified in the definition of the properties based on the template.

```

PROPERTY TEMPLATE CostPerProcess <float CostFunc(MPISummary)>
    (Region r, Experiment e, Process p, RankBasis)
LET
    cost = CostFunc(summary(r,e,p))
IN
    CONDITION:    cost > 0;
    CONFIDENCE:   1;
    SEVERITY:     cost/duration(RankBasis,e);
}

float IoCostFunc(MPISummary rs) = rs.IoTime;
float SyncCostFunc(MPISummary rs) = rs.SyncTime;
float CommCostFunc(MPISummary rs) = rs.CommTime;

PROPERTY CostPerProcess <IoCostFunc> IoCostPerProcess;
PROPERTY CostPerProcess <SyncCostFunc> SyncCostPerProcess;
PROPERTY CostPerProcess <CommCostFunc> CommCostPerProcess;

```

The parameter function of the template is defined separately. In the above example they are defined as *IoCostFunc*, *SyncCostFunc*, and *CommCostFunc*. Property *IoCostPerProcess* is then defined by *PROPERTY CostPerProcess <IoCostFunc> IoCostPerProcess* based on template *CostPerProcess*. Similar is

done for properties *SyncCostPerProcess* and *CommCostPerProcess*.

The other new concept in ASL are metaproperties. In contrast to normal properties, the definition of metaproperties is based on already defined properties.

In the following example we define a property template *PropertyOnAllProcesses* that specifies that the property *PropertyPerProcess* holds across the set of all processes. Its condition is fulfilled if the condition of property *PropertyPerProcess* and the corresponding severity hold across all processes involved in an experiment. Note that functions *condition* and *confidence* access the CONDITION and SEVERITY components of a property passed as parameter. A metaproperty *IoCost* is then defined by *PROPERTY PropertyOnAllProcesses <IoCostPerProcess> IoCost*.

```
PROPERTY TEMPLATE PropertyOnAllProcesses
  <PROPERTY x(Region, Experiment, Process, RankBasis)>
  (Region r, Experiment e, RankBasis) {

    CONDITION:  Forall p IN e.processors SUCH THAT
                  condition(x(r,e,p,RankBasis)) AND
                  confidence(x(r,e,p,RankBasis)) ==1;
    CONFIDENCE: 1;
    SEVERITY:   max(severity(x(r,e,p,RankBasis)) WHERE p in e.processes)
  }

PROPERTY PropertyOnAllProcesses <IoCostPerProcess> IoCost;
PROPERTY PropertyOnAllProcesses <SynCostPerProcess> SyncCost;
PROPERTY PropertyOnAllProcesses <CommCostPerProcess> CommCost;
```

3.3 Properties Based on Pattern Matching in Traces

The kind of available performance data has a large influence on the informative value of performance properties that can be defined using this data. Summary information as collected by profiling tools is sufficient to describe a multitude of frequently occurring performance properties. However, there are performance properties that are not visible in this kind of information. A much more detailed view on the behavior of a parallel application can be gained by using event traces. This allows to prove the presence of performance properties in the application by looking for *compound events* in an event trace.

This section presents an ASL language construct *PATTERN* intended to formally describe compound events that indicate the presence of performance properties in parallel applications. It is based on a framework defining two categories of abstractions, which can be used to provide programming model specific building blocks, on top of which a general specification of compound events is possible. The abstractions should represent entities of the underlying programming model such as MPI messages and are useful to measure their influence on the performance behavior. The resulting specifications can be easily transformed into an appropriate detection algorithm. A detailed description of this framework can be found in [WoMo 00].

3.3.1 Abstractions

An event trace is considered as a sequence $E = \{e_1, \dots, e_{max}\}$ of events in chronological order. Each event is an instance of a certain event type having an associated set of attributes. The framework comprises the following categories of abstractions, which are created for event traces related to a specific programming model:

- System states
- Pointer attributes

Compound events representing performance properties often exhibit some form of locality within the event trace. Usually the constituents of such a compound event depend on some context, which is represented by the state of the whole running parallel system at the time when the compound event occurs. In most cases, this *system state* is characterized by a set of ongoing activities in contrast to activities that are

already finished. An example for such a system state is the set of messages being in transfer at a given moment. A system state is a mapping $S : E \rightarrow \mathcal{P}(E)$ that maps an event $e_i \in E$ from the event trace onto a subset $S(e_i) \subseteq E$ of the events from the event trace. A system state $S(e_i)$ of an event e_i should reflect one aspect of the execution state of the application after the event e_i took place, so system states depend only on historic data and not on future events. A system state is inductively defined by a set of transition rules. Depending on the event type of the current event e_{i+1} , a transition rule defines how a state $S(e_{i+1})$ is created from its predecessor state $S(e_i)$.

Another useful abstraction is to link related events together, so that one can navigate from an event to another related event. This permits to navigate along a path of related events and to define relationships among constituents of a compound event using such paths. A natural way of representing such links is to provide event attributes with pointer semantics. Pointer attributes added to the attributes already associated with the event types are the second category of abstractions defined in the framework. The additional attributes are simply defined using the previously mentioned system states.

3.3.2 Example: MPI Related Abstractions

The dynamic view of an MPI application comprises events related to the control flow as well as to communication. So the ASL specification model contains two event classes **Enter** and **Exit** for entering and leaving a region as well as two event classes **Send** and **Receive** for sending and receiving a point-to-point message. All these event classes are derived from base class **Event**. **Enter** and **Exit** events provide an attribute denoting the region, to which they refer; **Send** and **Receive** events provide attributes denoting the source and destination location of the message as well as attributes describing the message itself.

We describe the activities performed by an MPI application at a given moment in terms of two different higher-level concepts: region instances and messages. The region instances being executed at a moment can be easily represented by the set of **Enter** events that determine their beginnings. So we define for each event e and each location l a system state $\text{Rs}(e, l)$ and call it the region stack for location l .¹ In a similar way, we can describe the set of messages currently being in transfer by the set of their **Send** events. So we define for each event e and for each pair of locations s and d a system state $\text{Mq}(e, s, d)$ and call it the message queue for messages being in transfer from source s to destination d .

In addition, we define for each event a pointer attribute pointing to the **Enter** event of the region instance, in which the event took place. The pointer attribute is expressed in ASL as a function $\text{enterptr}(e)$ of an event e . We also define an ASL function $\text{sendptr}(e)$, which provides a link from a **Receive** event to its corresponding **Send** event.

Now, we have powerful abstractions, which we can use to access the dynamic program entities involved in inefficiency situations.

3.3.3 Specifying Compound Events

A compound event comprises a set of constituents. This set can be divided into *fractions* constituting logical groups within the compound event. The fractions themselves are connected by functional dependencies, which allow to calculate one group as a function of other groups. There is one distinct group made up by one single event, which is called the root event. The root event does not depend on any other group, it is solely characterized by a condition requiring a certain type. So, the whole dependency graph forms a tree emanating from the root event. The search algorithm for locating a compound event in an event trace just looks for events having the correct type and tries to evaluate the dependency tree. If it succeeds, a compound-event instance has been found (i.e., it has been instantiated). If it fails, the algorithm can continue at another point in the trace. Apart from the dependencies, which are used to instantiate the different fractions of the compound event, there may be conditions that characterize the compound event as a whole and that are not needed for locating its fractions. We refer to these additional conditions as *constraints* of the compound event.

¹The transition rules can be easily implemented in ASL using a recursive definition.

3.3.4 Compound Events in ASL

Compound events are specified in ASL using the ASL *pattern* construct. Its name is motivated by thinking of compound events as event patterns. Its syntax is defined in Fig. 3.8.

```

pattern      is  PATTERN p-name '(' arg-list ')' '{'
                [LET
                  def *
                IN ]
                p-roottype
                p-instantiation
                p-constraint
                p-export
                '};'

arg          is  type ident
p-roottype   is  ROOTTYPE ':' ident ';'
p-instantiation is  INSTANTIATION ':' const-def *
p-constraint is  CONSTRAINT ':' bool-expr
p-export     is  EXPORT [m-name ] ':' const-def *

```

Figure 3.8: Pattern specification syntax

Compound event specifications can be parameterized by declaring formal parameters in the *arg-list*. These parameters as well as the local definitions from the optional LET clause can be used in the subsequent parts of the compound-event specification. The ROOTTYPE clause contains the type of the root event. The fractions of the compound event are defined in the INSTANTIATION clause. Fractions consisting of only a single element are expressed by simple constant definitions. Fractions consisting of more than one element have to be defined using a set type. A condition representing additional compound-event properties that are not needed for instantiation can be placed in the CONSTRAINT clause. The EXPORT clause defines attributes whose values are computed from the constituents of a compound-event instance. The attributes can be accessed through *match objects* that correspond to compound-event instances. How to obtain these match objects is explained in the following section. So, the export clause implicitly defines a class, to which these match objects will belong. If necessary, the class can be given a name *m-name*.

The root event as well as the complete event trace can be referenced in an ASL pattern by the two keywords ROOT and TRACE. In a future implementation, these keywords are bound to the current root event and the event trace being investigated by the search algorithm.

The *pattern* construct should not be seen as an alternative to the *property* construct. It is rather an instrument to increase its expressiveness as we will see later.

3.3.5 Pattern Matches

The ASL pattern construct is used to specify two things. Firstly, it specifies a compound event, that is a set of events being connected by some relationships and fulfilling some constraints. This first aspect is collectively expressed by the roottype, the instantiation, and the constraint clause. Secondly, the pattern specifies a class *m-name* of match objects, which are computed from compound-event instances and which are used to represent performance-relevant metrics such as idle times. This second aspect is expressed by the export clause.

ASL provides an intrinsic function to obtain the match objects computed from all compound event instances occurring in an event trace:

```
setof m-name PATTERN_MATCHES( p-name ( arg-list ), setof Event trace);
```

The function takes two arguments. The first argument is an ASL pattern provided with its own argument list according to its definition. The second argument is the event trace to be analyzed, which is represented by a set of events. When the function is invoked, the TRACE keyword mentioned in the preceding section

is bound to this set. The function returns the set of match objects corresponding to all compound-event instances according to the ASL pattern.

3.3.6 Example: MPI Related Compound Events

In this section we give an example of utilizing the ASL pattern construct for specifying compound events that indicate the existence of performance properties.

Our example describes the situation that occurs, when an MPI_Recv operation is posted before the corresponding MPI_Send has been started. Figure 3.9 shows the time lines of two MPI processes involved in this inefficiency situation. The receiver remains idle during the interval between the two calls instead of doing useful computation.

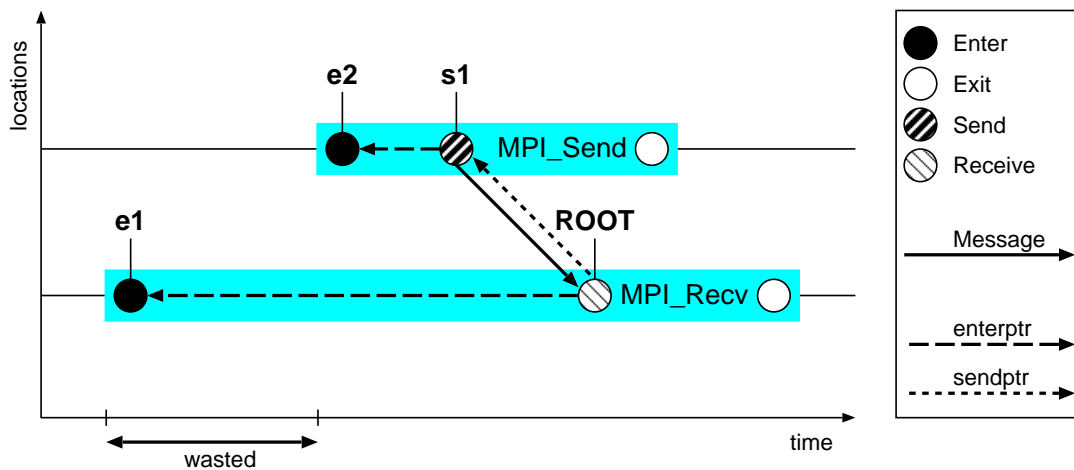


Figure 3.9: Late sender compound event

The ASL specification of this compound event is shown in Fig. 3.10. It consists of four fractions ROOT, s1, e1, and e2, each containing only a single event. The root event or fraction is just the event indicating the message arrival (i.e., an event of type Receive), which is expressed by the roottype clause.

```
PATTERN LateSender(Region r) {
    ROOTTYPE: Receive;

    INSTANTIATION:
        Send s1 = sendptr(ROOT);
        Enter e1 = enterptr(ROOT);
        Enter e2 = enterptr(s1);
    CONSTRAINT:
        EXISTS e IN Rs(ROOT, ROOT.process_id) SUCH THAT e.region == r AND
        enterptr(ROOT).region == MPI_Recv AND
        enterptr(s1).region == MPI_Send AND
        e2.timestamp > e1.timestamp;
    EXPORT:
        float idle_time = e2.timestamp - e1.timestamp;
}
```

Figure 3.10: Late sender specification in ASL

The other three events are the event of sending the message (s_1), the event of entering the MPI_Send region (e_2), and the event of entering the MPI_Recv region (e_1). They are defined in the instantiation clause using the previously defined pointer attributes as shown in Fig. 3.9.

```

PROPERTY late_sender(Region r, Experiment e, Region rank_basis){
  LET
    float idle_time = SUM m.idle_time
      WHERE m IN PATTERN_MATCHES(LateSender(r), e.trace);
  IN
    CONDITION: idle_time>0;
    CONFIDENCE: 1;
    SEVERITY: idle_time/duration(rank_basis, e);
}

```

Figure 3.11: Late sender property using a pattern

The first sub-proposition of the conjunction in the constraint clause requires the root event to occur when the process of the root event is executing in region r . This is expressed by using the region-stack abstraction. Region r is supplied as a parameter of the pattern. The next two sub-propositions require the involved region instances to be `MPI_Recv` and `MPI_Send`. The last sub-proposition describes the necessary temporal displacement between the two function calls.

The export clause makes this displacement accessible through an attribute `idle_time`.

3.3.7 Using Patterns in Property Definitions

The purpose of patterns is to make the very detailed information contained in event traces available to property definitions. In order to meet our goal, we define a `late_sender` property using the previously explained pattern in Fig. 3.11.

This property refers to a region r that creates late-sender instances during its execution. The confidence is 1, since our criterion is safe. The severity corresponds to the time lost by the sum of all late-sender instances. The time lost by individual late-sender instances is exported by the `idle_time` attribute of the pattern.

3.3.8 Summary

Event traces provide a very fine grained view on the performance behavior of a parallel application. Based on this view, performance properties that cannot be represented by profiling data can be specified in terms of compound events. For this reason, ASL provides a language construct *pattern*, which allows to specify complex performance properties by means of event traces. For a more detailed description we refer to [WoMo 00].

3.4 Paradigm Related Property Specification

This section presents the current set of performance properties for the selected programming paradigms. This set is not intended to be a full catalog of performance properties but is a collection of typical examples showing the applicability of the language features.

3.4.1 MPI

This section demonstrates the features of the APART specification language in the context of the message passing paradigm. Although most of the properties are independent of the specific message passing library, the terminology used is based on MPI.

The following MPI-related performance properties are presented:

- TotalCost
- CommCost

- SyncCost
- IoCost
- DominatingCommFunction
- DominatingCommCall
- FrequentCommunication
- BigMessages
- LateSender
- LateReceiver
- UnevenMpDistribution
- LoadImbalanceAtBarrier
- SlowSlaves
- MessagesInWrongOrder
- FrequentCommInProcess
- FrequentCommOnAllProcesses
- FrequentCommSomeProcesses

The following memory-related performance properties are presented:

- L1DCacheMisses
- L1ICacheMisses
- L2CacheMisses
- L1DCacheMissesPerInstance
- L1ICacheMissesPerInstance
- L2CacheMissesPerInstance
- L1DCacheMissesInAllInstancesInProcess
- L1ICacheMissesInAllInstancesInProcess
- L2CacheMissesInAllInstancesInProcess
- L1DCacheMissesInSomeNotAllInstancesInProcess
- L1ICacheMissesInSomeNotAllInstancesInProcess
- L2CacheMissesInSomeNotAllInstancesInProcess
- L1DCacheMissesInAllInstancesInAllProcess
- L1ICacheMissesInAllInstancesInAllProcess
- L2CacheMissesInAllInstancesInAllProcess

3.4.1.1 MPI global definitions

```

MPISummary Summary(MPIRegion r, Experiment e)=
    UNIQUE({sumr IN e.profile WITH sumr.region==r AND sumr.Proc==ALLPROCESSES});

MPISummary Summary(MPIRegion r, Experiment e, Process p)=
    UNIQUE({sumr IN e.profile WITH sumr.region==r AND sumr.Proc==p});

Setof MPISummary ProcessProfiles(MPIRegion r, Experiment e)=
    {sumr IN e.profile WITH sumr.region==r AND sumr.Proc!=ALLPROCESSES}

float Duration(MPIRegion r, Experiment e)= Summary(r,e).Incl;

float Duration(MPIRegion r, Experiment e, Process p)= Summary(r,e,p).Incl;

```

In most property specifications it is necessary to access the summary data of a given region for a given experiment. Therefore, we defined the *Summary* function that returns the appropriate *MPISummary* object. *Summary* is an overloaded function. Depending on the existence of the process parameter it selects summary information for a single process or for all processes. It is based on the set operation *UNIQUE* that selects arbitrarily one element from the set argument which has cardinality one due to the design of the data model.

The second function *Duration* returns the inclusive execution time for a single process or for all processes. For all MPI performance properties the severity is computed by relating some portion of the execution time to the duration of a given *RankBasis* region in the experiment.

3.4.1.2 TotalCost

```

PROPERTY TEMPLATE Cost <float CostFunc(Region r, Experiment e)>
    (Region r, Experiment e, Region RankBasis){
LET
    Cost = CostFunc(r,e)
IN
    CONDITION:    Cost > 0;
    CONFIDENCE:   1;
    SEVERITY:     Cost/Duration(RankBasis,e);
}

float AllCostFunc(Region r, Experiment e) =
    summary(r,e).CommTime +
    summary(r,e).SyncTime +
    summary(r,e).IoTime;

PROPERTY Cost<AllCostFunc> TotalCost;

```

The most general performance property *TotalCost* characterizes the region as having some performance overheads or costs. The costs of a region can be subdivided into time for communication, time for synchronization, i.e. barrier synchronization, and time for I/O.

The definition of this property is based on the property template *Cost*. The template has a single function parameter which defines the cost category.

A region has this property if *Cost* is greater than 0. Clearly the confidence in that condition is one. The severity of this property is the fraction of the time spent for costs compared to the duration of ranking basis, typically the duration of the main program. Note, that *CommTime*, *SyncTime*, *IoTime*, and *Duration* are sums of the time spent in each process.

The severity of this property may be larger than the severity of the individual properties for each of the categories. This may lead to the selection of the *TotalCost* property as a performance problem according to the predefined severity threshold while the individual properties, i.e. *CommCost*, *SyncCost*, and *IoCost* may not be marked as performance problems.

3.4.1.3 CommCost, SyncCost, IoCost

```
float CommCostFunc(Region r, Experiment e)= Summary(r,e).CommTime;
float SyncCostFunc(Region r, Experiment e)= Summary(r,e).SyncTime;
float IoCostFunc(Region r, Experiment e)= Summary(r,e).IoTime;
```

```
PROPERTY Cost<CommCostFunc> CommCost;
PROPERTY Cost<SyncCostFunc> SyncCost;
PROPERTY Cost<IoCostFunc> IoCost;
```

These properties determine whether a region includes communication, barrier synchronization, or IO, respectively.

3.4.1.4 CacheCost

```
PROPERTY TEMPLATE CacheCost <float CostFunc(Region, Experiment)>
    (Region r, Experiment e, Region RankBasis) {
LET
    float Costs=CostFunc(r,e);
IN
    CONDITION: Costs>0;
    CONFIDENCE: 1;
    SEVERITY: Cost/(Summary(RankBasis,e).LoadInstr
        + Summary(RankBasis,e).StoreInstr) ;
}
```

```
L1DCacheCostFunc(Region r, Experiment e)= Summary(r,e).L1DCacheMisses;
L1ICacheCostFunc(Region r, Experiment e)= Summary(r,e).L1ICacheMisses;
L2CacheCostFunc(Region r, Experiment e)= Summary(r,e).L2CacheMisses;
```

```
PROPERTY CacheCost<L1DCacheCostFunc> L1DataCacheMisses;
PROPERTY CacheCost<L1ICacheCostFunc> L1InstCacheMisses;
PROPERTY CacheCost<L2CacheCostFunc> L2CacheMisses;
```

These properties determine whether cache misses are a performance problem for a given region. In contrast to other properties, the severity is the percentage of the cache misses at the load and store instructions. If all properties should be ranked according to severity, this definition has to be replaced to reflect the execution time lost during cache misses. This time can be estimated based on some mean value for a cache miss.

3.4.1.5 DominatingCommFunction

```

PROPERTY DominatingCommFunction(MPIFunction r, Experiment e, Region RankBasis){
LET
  setof MPISummary CommSummaries=
    {x IN e.profile
     WITH
       x.Proc==ALLPROCESSES
     AND
       typeof(x.region)==MPICommFunc};
  float MaxCommTime = MAX(sum.Incl WHERE sum IN CommSummaries);
IN
  CONDITION: typeof(r)==MPICommFunc AND Duration(r,e)==MaxCommTime;
  CONFIDENCE: 1;
  SEVERITY: MaxCommTime/Duration(RankBasis,e);
}

```

The MPI communication function with the maximum communication time has the *DominatingCommFunction* property. The constant *CommSummaries* specifies the set of summary objects of communication functions, i.e. it excludes *MPI_barrier*. The constant *MaxCommTime* is the maximum execution time of all MPI communication functions. *typeof(obj)* denotes the type of a class such that *obj* is an instance of that class.

The condition of this property checks whether the region is a communication function, and whether its duration is equal to *MaxCommTime*. If this condition is fulfilled, this function is the dominating communication function. Its severity is the fraction of the communication time in relation to the execution time of the *RankBasis*.

3.4.1.6 DominatingCommCall

```

PROPERTY DominatingCommCall(FuncCall r, Experiment e, Region RankBasis){
LET
  setof MPISummary CommSummaries=
    {x IN e.profile
     WITH
       x.Proc==ALLPROCESSES AND typeof(x.region)==FuncCall
     AND
       typeof(x.region.CalledFunc)==MPICommFunc};
  float MaxCommTime = MAX(sum.Incl WHERE sum IN CommSummaries);
IN
  CONDITION: typeof(r.CalledFunc)==MPICommFunc AND
             Duration(r,e)==MaxCommTime;
  CONFIDENCE: 1;
  SEVERITY: MaxCommTime/Duration(RankBasis,e);
}

```

Frequently it is more interesting to the programmer to identify the region of the code where most of the communication takes place. This property is very similar to the previous one except that it determines the dominating call site of communication functions.

3.4.1.7 FrequentCommunication

```

PROPERTY FrequentCommunication (FuncCall r, Experiment e, Region RankBasis){
LET
  float Cost = Duration(r,e);
IN
  CONDITION: typeof(r.CalledFunc)==MPICommFunc AND Cost>0 AND
             Cost/Summary(r,e).NrExecutions<SmallMessagesThreshold;
  CONFIDENCE: 1;
  SEVERITY:   Cost/Duration(RankBasis,e);
}

```

A communication statement has the property *FrequentCommunication* if small messages are communicated. The condition compares the execution time per execution with the maximum communication time for small messages. Whether messages are called big depends on the opinion of the tool designer or the application programmer. Therefore, this threshold should be a parameter of the performance tool.

3.4.1.8 BigMessages

```

PROPERTY BigMessages (FuncCall r, Experiment e, Region RankBasis){
LET
  float Cost = Summary(r,e).CommTime;
  int AvgLength = summary(r,e).MessageLength/
                 summary(r,e).NrExecutions;
IN
  CONDITION: typeof(r.CalledFunc)==MPICommFunc AND Cost>0 AND
             AvgLength>BigMessagesThreshold;
  CONFIDENCE: 1;
  SEVERITY:   Cost/Duration(RankBasis,e);
}

```

The *BigMessages* property is fulfilled by a communication statement if the average message length is greater than a predefined threshold.

3.4.1.9 LateSender

```

PROPERTY LateSender(FuncCall r, Experiment e, Region RankBasis){
LET
  float IdleTime = summary(r,e).IdleTime;
IN
  CONDITION: typeof(r.CalledFunc)==MPICommFunc
             AND r.CalledFunc.Name == MPI_Receive AND IdleTime>0;
  CONFIDENCE: 1;
  SEVERITY: IdleTime/Duration(RankBasis,e);
}

```

Only point-to-point receive operations can have the *LateSender* property. Therefore, the region parameter in the parameter list must be of type *FuncCall*.

The condition checks that the statement is a receive statement and the idle time is greater than zero. The severity of this property compares the idle time to the duration of *RankBasis*.

3.4.1.10 LateReceiver

```

PROPERTY LateReceiver(FuncCall r, Experiment e, Region RankBasis){
LET
  float IdleTime = Summary(r,e).IdleTime;
IN
  CONDITION: typeof(r.CalledFunc)==MPIp2pFunc
             AND r.CalledFunc.Name == MPI_SEND
             AND r.Semantics == Blocking AND idle_time>0;
  CONFIDENCE: 1;
  SEVERITY: IdleTime/Duration(RankBasis,e);
}

```

This property can only be proven for send operations that are blocking. Nonblocking operations just setup the transmission and terminate. The severity compares idle time with the duration of *RankBasis*.

3.4.1.11 UnevenMpDistribution

```

PROPERTY UnevenMpDistribution(FuncCall r, Experiment e,
                             Region RankBasis)
{
LET
  float deviation=stdev(s.Incl WHERE s IN ProcessProfiles(r,e));
IN
  CONDITION: typeof(r.CalledFunc)==MPICommFunc AND
             deviation > uneven_threshold * summary(r,e).Incl/
             e.nr_processors;
  CONFIDENCE: 1;
  SEVERITY: summary(r,e).Incl/Duration(RankBasis,e);
}

```

Any communication statement can have the *UnevenMpDistribution* property. The constant *deviation* determines the standard deviation of the execution time of the processes. The condition checks whether the deviation is greater than a threshold multiplied with the mean execution time.

3.4.1.12 LoadImbalanceAtBarrier

```

PROPERTY LoadImbalanceAtBarrier(FuncCall r, Experiment e, Region RankBasis){
LET
  float MaxTime=max( x.Incl WHERE x IN ProcessProfiles(r,e) );
  float MinTime=min( x.Incl WHERE x IN ProcessProfiles(r,e) );
  float MaxWait=MaxTime - MinTime;
IN
  CONDITION: (COND1) r.CalledFunc.Name==MPI_Barrier AND
             max_wait>0
             || (COND2) r.CalledFunc.Name==MPI_Barrier AND
             summary(r,e).IdleTime>0;
  CONFIDENCE: 1;
  SEVERITY: MAX((COND1)->MaxWait/(Duration(RankBasis,e)/e.nr_processors),
                (COND2)->Summary(r,e).IdleTime/Duration(RankBasis,e));
}

```

The *LoadImbalanceAtBarrier* property has two conditions. The first condition can be evaluated if the idle times cannot be measured, while the second condition is based on the idle times. While the confidence value is equal for both conditions, the severity is specified by different formulas. If the first condition is satisfied, the severity is determined by dividing *MaxWait* time by the mean duration of each process. If

the second condition is fulfilled, the sum of the idle times in all processes is compared to the sum of the individual execution times.

3.4.1.13 SlowSlaves

```
PROPERTY SlowSlaves (FuncCall r, MPIApplication a,
                    Experiment e, Region RankBasis){

LET
  float IdleTime = summary(r,e).IdleTime;
IN
  CONDITION: a.Paradigm == MasterSlave AND r.Role == ReceiveMaster AND
             IdleTime>0;
  CONFIDENCE: 1;
  SEVERITY: IdleTime/Duration(RankBasis,e),
}

```

Both properties, *SlowSlaves* and *OverloadedMaster* described below, are related to the master slave paradigm. In this paradigm, four communication statements are special statements. In the master, a send operation distributes the task to the slaves and a receive operation collects the results. Those statements play the *SendMaster* and *ReceiveMaster* role. In the slaves, a receive operation (*ReceiveSlave* role) accepts tasks and a send operation (*SendSlave* role) returns the results.

The *SlowSlaves* property can be proven for the *ReceiveMaster* statement. It identifies a situation where the master waits for results instead of doing useful work.

3.4.1.14 OverloadedMaster

```
PROPERTY OverloadedMaster(FuncCall r, MPIApplication a,
                          Experiment e, Region RankBasis){

LET
  float IdleTime = summary(r,e).IdleTime/(e.NrProcessors-1);
IN
  CONDITION: (a.Paradigm == MasterSlave AND
             (r.Role == ReceiveSlave OR r.Role == SendSlave)) AND
             IdleTime>0;
  CONFIDENCE: 1;
  SEVERITY: IdleTime/Duration(RankBasis,e);
}

```

The *OverloadedMaster* property can be proven for the *ReceiveSlave* and the *SendSlave* operations. If the slaves have to wait for new tasks or for the delivery of the results of finished tasks, the master is too slow.

3.4.1.15 WrongOrder

```

PATTERN WrongOrder(FuncCall r) {

  ROOTTYPE: Receive;
  INSTANTIATION:
    Send  SendMsg = sendptr(ROOT);
    Enter EnterSend = enterptr(SendMsg);
    Enter EnterRecv = enterptr(ROOT);
    Exit  ExitRecv = UNIQUE({
      ev IN TRACE
      WITH enterptr(ev) == EnterRecv
    });
    Exit  ExitSend = UNIQUE({
      ev IN TRACE
      WITH enterptr(ev) == EnterSend
    });

  CONSTRAINT:
    EXISTS e IN Rs(ROOT, ROOT.Process) SUCH THAT e.Region==r AND
    EXISTS s IN Mq(ROOT, SendMsg.Process, ROOT.Process)
    SUCH THAT s.Timestamp < SendMsg.Timestamp;

  EXPORT:
    cost = (ExitSend.Timestamp - EnterSend.Timestamp) +
           (ExitRecv.Timestamp - EnterRecv.Timestamp);
}

```

The pattern identifies a situation in the trace of an MPI program, where a message is received although an older message sent from the same sender to the same destination was not yet received at that time. Thus the two messages were sent in the wrong order. The ROOT of the pattern is a receive event. The constraint checks first that ROOT is actually the receive statement passed to the pattern. The constraint checks also that there is a message in the set of currently active messages sent from the same sender to the same receiver. This message must have been sent before the send operation started that transferred the received message.

```

PROPERTY MessagesInWrongOrder(FuncCall r, Experiment e, Region RankBasis){
  LET
    float Cost= SUM(i.cost WHERE i IN
      PATTERN_MATCHES(WrongOrder(r), e));
  IN
    CONDITION: Cost>0;
    CONFIDENCE: 1;
    SEVERITY: Cost/Duration(RankBasis,e);
}

```

The property checks whether time was lost due to messages sent in wrong order.

3.4.1.16 FrequentCommInProgress

```

PROPERTY FrequentCommInProgress(Function r, Experiment e, Process p
                                Region RankBasis){
LET
  float cost = Summary(r,e,p).Incl;
IN
  CONDITION:  typeof(r)==MPICommFunc AND cost>0 AND
              cost/Summary(r,e,p).NumPasses < SmallMessagesThreshold;
  CONFIDENCE: 1;
  SEVERITY:   cost/duration(RankBasis,e,p);
}

```

This property is assigned to a communication function and a process if only small messages are sent in that process via that function.

3.4.1.17 FrequentCommOnAllProcesses

```

PROPERTY TEMPLATE OnAllProcesses< PROPERTY x(Region, Experiment,
Process, Region>(Region r, Experiment e, Region RankBasis) {

  CONDITION: FORALL p IN e.Processes
             SUCH THAT
               condition(x(r,e,p,RankBasis))
             AND
               confidence(x(r,e,p,RankBasis))==1;
  CONFIDENCE: 1;
  SEVERITY:   max(severity(x(r,e,p,RankBasis)))*e.NrProcessors;
}

```

```

PROPERTY OnAllProcesses<FrequentCommProcess> FrequentCommOnAllProcesses;

```

Property *FrequentCommOnAllProcesses* is a metaproperty. It is based on property *FrequentCommProcess* defined above. The property template *OnAllProcesses* facilitates the specification of the new property. The general check that the condition and confidence are fulfilled for all processes is specified in the template. This template can easily be applied to specify similar more global properties based on process-level properties.

3.4.1.18 CacheMissesPerInstance

```

PROPERTY TEMPLATE CacheCostPerInstance <float CostFunc(Exit)>
    (Exit re, Region RankBasis) {
LET
    float costs=CostFunc(re);
IN
    CONDITION: costs>0;
    CONFIDENCE: 1;
    SEVERITY: cost/(re.LoadInstr + re.StoreInstr) ;
}

L1DCacheCostInstFunc(Exit re)= re.L1DCacheMisses;
L1ICacheCostInstFunc(Exit re)= re.L1ICacheMisses;
L2CacheCostInstFunc(Exit re) = re.L2CacheMisses;

PROPERTY CacheCostPerInstance<L1DCacheCostInstFunc> L1DCacheMissesPerInstance;
PROPERTY CacheCostPerInstance<L1ICacheCostInstFunc> L1ICacheMissesPerInstance;
PROPERTY CacheCostPerInstance<L2CacheCostInstFunc> L2CacheMissesPerInstance;

```

The property template is used to defined properties that identify cache misses in an individual execution of a region. Such an instance in a process is represented in the data model by a pair of *Entry* and *Exit* events.

3.4.1.19 Property for all instances of a region

```

PROPERTY TEMPLATE AllInstancesInProcess
  <PROPERTY x(RegionExit)>(Region r, Experiment e,
                          Process p, Region RankBasis) {
LET
  Setof Exit AllRe = {re IN e.Trace WITH re.Process==p
                      AND re.Region==r}
IN
  CONDITION: FORALL re IN AllRe
              SUCH THAT
                condition(x(re))
              AND
                severity(x(re))>0.25;
  CONFIDENCE: 1;
  SEVERITY: max(severity(x(re)));
}

PROPERTY AllInstancesInProcess <L1DCacheMissesPerInstance>
  L1DCacheMissesInAllInstancesInProcess;
PROPERTY AllInstancesInProcess <L1ICacheMissesPerInstance>
  L1ICacheMissesInAllInstancesInProcess;
PROPERTY AllInstancesInProcess <L2CacheMissesPerInstance>
  L2CacheMissesInAllInstancesInProcess;

PROPERTY OnAllProcesses <L1DCacheMissesInAllInstancesInProcess>
  L1DCacheMissesInAllInstancesInAllProcesses;
PROPERTY OnAllProcesses <L1ICacheMissesInAllInstancesInProcess>
  L1ICacheMissesInAllInstancesInAllProcesses;
PROPERTY OnAllProcesses <L2CacheMissesInAllInstancesInProcess>
  L2CacheMissesInAllInstancesInAllProcesses;

```

This property template specifies that the property passed as parameter holds for all instances of a region in a specific process. As examples, properties about cache misses in all instances in a single process and in all processes are specified via the property templates *AllInstancesInProcess* and *OnAllProcesses*.

3.4.1.20 Property template for some instances

```

PROPERTY TEMPLATE SomeNotAllInstances< PROPERTY x(Region, Experiment,
Process, Region>(Region r, Experiment e, Region RankBasis) {
LET
  Setof Exit AllRe = {re IN e.Trace WITH re.Process==p
                      AND re.Region==r}
IN
  CONDITION: (EXISTS re IN AllRe
              SUCH THAT
                condition(x(re))
              AND
                severity(x(re))>0.25)
  AND
  (NOT FORALL re IN AllRe
   SUCH THAT
     condition(x(re))
   AND
     severity(x(re))>0.25)
  );
CONFIDENCE: 1;
SEVERITY: max(severity(x(re)));
}

```

```

PROPERTY SomeNotAllInstancesOfProcess <L1DCacheMissesPerInstance>
  L1DCacheMissesInSomeNotAllInstances;
PROPERTY SomeNotAllInstancesOfProcess <L1ICacheMissesPerInstance>
  L1ICacheMissesInSomeNotAllInstances;
PROPERTY SomeNotAllInstancesOfProcess <L2CacheMissesPerInstance>
  L2CacheMissesInSomeNotAllInstances;

```

This property template specifies that the property passed as parameter holds for some but not all instances of a region in a specific process.

3.4.1.21 FrequentCommSomeProcesses

```

PROPERTY TEMPLATE SomeNotAllProcesses< PROPERTY x(Region, Experiment,
Process, Region)>(Region r, Experiment e, Region RankBasis) {

    CONDITION: (EXISTS p IN e.Processes
                SUCH THAT
                condition(x(r,e,p,RankBasis))
                AND
                confidence(x(r,e,p,RankBasis))==1)
    AND
    (NOT FORALL p IN e.Processes
     SUCH THAT
     condition(x(r,e,p,RankBasis))
     AND
     confidence(x(r,e,p,RankBasis))
    );
    CONFIDENCE: 1;
    SEVERITY: max(severity(x(r,e,p,RankBasis)))*e.NrProcessors;
}

```

```

PROPERTY SomeNotAllProcesses<FrequentCommProcess>
    FrequentCommSomeProcesses;

```

The property template *SomeNotAllProcesses* specifies that the property passed as parameter is fulfilled only in some of the processes. In this example, the *FrequentCommSomeProcesses* is deduced from this template based on the property *FrequentCommProcess*.

3.4.2 HPF

This section introduces performance properties of data parallel programs in the context of HPF. The following performance properties are presented:

- costs
- communication_costs
- forall_synchronization_costs
- io_costs
- parallel_organization_costs
- redist_costs
- serialization_costs
- uneven_work_distribution
- inspector_cost

3.4.2.1 HPF global definitions and templates

In this section we define a function *summary* returns an object to *HPFRegionSummary*. This object reflects summary information for a specific region and experiment and is used by most HPF properties.

```
HPFRegionSummary summary(HPFRegion r, Experiment e)=
    UNIQUE({sumr IN e.profile | sumr.region==r});

float duration(Region r, Experiment e)=summary(r,e).sums.duration;
```

Function *duration* denotes the execution time of a region which is the arithmetic mean across all processes that execute the region.

For all HPF performance properties the severity is computed by relating some aspect of the the execution time to the duration of a given *rank_basis* region in the experiment.

In the following we define a template for generic costs which can be used for various HPF performance properties.

3.4.2.2 template costs

```
PROPERTY TEMPLATE costs <float cost_func(HPFRegionSummary)>
    (HPFRegion r, Experiment e, Region RankBasis)
LET
    cost = cost_func(summary(r,e))
IN
    CONDITION:    cost > 0;
    CONFIDENCE:   1;
    SEVERITY:     cost/duration(rank_basis,e);
}
```

3.4.2.3 costs

```
float par_costs(RegionSummary rs) = rs.sums.comm_time +
    rs.sums.sync_time + rs.sums.compiler_ovh_time + rs.sums.io_time;

PROPERTY costs <par_costs> cost;
```

The most general performance property cost specifies that a region implies some performance costs. The costs of a region described by cost function *par_costs* can be subdivided into communication, synchronization, compiler overhead, and input/output time. Template costs is used to define this performance property According to the template's definition the region accounts for this property if *cost_sum* is greater than 0. The confidence for this condition is one.

The severity of this property is the fraction of the time spent for costs compared to the duration of ranking basis, typically the duration of the main program. Note, that *comm_time*, *sync_time*, *compiler_ovh_time* *io_time*, and *duration* are summary figures across all processes executing the region.

The severity of this property is larger than the severity of the individual properties for each of the categories. This may lead to the selection of the cost property as a performance problem according to the predefined severity threshold while the individual properties, i.e. *communication_costs*, *synchronization_costs*, and *io_costs*, may not be marked as performance problems.

3.4.2.4 communication_costs

```
float comm_costs(RegionSummary rs) = rs.sums.comm_time;

PROPERTY costs <comm_costs> communication_costs;
```

This property determines whether a region implies communication. Its condition and severity is based on *comm_time* which is the arithmetic mean across all processes executing region *r*. The severity is the

communication time divided by the execution time of the ranking basis. Again this property is defined based on template *costs*.

3.4.2.5 forall_synchronization_costs

```
PROPERTY forall_synchronization_costs (HPFRegion r, Experiment e, Region rank_basis){
LET
  float forall_sync_time = summary(r,e).sums.sync_time;
IN
  CONDITION:   typeof(summary(r,e).region)==HPFLoop AND
               summary(r,e).region.ltype == FORALL AND
               forall_sync_time > 0;
  CONFIDENCE:  1;
  SEVERITY:    forall_sync_time/duration(rank_basis,e);
}
```

FORALL loops may invoke synchronization within and between loop body statements which is specified by property *forall_synchronization_costs*. The severity is the fraction of the synchronization costs at the execution time of the ranking basis.

3.4.2.6 io_costs

```
float io_time(RegionSummary rs) = rs.sums.io_time;
```

```
PROPERTY costs <io_time> io_costs;
```

Property *io_costs* – defined based on template *costs* – of a region reflects whether or not any process spends some time in input/output operations. The severity is the fraction of the input/output costs at the execution time of the ranking basis.

3.4.2.7 parallel_organization_costs

```
PROPERTY parallel_organization_costs( Region r, Experiment e, Region
rank_basis) {

  CONDITION:  COUNT(procs WHERE procs IN summary(r,e).processes) > 1 ;

  CONFIDENCE: 1;

  SEVERITY:   summary(r,e).compiler_ovh_time / duration(rank_basis,e);
}
```

Execution of a parallel region may be associated with some extra costs implied by a parallelizing compiler. For instance, execution of a statement may be conditional depending on which process is executing the statement. The condition for this property is that a region is executed by more than process. The severity is the time needed to execute the extra code inserted by the compiler.

3.4.2.8 `redist_costs`

```

PROPERTY redist_costs (HPFRegion r,Experiment e,Region rank_basis) {
LET
redist1 = summary(r,e).sums.impl_redistr_time;
redist2 = summary(r,e).sums.expl_redistr_time;
IN
CONDITION:
  (Cond1)
    ((typeof(summary(r,e).region)==HPFProcedure)
     AND
     (EXISTS dec IN summary(r,e).region.decls
      SUCH THAT
       dec.format==PRESCRIPTIVE OR dec.format==TRANSCRIPTIVE)
     AND
     (redist1 > 0))
  OR
  (Cond2)
    ((typeof(summary(r,e).region)==HPFRedistribute)
     AND
     (EXISTS dec IN summary(r,e).region.decls
      SUCH THAT dec.alloc == DYNAMIC)
     AND
     (redist2 > 0))
CONFIDENCE: MAX((Cond1)->0.8, (Cond2)->1.0);
SEVERITY:   MAX((Cond1)->redist1 / duration(rank_basis,e),
               (Cond2)->redist2 / duration(rank_basis,e));
}

```

Property `redist_costs` specifies the time spent in redistributing data inside (redistribution of dummy arrays and dynamic arrays) or outside of procedures (redistribution of dynamic arrays). We include two different conditions for this property to hold each of which is associated with its own confidence value. The first condition ensures that a region is a procedure. If the procedure has prescriptive or transcriptive mapping and `impl_redistr_time` is non-zero then redistribution may occur at the procedure boundary. The second condition covers the case where DYNAMIC arrays are redistributed based on the HPF REDISTRIBUTE directive which causes `expl_redistr_time` to be non-zero. A close integration between performance measurement tool and HPF compiler is needed in order to associate measured performance data with the location in the HPF program that causes redistribution. Commonly, the more conditions are included for a property the more options are available to determine whether a property holds and the more refined information about the cause of the property is supplied. For instance, if prediction fails, then monitoring could be employed to prove a certain performance property. For property `redist_costs`, we include two conditions with more detailed information than if only a single value for all redistribution costs would be provided.

In the following code excerpt, the first and second loop nest, respectively, favor column- and row-wise distribution. An HPF REDISTRIBUTE directive is used to redistribute array `A` row-wise between the two loop nests.

```

!HPF$ DISTRIBUTE A(*,BLOCK)
DO i=1,n
  DO j=1,n
    A(i,j) = A(i-1,j)+A(i,j)
  END DO
END DO
!HPF$ REDISTRIBUTE(BLOCK,*) A(BLOCK,*)
DO i=1,n
  DO j=1,n
    A(i,j) = A(i,j-1)+A(i,j)
  END DO
END DO

```

```

        END DO
    END DO

```

Clearly, for this example array redistribution enables perfect load balance with no communication inside of loops. However, assume that the second loop is insignificant in terms of computational overhead or is not executed at all. Then array redistribution may imply higher communication costs than performance that can be gained.

The confidence of property *redist_costs* is the maximum confidence value across both conditions as specified above. Note that for this property we assume that measuring implicit redistribution (condition 1) may be less precise than timings of explicit redistribution (condition 2) due to a lack of compiler information. The severity is the maximum across the time spent in, respectively, implicit and explicit redistribution of data relative to the execution time of the region selected as the rank basis.

3.4.2.9 serialization_costs

```

PROPERTY serialization_costs (HPFRegion r, Experiment seq, Experiment par, Region rank_basis){

LET
    float par_comp_costs = summary(r,par).sums.duration -
                          summary(r,par).sums.comm_time -
                          summary(r,par).sums.sync_time -
                          summary(r,par).sums.idle_time -
                          summary(r,par).sums.compiler_ovh_time -
                          summary(r,par).sums.io_time;

IN
    CONDITION:  par_comp_costs > (duration(r,seq) * loop_serial_threshold)
    CONFIDENCE:  1;
    SEVERITY:    par_comp_costs / (duration(rank_basis,par));
}

```

Property *serialization_costs* reflects whether parallelism has been exploited by a given region. *seq* and *par*, respectively, correspond to a single and multiprocessor experiment.

par_comp_costs defines the parallel computation costs implied by a region which excludes communication, synchronization, idle, compiler overhead, and input/output time. This figure is then compared against the sequential execution time. The severity is given by *par_comp_cost* divided by the parallel execution time of the *rank_basis*.

3.4.2.10 uneven_work_distribution

Property `uneven_work_distribution` (HPFRegion `r`, Experiment `seq`, Experiment `par`, Region `rank_basis`) {

LET

```
int nr_processes = COUNT(procs WHERE procs IN summary(r,par).processes);
float opt_duration = duration(r,seq)/nr_processes;
```

```
float deviation = SQRT(SUM (EXP(proc_sum.duration -
    proc_sum.comm_time -
    proc_sum.sync_time -
    proc_sum.idle_time -
    proc_sum.compiler_ovh_time -
    proc_sum.io_time - opt_duration, 2)
    WHERE proc_sum IN summary(r,par).proc_sums ))
```

IN

```
CONDITION: (deviation / opt_duration) > uneven_threshold
CONFIDENCE: 1;
SEVERITY: summary(r,par).sums.duration / duration(rank_basis,par)
```

}

Property *uneven_work_distribution* specifies how even the computations of a parallel program have been distributed across all processes executing a region. The standard deviation of the computational costs of every process with respect to the optimal duration (sequential execution time divided by number of processes) is computed. The condition is then given as the variation coefficient compared against a threshold. The severity is defined as the execution time divided by the *rank_basis*.

3.4.2.11 inspector_costs

```
float inspector_time(RegionSummary rs) = rs.sums.inspector_time;
```

```
PROPERTY costs <inspector_time> inspector_costs;
```

One parallelization strategy for irregular HPF programs implies for each loop (region) a preprocessing (inspector) and an executor phase. The inspector phase – commonly highly execution time intensive – is responsible for the analysis of access patterns and calculation of communication schedules. The executor phase gathers remote data, executes the loop and scatters data to the owning processes. A crucial aspect of this parallelization strategy deals with the problem to reuse the communication schedule of the inspector phase which in many cases is loop invariant.

Property *inspector_costs* denotes the average time spent in the inspector phase across all involved processes. This property is again defined based on template *costs* with the given severity and confidence figures.

3.4.3 OpenMP

This section introduces performance properties of shared memory programs. The following performance properties are presented:

- costs
- measurable_costs
- unmeasurable_costs
- non_parallelized_code

- synchronization
- irregular_sync_across_instances
- load_imbalance
- remote_accesses
- remote_access_to_variable
- multiple_transfer_of_same_data
- wrong_page_distribution_for_variable
- parallel_organization

3.4.3.1 OpenMP global definitions

```

SmRegionSummary summary(Region r, Experiment e)=
    UNIQUE({s IN e.profile WITH s.region==r});

float sync(Region r, Experiment e)=summary(r,e).sums.region_wait +
    summary(r,e).sums.region_ctrl +
    summary(r,e).sums.cross_thread_dep_wait +
    summary(r,e).sums.cross_thread_dep_ctrl;

float duration(Region r, Experiment e)=summary(r,e).sums.duration;

float remote_access_time(Region r, Experiment e)=
    summary(r,e).sums.nr_remote_accesses
    * e.system.remote_access_time;

```

The following property specifications make use of those four functions. The *summary* function determines the summary information for a given region and a given experiment.

The *sync* function determines the overhead for synchronization in a given region. It computes the sum of the relevant attributes in the summary class.

The *duration* function returns the execution time of a region. The execution time is determined by the execution time of the master thread in the OpenMP model.

The *remote_access_time* function estimates the overhead for accessing remote memory based on the measured number of accesses and the mean access time of the parallel machine.

In the following we define two templates for generic costs. One to be used for parallel costs which are evaluated relative to a serial execution which, therefore requires both a sequential and a parallel experiment, and one for costs which are evaluated with reference to a single (parallel) experiment. These can be used to define various OpenMP performance properties.

3.4.3.2 Template costs

```

PROPERTY TEMPLATE costs <float cost_func(Region r, Experiment e)>
    (Region r, Experiment e, Region RankBasis){
LET
    cost = cost_func(r,e)
IN
    CONDITION:    cost > 0;
    CONFIDENCE:   1;
    SEVERITY:     cost/duration(rank_basis,e);
}

```

3.4.3.3 template rel_costs

```

PROPERTY TEMPLATE rel_costs
  <float cost_func(Region r, Experiment seq,Experiment par)>
  (Region r, Experiment seq, Experiment par, Region RankBasis){
LET
  cost = cost_func(r,seq,par)
IN
  CONDITION:  cost > 0;
  CONFIDENCE: 1;
  SEVERITY:   cost/duration(rank_basis,par);
}

```

3.4.3.4 costs

```

float par_costs(Region r, Experiment seq, Experiment par) = duration(r,par) -
  (duration(r,seq)/par.nr_processors);

PROPERTY rel_costs <par_costs> costs;

```

This property specifies that the speedup of the application is not linear. It uses information from two experiments, a sequential run and a parallel run, to compute the costs of parallel execution. Those costs determine the severity of the property.

3.4.3.5 measurable_costs

```

float par_costs(Region r, Experiment e) =
  summary(r,e).sums.non_parallelized_code +
  sync(r,e) +
  remote_access_time(r,e) +
  summary(r,e).sums.scheduling +
  summary(r,e).sums.additional_calc;

PROPERTY costs <par_costs> measurable_costs;

```

Performance analysis tools only help in analyzing measurable costs. A region has the *measurable_costs* property if the sum of those costs is greater than zero. The severity of this property is the fraction of those costs relative to the execution time of *rank_basis*.

3.4.3.6 unmeasurable_costs

```

Property unmeasurable_costs(Region r, Experiment seq, Experiment e, Region rank_basis){
LET
  float total_costs = duration(r,e) - (duration(r,seq)/e.nr_processors);
  float costs = summary(r,e).sums.non_parallelized_code +
                sync(r,e) +
                remote_access_time(r,e) +
                summary(r,e).sums.scheduling +
                summary(r,e).sums.additional_calc;
IN
  CONDITION: total_costs-costs>0;

  CONFIDENCE: 1;

  SEVERITY: total_costs(r,e)/duration(rank_basis,e);
}

```

The total cost of the parallel program is the sum of the measurable and the unmeasurable overhead. The *unmeasurable_costs* property determines whether an unmeasurable overhead exists. Its severity is the fraction of this overhead in relation to the execution time of *rank_basis*. If this fraction is high, further tool-supported performance analysis might not be very helpful.

3.4.3.7 non_parallelized_code

```

float non_parallel_code(Region r, Experiment e) =
    summary(r,e).sums.non_parallelized_code;

PROPERTY costs <non_parallel_code> non_parallelized_code;

```

Non-parallelized code is a very severe problem for application scaling. In the context of analyzing a given program run, its severity is determined in the usual way. If the focus of the analysis is more on application scaling the severity should stress the importance of this property.

3.4.3.8 synchronization

```

PROPERTY costs <sync> synchronization;

```

This property uses the globally defined function *sync* defined in Section 3.4.3.1.

A region has the *synchronization* property if any synchronization overhead occurs during its execution. One of the obvious reasons for high synchronization cost is load imbalance.

3.4.3.9 irregular_sync_across_instances

```
Property irregular_sync_across_instances
  (Region r, Experiment e, Region rank_basis){
LET
  float inst_sync(SmInstanceSums sum)=sum.region_wait +
                                     sum.region_ctrl +
                                     sum.cross_thread_dep_wait +
                                     sum.cross_thread_dep_ctrl ;

IN
  CONDITION: stdev(inst_sync(inst_sum)
                  WHERE inst_sum IN summary(r,e).instance_sums)
             > irreg_behaviour_threshold * sync(r,e)/r.nr_executions;

  CONFIDENCE: 1;

  SEVERITY: sync(r,e)/duration(rank_basis,e);
}
```

The *synchronization* property defined above is assigned to regions with synchronization. If the dynamic behaviour of an application changes over the execution time - load imbalance, for example, might occur only in specific phases of the simulation - the whole synchronization overhead might result from specific instances of the region. A region with the *irregular_sync_across_instances* property has an irregular distribution of the synchronization overhead across different instances. The severity is equal to the severity of the synchronization property since the *irregular_sync_across_instances* property is only a more detailed explanation.

3.4.3.10 load_imbalance

```
float load_imb_cost(Region r, Experiment e) =
  summary(r,e).sums.region_wait;

PROPERTY costs <load_imb_cost> load_imbalance;
```

Work is unevenly distributed to threads in the region. This manifests itself in *region_wait* time. If the *region_wait* time cannot be measured, the property can also be proven based on the execution time of the thread with the longest duration minus the average duration.

3.4.3.11 remote_accesses

```
PROPERTY costs <remote_access_time> remote_accesses;
```

An important property for code on ccNUMA machines is access to remote memory. Remote memory access implements communication among parallel threads. Since usually only the number of accesses can be measured, the severity is estimated based on the mean access time. This property uses the global function `remote_access_time` defined in Section 3.4.3.1.

3.4.3.12 remote_access_to_variable

```

Property remote_access_to_variable
  (Region r, Experiment e, String var, Region rank_basis)
{
  LET
    VariableRemoteAccesses var_sum =
      UNIQUE({info IN summary(r,e).sums.accessed_variables
              WITH info.var_name==var});
  IN
    CONDITION: var_sum.nr_remote_accesses > 0;

    CONFIDENCE: 1;

    SEVERITY: var_sum.nr_remote_accesses * e.system.remote_access_time
              /duration(rank_basis,e);
}

```

The previous property identifies regions with remote accesses. This property is more specific since its context also includes a specific variable. The property indicates whether accesses to a variable in this region result in remote accesses. It is based on address-range-specific remote access counters, such as the counters provided in the SGI Origin 2000 on page level. The severity of this property is based on the time spent in remote accesses to this variable. Since this property is very useful in explaining a severe remote access overhead for the region, it might be ranked with respect to this region during a more detailed analysis.

3.4.3.13 multiple_transfer_of_same_data

```

Property multiple_transfer_of_same_data
  (Region r, Experiment e, String var, Region rank_basis)
{
  LET
    VariableRemoteAccesses var_info =
      UNIQUE({info IN summary(r,e).sums.accessed_variables
              WITH info.var_name==var});
  IN
    CONDITION: (Cond1)
      (var_info.nr_remote_accesses * e.nr_processors > var_info.size
       / e.system.cache_line_size*summary(r,e).nr_executions)
    OR
      (Cond2)
      (var_info.nr_remote_accesses * e.nr_processors
       > 0.5 * var_info.size / e.system.cache_line_size
       * summary(r,e).nr_executions);

    CONFIDENCE: MAX( (Cond1)->1, (Cond2)->0.5 );

    SEVERITY: var_info.nr_remote_accesses * e.system.remote_access_time
              /duration(rank_basis,e);
}

```

This property indicates that the same coherence unit, e.g. cache line in ccNUMA-systems, is transferred multiple times between processors. Some of the transfers might be unnecessary.

The conditions in this property compare the number of remote accesses in all threads to the number of cache lines in the data structure. Is the number of remote accesses is larger than the number of cache lines multiplied by the number of executions of that region, i.e. the maximum number of accesses without transferring elements twice, the property is proven with a confidence of one. Is the number of remote accesses only larger than half the potential accesses (Cond2) the confidence is 0.5.

The severity is the same in both cases, i.e. the estimated loss in execution time.

3.4.3.14 wrong_page_distribution_for_variable

```
Property wrong_page_distribution_for_variable
  (Region r, Experiment e, String var, Region rank_basis)
{
  LET
    VariableRemoteAccesses var_sum(String var, setof VariableRemoteAccesses info_set) =
      UNIQUE({info IN info_set WITH info.var_name==var});
  IN
    CONDITION:
      EXISTS
        thr_sum   IN summary(r,e).thread_sums,
        thr_psum  IN var_sum(var,thr_sum.accessed_variables).page_sums,
        glo_psum  IN {s IN var_sum(var,summary(r,e).accessed_variables).page_sums
                      WITH s.page_no==thr_psum.page_no}
      SUCH THAT
        thr_psum.nr_remote_accesses!=0
      AND
        thr_psum.nr_remote_accesses==glo_psum.nr_remote_accesses;

    CONFIDENCE: 0.5;

    SEVERITY:  var_sum(var,summary(r,e).accesses_variables).nr_remote_accesses
              * e.system.remote_access_time/duration(rank_basis);
}
```

The property identifies a specific reason for remote memory accesses. The condition checks whether remote accesses to at least one of the pages of the variable occur in only a single process. The remote access counts of the other threads for this page have to be zero.

This condition is only an indication for a possibly wrong page distribution. The confidence value is lower than one since it might be possible that another thread executing in the node where the page is allocated accesses this page. If the page would be migrated, remote accesses would then occur for this thread. Since local access counts are not available in that data model, a more precise condition cannot be determined.

The severity of the property depends on the average number of remote accesses in the threads.

3.4.3.15 parallel_organization

```
Property parallel_organization( Region r, Experiment e, Region rank_basis) {
  CONDITION:  typeof(summary(r,e).region)==ParallelRegion
              AND summary(r, e).nr_executions > 1 ;

  CONFIDENCE: 1;

  SEVERITY:  summary(r,e).nr_executions *
              e.system.parallel_region_cost / duration(rank_basis,e);
}
```

Execution of a parallel region is associated with costs for "going parallel". The severity of these costs depends on the number of instances of the parallel region.

The condition for this property is that the number of executions for a region is greater than one. The severity is the time associated with initiating a parallel regions (sum of *region_ctrl* time) relative to the execution time of the region selected as the ranking basis. It is assumed that the average cost associated with a parallel region (basically the barrier time) is stored as a machine constant (possibly dependent upon p , the number of processors).

Chapter 4

Conclusions and Future Work

This report has described the specification language (ASL) that will be used in the APART working group to describe performance problems in parallel programs. ASL provides constructs to specify performance-related data as an object model, and constructs to describe performance properties, including conditions to prove existence, confidence expressions to support fuzzy information, and severity measures which allows the ranking of performance problems. ASL has been extended by new concepts that include templates and metaproperties. Templates are used to describe similar performance properties with a generic representation. Moreover, in order to support the specification of performance properties based on other performance properties, we invented metaproperties. By incorporating this concept we can formulate properties based on performance-related data and components (condition, confidence, and severity) of existing performance properties. ASL has been further extended by patterns to describe compound events that indicated the existence of performance properties which can only be detected in event traces.

The examples presented in this report are data models and performance properties of implementations of three major programming paradigms: MPI, HPF, and OpenMP. Currently, for a specific performance analysis environment on a specific parallel machine, specialized specifications have to be developed since the performance-related data available in the environment must be taken into account.

Since the data models for the three paradigms do have a common structure, and this common structure will very likely show up in real performance analysis environments, it is covered by a set of base classes that can be reused in new designs. The list of base classes will be extended in the future to cover other common aspects, such as classes representing typical regions, and classes for a standard set of trace events.

Also, in the future, we would like to test the specification techniques in the context of other programming paradigms, such as object oriented programming, distributed applications, multimedia applications, and databases. We would like to verify whether the language is powerful enough to describe the performance properties found in these environments.

The specification language presented in this report will have to be supplemented by other specification notations. For example, a notation is required to describe the data supplied by existing analysis tools in the target performance analysis environment. Also required is a notation to provide the means for specifying the analysis process of automated performance analysis tools. Both specifications have been introduced in Section 1.2. Besides the coordinated design of these languages, the issue of the efficient translation of the specifications into data and/or code for use by automated analysis tools needs to be investigated. We anticipate that this latter topic might well lead to a design for all the languages that is tailored more towards tool implementation.

Bibliography

- [AhSeUl 88] A. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques and Tools*. Series in Computer Science. Addison Wesley, 1988.
- [BaWi 83] P. Bates, J.C. Wileden: *High-Level Debugging of Distributed Systems: The Behavioral Abstraction Approach*, The Journal of Systems and Software, Vol. 3, pp. 255-264, 1983
- [BeGeKr 96] R. Berrendorf, M. Gerndt, A. Krumme: *A Programming Environment for Parallel Computers with Global Address Space*, Workshop on High-Level Programming Models and Supportive Environments (HIPS '96), in combination with IPPS '96, IEEE, pp. 10-16, 1996
- [CuSiGu 99] D. E. Culler and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publisher Inc., 1999.
- [DaMe 98] Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46-55, January/March 1998.
- [EsMaLu 98] A. Espinosa, T. Margalef, E. Luque: *Automatic Performance Evaluation of Parallel Programs*, Sixth Euromicro Workshop on Parallel and Distributed Processing, 1998
- [FaScPa 99] Thomas Fahringer, Bernhard Scholz, and Mario Pantano. Execution-Driven Performance Analysis for Distributed and Parallel Systems. Institute for Software Technology and Parallel Systems, University of Vienna, Liechtensteinstr. 22, A-1090 Wien. Technical Report, June, 1999.
- [Fa 95] T. Fahringer. Estimating and Optimizing Performance for Parallel Programs. *IEEE Computer*, 28(11), pp. 47-56, Nov. 1995.
- [Fa 96] T. Fahringer. Automatic Performance Prediction of Parallel Programs. Kluwer Academic Publishers, Boston, USA, ISBN 0-7923-9708-8, March, 1996.
- [GeKr 97] M. Gerndt, A. Krumme: *A Rule-based Approach for Automatic Bottleneck Detection in Programs on Shared Virtual Memory Systems*, Second Workshop on High-Level Programming Models and Supportive Environments (HIPS '97), in combination with IPPS '97, IEEE, 1997
- [GeKrOz 95] M. Gerndt, A. Krumme, S. Özmen: *Performance Analysis for SVM-Fortran with OPAL*, Proceedings Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'95), Athens, Georgia, pp. 561-570, 1995
- [HeMa 95] B.R. Helm, A.D. Malony: *Automating Performance Diagnosis: a Theory and Architecture*, International Workshop on Computer Performance Measurement and Analysis (PERMEAN '95), 1995
- [HPF 93] High Performance FORTRAN Language Specification. Technical Report, Version 2.0.δ, Rice University, Houston, TX, October 1996.
- [KaLeObWa 98] Hockauf, R.; Karl, W.; Leberecht, M.; Oberhuber, M.; Wagner, M.: Exploiting Spatial and Temporal Locality of Accesses: A New Hardware-Based Monitoring Approach for DSM Systems. In: D. Pritchard, Jeff Reeve (Eds.): Euro-Par'98 Parallel Processing / 4th International Euro-Par Conference Southampton, UK, September 1-4, 1998 Proceedings. Springer-Verlag, Heidelberg, Lecture Notes in Computer Science Vol.1470, 1998, pp. 206-215

- [MCCHI 95] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, T. Newhall: *The Paradyn Parallel Performance Measurement Tool*, IEEE Computer, Vol. 28, No. 11, pp. 37-46, 1995
- [GaMo 98] J. Galarowicz, B. Mohr: *Analyzing Message Passing Programs on the Cray T3E with PAT and VAMPIR*, Fourth European CRAY-SGI MPP Workshop Garching/München, Research Centre Juelich Technical Report, FZJ-ZAM-IB-9809, 1998
- [MoBrMa 94] B. Mohr and D. Brown and A. Malony. TAU: A portable parallel program analysis environment for pC++. CONPAR, Linz, Austria, 1994.
- [NaArWeHoSo 96] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach VAMPIR: Visualization and Analysis of MPI Resources. Supercomputer 12(1), pp. 69-80, 1996.
- [Paradyn 98] Paradyn Project: *Paradyn Parallel Performance Tools: User's Guide*, Paradyn Project, University of Wisconsin Madison, Computer Sciences Department, 1998
- [Mu 99] Nandini Mukhopadhyay (Mukherjee). On the effectiveness of feedback-guided parallelisation. PhD Thesis, University of Manchester, Department of Computer Science, September, 1999.
- [Mu 00] N. Mukherjee, G.D. Riley and J.R. Gurd. FINESSE: A Prototype Feedback-guided Performance Enhancement System, Accepted for PDP2000, to be held in Rhodes in January 2000.
- [Re 93] D. A. Reed, R. A. Ayt, R. J. Noe, P.C. Roth, K. A. Shields, B. W. Schwartz, L.F. Tavera. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In Proc. of Scalable parallel Libraries Conf., Ieee Computer Society, pp. 104-113, 1993.
- [Re 98] R. Ribler, J. Vetter, H. Simitci, D. A. Reed. Autopilot: Adaptive Control of Distributed Applications. In Proc. of the 7th IEEE Symposium on High-Performance Distributed Computing, 1998.
- [RuJaBo 99] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison Wesley Longman, Reading, Mass., 1999.
- [SnOtHUVaDo 98] M. Snir, St. Otto, St. Huss-Lederman, D. Walker, J. Dongarra: *MPI - The Complete Reference*, MIT Press, ISBN 0-262-69216-3, 1998
- [WoMo 98] F. Wolf, B. Mohr: *EARL - A Programmable and Extensible Toolkit for Analyzing Event Traces of Message Passing Programs*, 7th International Conference on High-Performance Computing and Networking (HPCN'99), A. Hoekstra, B. Hertzberger (Eds.), Lecture Notes in Computer Science, Vol. 1593, pp. 503-512, 1999
- [WoMo 00] F. Wolf, B. Mohr: *Specifying Performance Properties Using Compound Runtime Events*, Technical Report, ESPRIT IV Working Group APART, August 2000.

Appendix A

Unified Modeling Language Class Diagrams

The object models are presented as Unified Modelling Language (UML) class diagrams [RuJaBo 99],

<http://www.rational.com/uml>

and as textual property specification documents. Figure A.1 shows a simple UML class diagrams. The boxes represent classes, the class name is shown in the upper part and the class's attributes in the lower part. The attributes are represented by its name followed by its type. Closed arrows represent the specialization of generalization relationship. For example, SaloonCar and Caravan are a specializations or subclasses of Car.

UML diagrams provide two other types of relationships: associations and aggregations. Associations are represented by plain lines and open arcs. In contrast to a plain line, an arc defines that the association is navigable in this direction. The name for accessing the associated object is written near to this object, in the diagrams in Figure A.1 the potential drivers of a car are identified via Drivers. In addition, the cardinality of the associated objects can be specified at the end points of an association or aggregation. For example, each car can have 1 or more potential drivers and each person can drive 0 or more cars.

Aggregations are represented as lines or arcs with a diamond near to the aggregate class. An aggregation is a *has a* relationship. The diagrams in Figure A.1 specifies that a car has four wheels. An aggregation relationship is used instead of an association if the destruction of the aggregate object also leads to the destruction of the attribute objects.

The UML is a very powerful specification language. The above paragraphs only explain those feature that are used in the performance model diagrams. When designing and implementing software, UML

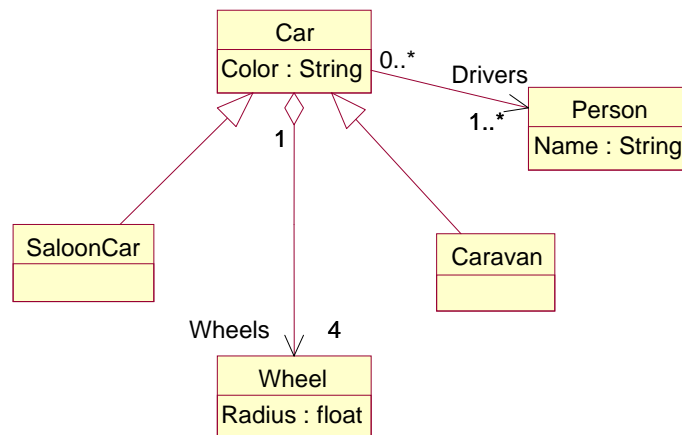


Figure A.1: Object model in the Unified Modelling Language (UML)

specification are automatically translated into appropriate programming language constructs, such as classes in Java or C++ and SQL specifications for relational databases. In a similar procedure the diagrams of performance-related data models can be transferred into the APART specification language.

Appendix B

APART Base Class Library

```
//*****  
//  
// APART Base Classes  
//  
//*****  
  
class Application {  
    String name;           //Name of application  
    setof Version versions; //Versions  
  
}  
  
class Version {  
    int version_no;       //Version number  
    setof SourceFile files; //Source files  
    setof Experiment experiments; //Multiple experiments  
  
}  
  
// Classes SourceFile and Region model pure static information  
  
class SourceFile {  
    String name;           //File name  
    String contents;      //File contents  
    setof Region regions; //Regions included in file  
  
}  
  
class Region {  
    Position start_pos;    //Start position (line, column) in file  
    Position end_pos;     //End position  
    setof Region sub_regions; //Regions nested in that region  
    setof Region successors; //Successor regions according to region  
    setof Region predecessors; //Predecessor regions according to region  
    //control flow  
  
}  
  
//The following classes are used to model dynamic performance  
//information. It includes the summary data per region  
//(entire run, all processes) and the traces of the processes.
```

```

class Experiment {
    DateTime start_time;           //Start time of the experiment
    int nr_processors;             //Number of used processors
    setof RegionSummary profile;   //Summed up information for events
    setof Event trace;             //Trace records for individual events
    Machine system;                //Machine chracterization
}

class Event {
    float timestamp;               //Timestamp of event
    Process process_id;            //Process number
}

class RegionSummary {
    Region region;                 //Region with measured data
}

//The following classes are utility classes for the different
//programming paradigms.

class Dependence {
    Region src;                    //Source of dependence
    Region dst;                    //Destination of dependence
    GeneralTypes.dep_type type;    //Type of data dependence
    GeneralTypes.dep_dir direction; //Direction of dependence
    int distance;                  //Distance of dependence
    int level;                     //Loop level that carries dependence
}

class GeneralTypes{
    enum dep_dir { '<', '>', '='); //Direction of dependence
    enum dep_type{ True, Anti, Output}; //Type of dependence
}

class LoopHeader {
    String lower;                  //Describes loop bounds
    String upper;                  //Lower bound
    String stride;                 //Upper bound
}

class Position {
    int line;                       //Stride of loop
    int col;                         //Line in file
}

class Process {
    int process_id;                 //Column in that line
}

class Machine {
    int nr_processors;              //Process number
}

class Machine {
    int nr_processors;              //Generic parallel machine
}

```

Appendix C

MPI Property Specification

```
//*****  
// APART Example Property Specification  
//  
// Message Passing Paradigm Performance Data Model  
//  
//*****  
  
PERFORMANCE DATA  
  
//*****  
//MPI static information  
//*****  
  
class MPIApplication extends Application {  
    MPITypes.ParadigmType Paradigm;    //defines implemented paradigm  
}  
  
class Function extends Region {  
    int NumCallSites;                //call sites in function  
    String Name;                      //function name  
    Setof FuncCall Calls;            //call sites in function  
}  
  
class FuncCall extends Region {  
    Function CalledFunc;              //called function  
    Function Caller;                  //calling function  
    MPITypes.ParadigmRole role;      //role of this region in paradigm  
}  
  
class MPIFunction extends Function {  
}  
  
class MPICommFunc extends MPIFunction {  
}  
  
class MPISyncFunc extends MPIFunction {  
}  
  
class MPIioFunc extends MPIFunction {  
}
```

```

class MPIp2pFunc extends MPICommFunc {
    MPITypes.CommType type;           //e.g. send, receive
    MPITypes.CommMode mode;          //buffered, synchronous, ready
    MPITypes.CommSemantics semantics; //blocking, nonblocking
}

class MPICollFunc extends MPICommFunc {
    MPITypes.CollType type;          //reduction, broadcast
}

/*****
//MPI dynamic information: Summary data
*****/

class MPISummary extends RegionSummary {
    Process Proc;                    //process or ALLPROCESSES
    float Excl;                      //exclusive execution time
    float Incl;                      //inclusive execution time
    float CommTime;                  //communication time
    float SyncTime;                  //synchronization time
    float IoTime;                    //IO time
    float IdleTime;                  //idle time in communication
    int MessageLength;               //sum of sent/received messages
    int NrExecutions;                //number of region instances
    int L1DCacheMisses;              //level 1 cache misses
    int L1ICacheMisses;              //instruction cache misses
    int L2CacheMisses;              //level two cache misses
    int Instinstr;                   //integer instructions
    int FpInstr;                     //floating point instructions
    int LoadInstr;                   //load instructions
    int StoreInstr;                  //store instructions
}

/*****
//MPI dynamic information: Events
*****/

class Enter extends Event {
    Region region;                   //entered region
}

class Exit extends Event {
    Region region;                   //left region
    float Excl;                      //exclusive execution time
    float Incl;                      //inclusive execution time
    float CommTime;                  //communication time
    float SyncTime;                  //synchronization time
    float IoTime;                    //IO time
    float IdleTime;                  //idle time in communication
    int MessageLength;               //sum of sent/received messages
    int NrExecutions;                //number of region instances
    int L1DCacheMisses;              //level 1 cache misses
    int L1ICacheMisses;              //instruction cache misses
    int L2CacheMisses;              //level two cache misses
}

```

```

    int InstInstr;           //integer instructions
    int FpInstr;            //floating point instructions
    int LoadInstr;         //load instructions
    int StoreInstr;        //store instructions
}

class Send extends Event {
    int Tag;                //message tag
    int Length;            //message length
    Process Destination;    //target process
}

class Receive extends Event {
    int Tag;                //message tag
    int Length;            //message length
    Process Sender;        //source process
}

class Collective extends Event {
    Communicator Communic; //active communicator
    int Length;            //total length of all messages
}

class Communicator {
    setof Process processes;
}

//*****
//MPI utility classes
//*****

class MPITypes {
    enum CommMode {Buffered, Sync, Ready};
    enum CollType {Reduction, Broadcast};
    enum CommType {Send, Recv, SendRecv};
    enum CommSemantics {Blocking, Nonblocking};
    enum ParadigmType {MasterSlave, DivideConquer, Farming};
    enum ParadigmRole {MasterSend, MasterRecv, SlaveSend, SlaveRecv};
}

//*****
// APART Example Property Specification
//
// Message Passing Paradigm Performance Properties
//
//*****

PERFORMANCE PROPERTIES

LET

MPISummary Summary(MPIRegion r, Experiment e, Process p)=
    UNIQUE({sumr IN e.profile WITH sumr.region==r AND sumr.Proc==p});

Setof MPISummary ProcessProfiles(MPIRegion r, Experiment e)=
    {sumr IN e.profile WITH sumr.region==r AND sumr.Proc!=ALLPROCESSES}

```

```

float Duration(MPIRegion r, Experiment e)= Summary(r,e).Incl;

float Duration(MPIRegion r, Experiment e, Process p)= Summary(r,e,p).Incl;

IN
PROPERTY TEMPLATE Cost <float CostFunc(Region r, Experiment e)>
    (Region r, Experiment e, Region RankBasis){
LET
    Cost = CostFunc(r,e)
IN
    CONDITION:    Cost > 0;
    CONFIDENCE:   1;
    SEVERITY:     Cost/Duration(RankBasis,e);
}

float AllCostFunc(Region r, Experiment e) =
    summary(r,e).CommTime +
    summary(r,e).SyncTime +
    summary(r,e).IoTime;

float CommCostFunc(Region r, Experiment e)= Summary(r,e).CommTime;
float SyncCostFunc(Region r, Experiment e)= Summary(r,e).SyncTime;
float IoCostFunc(Region r, Experiment e)= Summary(r,e).IoTime;

PROPERTY Cost<AllCostFunc> TotalCost;
PROPERTY Cost<CommCostFunc> CommCost;
PROPERTY Cost<SyncCostFunc> SyncCost;
PROPERTY Cost<IoCostFunc> IoCost;

PROPERTY TEMPLATE CacheCost <float CostFunc(Region, Experiment)>
    (Region r, Experiment e, Region RankBasis) {
LET
    float Costs=CostFunc(r,e);
IN
    CONDITION: Costs>0;
    CONFIDENCE: 1;
    SEVERITY: Cost/(Summary(RankBasis,e).LoadInstr
        + Summary(RankBasis,e).StoreInstr) ;
}

L1DCacheCostFunc(Region r, Experiment e)= Summary(r,e).L1DCacheMisses;
L1ICacheCostFunc(Region r, Experiment e)= Summary(r,e).L1ICacheMisses;
L2CacheCostFunc(Region r, Experiment e)= Summary(r,e).L2CacheMisses;

PROPERTY CacheCost<L1DCacheCostFunc> L1DataCacheMisses;
PROPERTY CacheCost<L1ICacheCostFunc> L1InstCacheMisses;
PROPERTY CacheCost<L2CacheCostFunc> L2CacheMisses;

PROPERTY DominatingCommFunction(MPIFunction r, Experiment e, Region RankBasis){
LET
    setof MPISummary CommSummaries=
        {x IN e.profile
        WITH
            x.Proc==ALLPROCESSES
        AND

```



```

        typeof(x.region)==MPICommFunc};
float MaxCommTime = MAX(sum.Incl WHERE sum IN CommSummaries);
IN
CONDITION: typeof(r)==MPICommFunc AND Duration(r,e)==MaxCommTime;
CONFIDENCE: 1;
SEVERITY: MaxCommTime/Duration(RankBasis,e);
}

PROPERTY DominatingCommCall(FuncCall r, Experiment e, Region RankBasis){
LET
  setof MPISummary CommSummaries=
    {x IN e.profile
     WITH
      x.Proc==ALLPROCESSES AND typeof(x.region)==FuncCall
     AND
      typeof(x.region.CalledFunc)==MPICommFunc};
float MaxCommTime = MAX(sum.Incl WHERE sum IN CommSummaries);
IN
CONDITION: typeof(r.CalledFunc)==MPICommFunc AND
           Duration(r,e)==MaxCommTime;
CONFIDENCE: 1;
SEVERITY: MaxCommTime/Duration(RankBasis,e);
}

PROPERTY FrequentCommunication (FuncCall r, Experiment e, Region RankBasis){
LET
  float Cost = Duration(r,e);
IN
CONDITION: typeof(r.CalledFunc)==MPICommFunc AND Cost>0 AND
           Cost/Summary(r,e).NrExecutions<SmallMessagesThreshold;
CONFIDENCE: 1;
SEVERITY: Cost/Duration(RankBasis,e);
}

PROPERTY BigMessages (FuncCall r, Experiment e, Region RankBasis){
LET
  float Cost = Summary(r,e).CommTime;
  int AvgLength = summary(r,e).MessageLength/
                 summary(r,e).NrExecutions;
IN
CONDITION: typeof(r.CalledFunc)==MPICommFunc AND Cost>0 AND
           AvgLength>BigMessagesThreshold;
CONFIDENCE: 1;
SEVERITY: Cost/Duration(RankBasis,e);
}

PROPERTY LateSender(FuncCall r, Experiment e, Region RankBasis){
LET
  float IdleTime = summary(r,e).IdleTime;
IN
CONDITION: typeof(r.CalledFunc)==MPICommFunc
           AND r.CalledFunc.Name == MPI_Receive AND IdleTime>0;
CONFIDENCE: 1;
SEVERITY: IdleTime/Duration(RankBasis,e);
}

```

```

PROPERTY LateReceiver(FuncCall r, Experiment e, Region RankBasis){

LET
  float IdleTime = Summary(r,e).IdleTime;
IN
  CONDITION: typeof(r.CalledFunc)==MPIp2pFunc
             AND r.CalledFunc.Name == MPI_SEND
             AND r.Semantics == Blocking AND idle_time>0;
  CONFIDENCE: 1;
  SEVERITY: IdleTime/Duration(RankBasis,e);
}

PROPERTY UnevenMpDistribution(FuncCall r, Experiment e,
                              Region RankBasis)
{
LET
  float deviation=stdev(s.Incl WHERE s IN ProcessProfiles(r,e));
IN
  CONDITION: typeof(r.CalledFunc)==MPICommFunc AND
             deviation > uneven_threshold * summary(r,e).Incl/
             e.nr_processors;
  CONFIDENCE: 1;
  SEVERITY: summary(r,e).Incl/Duration(RankBasis,e);
}

PROPERTY LoadImbalanceAtBarrier(FuncCall r, Experiment e, Region RankBasis){
LET
  float MaxTime=max( x.Incl WHERE x IN ProcessProfiles(r,e) );
  float MinTime=min( x.Incl WHERE x IN ProcessProfiles(r,e) );
  float MaxWait=MaxTime - MinTime;
IN
  CONDITION: (COND1) r.CalledFunc.Name==MPI_Barrier AND
             max_wait>0
             || (COND2) r.CalledFunc.Name==MPI_Barrier AND
             summary(r,e).IdleTime>0;
  CONFIDENCE: 1;
  SEVERITY: MAX((COND1)->MaxWait/(Duration(RankBasis,e)/e.nr_processors),
                (COND2)->Summary(r,e).IdleTime/Duration(RankBasis,e));
}

PROPERTY SlowSlaves (FuncCall r, MPIApplication a,
                    Experiment e, Region RankBasis){

LET
  float IdleTime = summary(r,e).IdleTime;
IN
  CONDITION: a.Paradigm == MasterSlave AND r.Role == ReceiveMaster AND
             IdleTime>0;
  CONFIDENCE: 1;
  SEVERITY: IdleTime/Duration(RankBasis,e),
}

PROPERTY OverloadedMaster(FuncCall r, MPIApplication a,
                          Experiment e, Region RankBasis){

LET
  float IdleTime = summary(r,e).IdleTime/(e.NrProcessors-1);

```

```

IN
  CONDITION: (a.Paradigm == MasterSlave AND
             (r.Role == ReceiveSlave OR r.Role == SendSlave)) AND
             IdleTime>0;
  CONFIDENCE: 1;
  SEVERITY: IdleTime/Duration(RankBasis,e);
}

PATTERN WrongOrder(FuncCall r) {

  ROOTTYPE: Receive;
  INSTANTIATION:
    Send  SendMsg = sendptr(ROOT);
    Enter EnterSend = enterptr(SendMsg);
    Enter EnterRecv = enterptr(ROOT);
    Exit  ExitRecv = UNIQUE({
        ev IN TRACE
        WITH enterptr(ev) == EnterRecv
    });
    Exit  ExitSend = UNIQUE({
        ev IN TRACE
        WITH enterptr(ev) == EnterSend
    });

  CONSTRAINT:
    EXISTS e IN Rs(ROOT, ROOT.Process) SUCH THAT e.Region==r AND
    EXISTS s IN Mq(ROOT, SendMsg.Process, ROOT.Process)
    SUCH THAT s.Timestamp < SendMsg.Timestamp;

  EXPORT:
    cost = (ExitSend.Timestamp  EnterSend.Timestamp) +
           (ExitRecv.Timestamp  EnterRecv.Timestamp);
}

PROPERTY MessagesInWrongOrder(FuncCall r, Experiment e, Region RankBasis){
LET
  float Cost= SUM(i.cost WHERE i IN
                  PATTERN_MATCHES(WrongOrder(r), e));
IN
  CONDITION: Cost>0;
  CONFIDENCE: 1;
  SEVERITY: Cost/Duration(RankBasis,e);
}

PROPERTY FrequentCommInProcess(Function r, Experiment e, Process p
                               Region RankBasis){
LET
  float cost = Summary(r,e,p).Incl;
IN
  CONDITION:  typeof(r)==MPICommFunc AND cost>0 AND
              cost/Summary(r,e,p).NumPasses < SmallMessagesThreshold;
  CONFIDENCE: 1;
  SEVERITY:   cost/duration(RankBasis,e,p);
}

PROPERTY TEMPLATE OnAllProcesses< PROPERTY x(Region, Experiment,
Process, Region>(Region r, Experiment e, Region RankBasis) {

```

```

CONDITION: FORALL p IN e.Processes
    SUCH THAT
        condition(x(r,e,p,RankBasis))
    AND
        confidence(x(r,e,p,RankBasis))==1;
CONFIDENCE: 1;
SEVERITY: max(severity(x(r,e,p,RankBasis)))*e.NrProcessors;
}

PROPERTY OnAllProcesses<FrequentCommProcess> FrequentCommOnAllProcesses;

PROPERTY TEMPLATE CacheCostPerInstance <float CostFunc(Exit)>
    (Exit re, Region RankBasis) {
LET
    float costs=CostFunc(re);
IN
    CONDITION: costs>0;
    CONFIDENCE: 1;
    SEVERITY: cost/(re.LoadInstr + re.StoreInstr) ;
}

L1DCacheCostInstFunc(Exit re)= re.L1DCacheMisses;
L1ICacheCostInstFunc(Exit re)= re.L1ICacheMisses;
L2CacheCostInstFunc(Exit re) = re.L2CacheMisses;

PROPERTY CacheCostPerInstance<L1DCacheCostInstFunc> L1DCacheMissesPerInstance;
PROPERTY CacheCostPerInstance<L1ICacheCostInstFunc> L1ICacheMissesPerInstance;
PROPERTY CacheCostPerInstance<L2CacheCostInstFunc> L2CacheMissesPerInstance;

PROPERTY TEMPLATE AllInstancesInProcess
    <PROPERTY x(RegionExit)>(Region r, Experiment e,
        Process p, Region RankBasis) {
LET
    Setof Exit AllRe = {re IN e.Trace WITH re.Process==p
        AND re.Region==r}
IN
    CONDITION: FORALL re IN AllRe
        SUCH THAT
            condition(x(re))
        AND
            severity(x(re))>0.25;
    CONFIDENCE: 1;
    SEVERITY: max(severity(x(re)));
}

PROPERTY AllInstancesInProcess <L1DCacheMissesPerInstance>
    L1DCacheMissesInAllInstancesInProcess;
PROPERTY AllInstancesInProcess <L1ICacheMissesPerInstance>
    L1ICacheMissesInAllInstancesInProcess;
PROPERTY AllInstancesInProcess <L2CacheMissesPerInstance>
    L2CacheMissesInAllInstancesInProcess;

PROPERTY OnAllProcesses <L1DCacheMissesInAllInstancesInProcess>
    L1DCacheMissesInAllInstancesInAllProcesses;
PROPERTY OnAllProcesses <L1ICacheMissesInAllInstancesInProcess>

```


Appendix D

HPF Property Specification

```
//*****
// APART Example Property Specification
//
// Data Parallel Programming Paradigm Performance Data Model
//
//*****

PERFORMANCE DATA

//*****
//HPF static information
//*****

class HPFRegion extends Region {
    setof Dependence deps;           // data dependence information of this region
    setof HPFDirective dirs;        // HPF directives
    setof HPFDataDeclaration decls; // HPF declarations
}

//Examples for HPF regions

class HPFProcedure extends HPFRegion { // Procedure is function or subroutine
}

class HPFLoop extends HPFRegion {
    HPFLoopType ltype;           // type of loop
}

class HPFIfBlock extends HPFRegion {
}

class HPFBasicBlock extends HPFRegion {
}

class HPFProcedureCall extends HPFRegion {
}
```

```

class HPFArrayAssignment extends HPFRegion {

}

//*****
//HPF dynamic information: summary data
//*****

class HPFRegionSummary extends RegionSummary {
    setof Process processes;           // set of processes executing this region
    HPFSummary sums;                   // performance summary across all processes
    setof HPFProcessSummary process_sums; // performance summary per process
}

class HPFSummary {
    // summary information (arithmetic mean) across all
    // processes for a given region

    int nr_executions;
        // average number of times this region has been executed
        // across all processes executing this region
    float duration;           // execution time
    float comm_time;         // communication time
    float dep_comm_time;     // communication time caused by data dependences
    float align_comm_time;   // communication time caused by data alignment
    float sync_time;        // barrier, reduce, allreduce, ...
    float idle_time;        // idle time
    float io_time;          // input/output time
    float compiler_ovh_time; // compiler overhead time
    float inspector_time;   // time for inspector phase
    float redistrib_time;   // time for redistribution of data structures
    int nr_cache_misses;    // number of cache misses
}

class HPFProcessSummary {
    // performance summary (arithmetic mean) for
    // individual process across all region instances

    Process process;           // process identification

    int nr_executions;
        // number of times this region has been executed by process
    float duration;           // execution time
    float comm_time;         // communication time
    float dep_comm_time;     // communication time caused by data dependences
    float align_comm_time;   // communication time caused by data alignment
    float sync_time;        // barrier, reduce, allreduce, ...
    float idle_time;        // idle time
    float io_time;          // input/output time
    float compiler_ovh_time; // compiler overhead time
    float inspector_time;   // time for inspector phase
    float redistrib_time;   // time for redistribution of data structures
    int nr_cache_misses;    // number of cache misses
}

//*****

```



```

//HPF untility classes
//*****

class HPFTypes {
    enum hpf_directive { PROCESSORS, DISTRIBUTE, ALIGN, RESHAPE, INDEPENDENT,DYNAMIC,...};
    enum hpf_loop_type { DO, INDEPENDENT, FORALL,...};
    enum hpf_var_arr {VARIABLE, ARRAY};
    enum hpf_distr_type {BLOCK, CYCLIC, "*", ":"}
    enum hpf_distr_format {PRESCRIPTIVE, DESCRIPTIVE, TRANSCRIPTIVE, INHERIT}
    enum hpf_alloc {DYNAMIC, STATIC};
}

class HPFDataDeclaration {
    String name;                // name of data
    String data_type;          // type of data (int, float, ...)
    int rank;                  // rank of data
    HPFTypes.hpf_var_arr type; // type of data (variable or array)
    HPFTypes.hpf_alloc alloc;  // allocation type
    HPFTypes.hpf_distr_format format; // prescriptive, descriptive, transcriptive,
                                // inherit distribution format
    setof ArrayDimension dims; // more information for each dimension of arrays
}

class ArrayDimension {
    HPFDataDeclaration decl;    // declaration of associated array
    int size;                   // size of dimension
    setof Process processes;    // set of processes onto which dimension is mapped
    HPFTypes.hpf_distr_type type; // data distribution type
    int block_size;            // for CYCLIC(block_size) or BLOCK(block_size)
    ArrayDimension align;      // aligned with some other array dimension
}

class HPFDirective {
    HPFTypes.hpf_directive hpf_dir; // HPF directives
}

class HPFLoopType {
    HPFTypes.hpf_loop_type ltype; // HPF loop type
}

//*****
// APART Example Property Specification
//
// Data Parallel Programming Paradigm Performance Properties
//
//*****

PERFORMANCE PROPERTIES

LET

HPFRegionSummary summary(HPFRegion r, Experiment e)=
    UNIQUE({sumr IN e.profile | sumr.region==r});

```

```
float duration(Region r, Experiment e)=summary(r,e).sums.duration;
```

```
IN
```

```
PROPERTY TEMPLATE costs <float cost_func(HPFRegionSummary)>
```

```
(HPFRegion r, Experiment e, Region RankBasis)
```

```
LET
```

```
cost = cost_func(summary(r,e))
```

```
IN
```

```
CONDITION: cost > 0;
```

```
CONFIDENCE: 1;
```

```
SEVERITY: cost/duration(rank_basis,e);
```

```
}
```

```
float comm_costs(RegionSummary rs) = rs.sums.comm_time;
```

```
PROPERTY costs <comm_costs> communication_costs;
```

```
Property forall_synchronization_costs (HPFRegion r, Experiment e, Region rank_basis){
```

```
LET
```

```
float forall_sync_time = summary(r,e).sums.sync_time;
```

```
IN
```

```
CONDITION: typeof(summary(r,e).region)==HPFLoop AND  
summary(r,e).region.ltype == FORALL AND  
forall_sync_time > 0;
```

```
CONFIDENCE: 1;
```

```
SEVERITY: forall_sync_time/duration(rank_basis,e);
```

```
float io_time(RegionSummary rs) = rs.sums.io_time;
```

```
PROPERTY costs <io_time> io_costs;
```

```
Property parallel_organization_costs( Region r, Experiment e, Region  
rank_basis) {
```

```
CONDITION: COUNT(procs WHERE procs IN summary(r,e).processes) > 1 ;
```

```
CONFIDENCE: 1;
```

```
SEVERITY: summary(r,e).compiler_ovh_time / duration(rank_basis,e);
```

```
}
```

```
Property redist_costs (HPFRegion r,Experiment e,Region rank_basis) {
```

```
LET
```

```
redist1 = summary(r,e).sums.impl_redistr_time;
```

```
redist2 = summary(r,e).sums.expl_redistr_time;
```

```
IN
```

```
CONDITION:
```

```
(Cond1)
```

```
((typeof(summary(r,e).region)==HPFProcedure)
```

```
AND
```

```
(EXISTS dec IN summary(r,e).region.decls
```

```
SUCH THAT
```

```
dec.format==PRESCRIPTIVE OR dec.format==TRANSCRIPTIVE)
```

```

        AND
        (redist1 > 0))
    OR
    (Cond2)
        ((typeof(summary(r,e).region)==HPFRedistribute)
        AND
        (EXISTS dec IN summary(r,e).region.decls
        SUCH THAT dec.alloc == DYNAMIC)
        AND
        (redist2 > 0))
CONFIDENCE:  MAX((Cond1)->0.8, (Cond2)->1.0);
SEVERITY:    MAX((Cond1)->redist1 / duration(rank_basis,e),
                (Cond2)->redist2 / duration(rank_basis,e));
}

```

```

Property serialization_costs (HPFRegion r, Experiment seq, Experiment par, Region rank_basis){
LET
    float par_comp_costs = summary(r,par).sums.duration -
        summary(r,par).sums.comm_time -
        summary(r,par).sums.sync_time -
        summary(r,par).sums.idle_time -
        summary(r,par).sums.compiler_ovh_time -
        summary(r,par).sums.io_time;

IN
    CONDITION:  par_comp_costs > (duration(r,seq) * loop_serial_threshold)
    CONFIDENCE:  1;
    SEVERITY:   par_comp_costs / (duration(rank_basis,par));
}

```

```

Property uneven_work_distribution (HPFRegion r, Experiment seq, Experiment par,
                                   Region rank_basis) {
LET
    int nr_processes = COUNT(procs WHERE procs IN summary(r,par).processes);
    float opt_duration = duration(r,seq)/nr_processes;
    float deviation = SQRT(SUM (EXP(proc_sum.duration -
        proc_sum.comm_time -
        proc_sum.sync_time -
        proc_sum.idle_time -
        proc_sum.compiler_ovh_time -
        proc_sum.io_time - opt_duration, 2)
        WHERE proc_sum IN summary(r,par).proc_sums ))

IN
    CONDITION: (deviation / opt_duration) > uneven_threshold
    CONFIDENCE:  1;
    SEVERITY:   summary(r,par).sums.duration / duration(rank_basis,par)
}

```

```

float inspector_time(RegionSummary rs) = rs.sums.inspector_time;
PROPERTY costs <inspector_time> inspector_costs;

```

```

END

```


Appendix E

OpenMP Property Specification

```
//*****  
// APART Example Property Specification  
//  
// Shared Memory Paradigm Performance Data Model  
//  
//*****  
  
PERFORMANCE DATA  
  
//*****  
//SMP static information  
//*****  
  
class SmRegion extends Region {           //Region subclass for SM regions  
    Dependence deps[];                   //Dependence information  
  
}  
  
//Sequential regions  
  
class SequentialRegion extends SmRegion {  
                                           //Common information for  
                                           // sequential regions go here  
  
}  
  
//Examples for sequential regions  
  
class FunctionCall extends SequentialRegion {  
  
}  
  
class Function extends SequentialRegion {  
  
}  
  
class IfBlock extends SequentialRegion {
```

```

}
```

```

class BasicBlock extends SequentialRegion {
```

```

}
```

```

//Parallel regions
```

```

class ParallelRegion extends SmRegion {
```

```

    //Common information for
```

```

    // sequential regions go here
```

```

    boolean no_wait_exit;
```

```

    //Region is not terminated by barrier
```

```

}
```

```

//Examples for parallel regions
```

```

class PDo extends ParallelRegion {
```

```

    SMTypes.scheduling_type scheduling_strategy;
```

```

    //Static, Dynamic, Guided ...
```

```

}
```

```

class PSection extends ParallelRegion {
```

```

}
```

```

class PRegion extends ParallelRegion {
```

```

}
```

```

class SMTypes {
```

```

    enum scheduling_type {Static, Dynamic, Guided};
```

```

}
```

```

class DSMMachine extends Machine {    //Properties of target machine
```

```

    int remote_access_time;           //Remote access time
```

```

}
```

```

//*****
```

```

//SMP dynamic information: summary data
```

```

//*****
```

```

class SmRegionSummary extends RegionSummary {
```

```

    //Summary information for region
```

```

    SmSums sums;                       //Sums for whole execution
```

```

    setof SmInstanceSums instance_sums; //Sums per region instance
```

```

    int nr_executions                   //Number of instances
```

```

}
```

```

class SmSums {
```

```

    float duration;                     //Execution time of master
```

```

    float non_parallelized_code;        //Sequential time (duration -
```

```

    // duration for parallel regions)
```

```

    float seq_fraction;                 //Seq_time / duration
```

```

    int nr_remote_accesses;             //Number of remote memory accesses
```

```

    float scheduling;                   //Compiler and or user scheduling time
```

```

    float additional_calc;              //Time for additional calculations
```

```

float cross_thread_dep_ctrl; //Synchronization except entry and exit
                             // barrier and except waiting time
                             // in locks etc.

float cross_thread_dep_wait; //Synchronization waiting time except
                              // waiting in entry or exit barrier

float region_wait; //Waiting in entry or exit barrier
float region_ctrl; //Time for instructions at master
                  // e.g. barriers and organization

int nr_cache_misses; //Number of cache misses
setof SmThreadSums thread_sums; //Thread specific summary data
setof VariableRemoteAccesses
    accessed_variables; //These objects determine the number of
                       // remote accesses for variables
                       // accessed in that region
}

class SmThreadSums {
    int thread_no; //Thread id
    float region_wait;
    int nr_remote_accesses;
    float additional_calc;
    float cross_thread_dep_ctrl;
    float cross_thread_dep_wait;
    int nr_cache_misses;
    setof VariableRemoteAccesses
        accessed_variables; //These objects determine the number of
                           // remote accesses for variables
                           // accessed in that region
}

class SmInstanceSums {
    int nr_threads; //Number of threads executing the region
    float duration; //Execution time of master
    float non_parallelized_code;
    float seq_fraction;
    int nr_remote_accesses;
    float scheduling;
    float additional_calc;
    float cross_thread_dep_ctrl;
    float cross_thread_dep_wait;
    float region_wait;
    float region_ctrl;
    int nr_cache_misses;
    setof SmThreadInstanceSums thread_sums;
    //Thread specific instance information

    setof VariableRemoteAccesses
        accessed_variables; //These objects determine the number of
                           // remote accesses for variables
                           // accessed in that region
}

class SmThreadInstanceSums {
    int thread_no; //Thread id

```

```

float region_wait;
int nr_remote_accesses;
float additional_calc;
float cross_thread_dep_ctrl;
float cross_thread_dep_wait;
int nr_cache_misses;
setof VariableRemoteAccesses
    accessed_variables;           //These objects determine the number of
                                // remote accesses for variables
                                // accessed in that region
}

```

```

class VariableRemoteAccesses {
    String var_name;             //Name of a variable accessed in region
    int nr_remote_accesses;     //Number of remote accesses via
                                // references to this variable
    int size;                   //Size in bytes
    setof PageRemoteAccesses;   //For each page of this variable the
                                // number of remote accesses
}

```

```

class PageRemoteAccesses{
    int page_no;                //Page number related to
                                // virtual address space
    int nr_remote_accesses;     //Number of remote accesses
}

```

```

//*****
// APART Example Property Specification
//
// Shared Memory Paradigm Performance Properties
//
//*****

```

PERFORMANCE PROPERTIES

LET

```

SmRegionSummary summary(Region r, Experiment e)=
    UNIQUE({s IN e.profile WITH s.region==r});

```

```

float sync(Region r, Experiment e)=summary(r,e).sums.region_wait +
    summary(r,e).sums.region_ctrl +
    summary(r,e).sums.cross_thread_dep_wait +
    summary(r,e).sums.cross_thread_dep_ctrl ;

```

```

float duration(Region r, Experiment e)=summary(r,e).sums.duration;

```

```

float remote_access_time(Region r, Experiment e)=
    summary(r,e).sums.nr_remote_accesses
    * e.system.remote_access_time);

```

IN


```

PROPERTY TEMPLATE costs <float cost_func(Region r, Experiment e)>
    (Region r, Experiment e, Region RankBasis){
LET
    cost = cost_func(r,e)
IN
    CONDITION:    cost > 0;
    CONFIDENCE:   1;
    SEVERITY:     cost/duration(rank_basis,e);
}

PROPERTY TEMPLATE rel_costs
    <float cost_func(Region r, Experiment seq,Experiment par)>
    (Region r, Experiment seq, Experiment par, Region RankBasis){
LET
    cost = cost_func(r,seq,par)
IN
    CONDITION:    cost > 0;
    CONFIDENCE:   1;
    SEVERITY:     cost/duration(rank_basis,par);
}

float par_costs(Region r, Experiment seq, Experiment par) = duration(r,par) -
    (duration(r,seq)/par.nr_processors);

PROPERTY rel_costs <par_costs> costs;

float par_costs(Region r, Experiment e) =
    summary(r,e).sums.non_parallelized_code +
    sync(r,e) +
    remote_access_time(r,e) +
    summary(r,e).sums.scheduling +
    summary(r,e).sums.additional_calc;

PROPERTY costs <par_costs> measurable_costs;

Property unmeasurable_costs(Region r, Experiment seq, Experiment e, Region rank_basis){
LET
    float total_costs = duration(r,e) - (duration(r,seq)/e.nr_processors);
    float costs = summary(r,e).sums.non_parallelized_code +
        sync(r,e) +
        remote_access_time(r,e) +
        summary(r,e).sums.scheduling +
        summary(r,e).sums.additional_calc;
IN
    CONDITION: total_costs-costs>0;

    CONFIDENCE: 1;

    SEVERITY: total_costs(r,e)/duration(rank_basis,e);
}

```

```
float non_parallel_code(Region r, Experiment e) =
    summary(r,e).sums.non_parallelized_code;
```

```
PROPERTY costs <non_parallel_code> non_parallelized_code;
```

```
PROPERTY costs <sync> synchronization;
```

```
Property irregular_sync_across_instances
    (Region r, Experiment e, Region rank_basis){
LET
    float inst_sync(SmInstanceSums sum)=sum.region_wait +
        sum.region_ctrl +
        sum.cross_thread_dep_wait +
        sum.cross_thread_dep_ctrl ;
IN
    CONDITION: stdev(inst_sync(inst_sum)
        WHERE inst_sum IN summary(r,e).instance_sums)
        > irreg_behaviour_threshold * sync(r,e)/r.nr_executions;
    CONFIDENCE: 1;
    SEVERITY: sync(r,e)/duration(rank_basis,e);
}
```

```
float load_imb_cost(Region r, Experiment e) =
    summary(r,e).sums.region_wait;
```

```
PROPERTY costs <load_imb_cost> load_imbalance;
```

```
PROPERTY costs <remote_access_time> remote_accesses;
```

```
Property remote_access_to_variable
    (Region r, Experiment e, String var, Region rank_basis)
{
LET
    VariableRemoteAccesses var_sum =
        UNIQUE({info IN summary(r,e).sums.accessed_variables
            WITH info.var_name==var});
IN
    CONDITION: var_sum.nr_remote_accesses > 0;
    CONFIDENCE: 1;
    SEVERITY: var_sum.nr_remote_accesses * e.system.remote_access_time
        /duration(rank_basis,e);
}
```

```
Property multiple_transfer_of_same_data
    (Region r, Experiment e, String var, Region rank_basis)
{
LET
    VariableRemoteAccesses var_info =
        UNIQUE({info IN summary(r,e).sums.accessed_variables WITH info.var_name==var});
IN
```

```

CONDITION: (Cond1)
    (var_info.nr_remote_accesses * e.nr_processors > var_info.size
    / e.system.cache_line_size*summary(r,e).nr_executions)
OR
(Cond2)
    (var_info.nr_remote_accesses * e.nr_processors
    > 0.5 * var_info.size / e.system.cache_line_size * summary(r,e).nr_executions);

CONFIDENCE: MAX( (Cond1)->1, (Cond2)->0.5 );

SEVERITY: var_info.nr_remote_accesses * e.system.remote_access_time
    /duration(rank_basis,e);
}

```

```

Property wrong_page_distribution_for_variable
    (Region r, Experiment e, String var, Region rank_basis)
{
LET
    VariableRemoteAccesses var_sum(String var,setof VariableRemoteAccesses info_set) =
        UNIQUE({info IN info_set WITH info.var_name==var});
IN
    CONDITION:
        EXISTS
            thr_sum    IN summary(r,e).thread_sums,
            thr_psum   IN var_sum(var,thr_sum.accessed_variables).page_sums,
            glo_psum   IN {s IN var_sum(var,summary(r,e).accessed_variables).page_sums
                WITH s.page_no==thr_psum.page_no}
        SUCH THAT
            thr_psum.nr_remote_accesses!=0
        AND
            thr_psum.nr_remote_accesses==glo_psum.nr_remote_accesses;

    CONFIDENCE: 0.5;
    SEVERITY: var_sum(var,summary(r,e).accesses_variables).nr_remote_accesses
        * e.system.remote_access_time/duration(rank_basis);
}

```

```

Property parallel_organization( Region r, Experiment e, Region rank_basis) {

    CONDITION: typeof(summary(r,e).region)==ParallelRegion
        AND summary(r, e).nr_executions > 1 ;
    CONFIDENCE: 1;
    SEVERITY: summary(r,e).nr_executions *
        e.system.parallel_region_cost / duration(rank_basis,e);
}

END

```