Formalizing OpenMP Performance Properties with ASL *

Thomas Fahringer[1], Michael Gerndt[2], Graham Riley[3], Jesper Larsson Träff[4]

[1] Institute for Software Technology and Parallel Systems
University of Vienna, tf@par.univie.ac.at
[2] Institute for Computer Science, LRR, Technical University of Munich
m.gerndt@in.tum.de
[3] Department of Computer Science, University of Manchester
griley@cs.man.ac.uk
[4] C&C Research Laboratories, NEC Europe Ltd.
traff@ccrl-nece.technopark.gmd.de

Abstract. Performance analysis is an important step in tuning performance critical applications. It is a cyclic process of measuring and analyzing performance data which is driven by the programmer's hypotheses on potential performance problems. Currently this process is controlled manually by the programmer. We believe that the implicit knowledge applied in this cyclic process should be formalized in order to provide automatic performance analysis for a wider class of programming paradigms and target architectures. This article describes the performance property specification language (ASL) developed in the APART Esprit IV working group which allows specifying performance-related data by an object-oriented model and performance properties by functions and constraints defined over performance-related data. Performance problems and bottlenecks can then be identified based on user- or tool-defined thresholds. In order to demonstrate the usefulness of ASL we apply it to OpenMP by successfully formalizing several OpenMP performance properties.

Keywords: performance analysis, knowledge representation, OpenMP, performance problems, language design

1 Introduction

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 cycles of experimentation involving gathering performance data, performance data analysis (a-priori and postmortem), detection of performance problems, and code refinements in slow progression. Clearly, the programmer must be intimately familiar

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

with many aspects related to this experimentation process. Although there exists a large number of tools assisting the programmer in performance experimentation, it is still the programmer's responsibility to take most strategic decisions.

In this article we describe a novel approach to formalize performance bottlenecks and the data required in detecting those bottlenecks with the aim to support automatic performance analysis for a wider class of programming paradigms and architectures. This research is done as part of APART Esprit IV *Working Group on Automatic Performance Analysis: Resources and Tools* [Apart 99]. In the remainder of this article 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. Second, dynamic performance-related data describes the dynamic behavior of a program during execution on a target machine.

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 figure* is provided that specifies the importance of the property.

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 can have one or several performance bottlenecks which are characterized by having the highest severity figure. If these bottlenecks are not a performance problem, then the program's performance is acceptable and does not need any further tuning.

This paper 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.

The organization of this article is as follows. Section 2 presents in related work. Section 3 presents ASL constructs for specifying performance-related data and, as examples, classes specifying performance-related data for OpenMP programs. The syntax for the specification of performance properties is described in Section 4. Examples for OpenMP property specifications are presented in Section 5. Conclusions and Future work are discussed in Section 6.

2 Related work

The use of specification languages in the context of automatic performance analysis tools is a new approach. Paradyn [MCCHI 95] performs an automatic online analysis and is based on dynamic monitoring. While the underlying metrics can be defined via the *Metric Description Language* (MDL), the set of searched bottlenecks is fixed. It includes *CPUbound*, *ExcessiveSyncWaitingTime*, *ExcessiveIOBlockingTime*, and *TooManySmallIOOps*.

A rule-based specification of performance bottlenecks and of the analysis process was developed for the performance analysis tool OPAL [GKO 95] in the SVM-Fortran project. The rule base consists of a 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 [GeKr 97].

Another approach is to define a performance bottleneck as an event pattern in program traces. EDL [Bates 83] allows the definition of compound events based on extended regular expressions. EARL [WoMo 99] 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.

The language presented here served as a starting point for the definition of the ASL but is too limited. MDL does not allow to access static program information and to integrate information for multiple performance tools. It is specially designed for Paradyn. The design of OPAL focuses more on the formalization of the analysis process and EDL and EARL are limited to pattern matching in performance traces.

Some other performance analysis and optimization tools apply automatic techniques without being based on a special bottleneck specification, such as KAPPA-PI [EsMaLu 98], FINESSE [MuRiGu 00], and the online tuning system Autopilot [RVSR 98].

3 Performance-related Data Specification

This section presents the performance-related data specification for OpenMP [DaMe 99]. Performance-related data are specified in ASL by a set of of classes following an object-oriented style with single-inheritance. In this article we present the classes as UML diagramms. More details on the specification including a number of base classe for all models can be found in [ApartWP2 99]

Several classes were defined that model static information for OpenMP programs. Class *SmRegion* is a subclass of the standard class *Region* 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. Parallel regions include a boolean variable *no_wait_exit* which denotes whether or not the region is terminated by an implicit exit barrier operation. A specific execution of a region corresponds to a region instance.

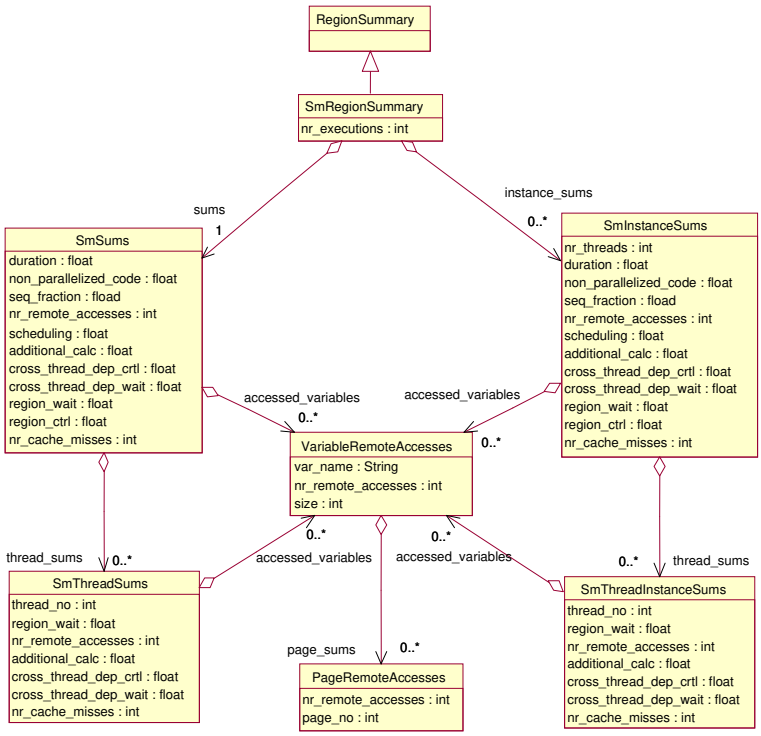

Fig. 1. OpenMP classes for dynamic information

Figure 1 shows the OpenMP class library for dynamic information. Class *SmRegionSummary* extends the standard class *RegionSummary* 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*: $\frac{non_parallelized_code}{duration}$
- *nr_remote_accesses*: number of accesses to remote memory by load and store operations in ccNUMA machines
- *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)

- *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. 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.

4 Performance Property Specification

A performance property (e.g. load imbalance, remote accesses, cache misses, redundant computations, etc.) characterizes a specific performance behavior of a program. The ASL property specification syntax defines the name of the property, its context via a list of parameters, and the condition, confidence, and severity expressions. The property specification is based on a set of parameters.

These parameters specify the property's context and parameterize the expressions. The context specifies the environment in which the property is evaluated, e.g. the program region and the test run. Details can be found in [ApartWP2 99].

The condition specification consists of a list of conditions. A condition is a predicate that can be prefixed by a condition identifier. The identifiers have to be unique with respect to the property since the confidence and severity specifications may refer to the conditions by using the condition identifiers.

The confidence specification is an expression that computes the maximum of a list of confidence values. Each confidence value is computed as an arithmetic expression in an interval between zero and one. The expression may be guarded by a condition identifier introduced in the condition specification. The condition identifier represents the value of the condition.

The severity specification has the same structure as the confidence specification. It computes the maximum of the individual severity expressions of the conditions. The severity specification will typically be based on a parameter specifying the ranking basis. If, for example, a representative test run of the application has been monitored, the time spent in remote accesses may 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 remote access overhead may be compared to the execution time of the shortened loop.

5 OpenMP Performance Properties

This section demonstrates the ASL constructs for specifying performance properties in the context of the shared memory, OpenMP paradigm. Some global definitions are presented first. These are then used in the definitions of a number of OpenMP properties.

Global definitions

In most property specifications it is necessary to access the summary data of a given region for a given experiment. Therefore, we define the *summary* function that returns the appropriate *SmRegionSummary* object. 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.

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

The *sync* function determines the overhead for synchronization in a given region. It computes the sum of the relevant attributes (which are deemed by the property specifier to be components of synchronisation cost) in the summary class.

```
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 ;
```

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.

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

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.

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

Property specifications

The *costs* property determines whether the total parallel overhead in the execution is non-zero.

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

LET
   float total_costs = duration(r,par) - (duration(r,seq)/
                       par.nr_processors);
IN
   CONDITION: total_costs>0;
   CONFIDENCE: 1;
   SEVERITY: total_costs / duration(rank_basis,par);
}
```

This property specifies that the speedup of the application is not simply the serial execution time divided by the number of processors, i.e. that the naive ideal linear speedup is not being achieved. 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.

Performance analysis tools initially only help in analyzing costs for properties which are known to affect performance. After accounting for these properties, any remaining costs must be due new, as yet unencountered, effects. A region has the *identified_costs* property if the sum of the expected potential costs is greater than zero. The severity of this property is the fraction of those costs relative to the execution time of *rank_basis*.

```
Property identified_costs(Region r, Experiment e, Region rank_basis){
LET
   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: costs>0;
   CONFIDENCE: 1;
   SEVERITY: costs(r,e)/duration(rank_basis,e);
}
```

The total cost of the parallel program is the sum of the identified and the unidentified overhead. The *unidentified_costs* property determines whether an unidentified 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 be required.

```
Property unidentified_costs(Region r, Experiment seq, Experiment e,
                            Region rank_basis){
LET
   float totcosts = 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: totcosts-costs>0;
   CONFIDENCE: 1;
   SEVERITY: totcosts(r,e)/duration(rank_basis,e);
}
```

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, relative to the duration of the rank basis. If the focus of the analysis is more on application scaling the severity could be redefined to stress the importance of this property.

```
Property non_parallelized_code(Region r, Experiment e,
                               Region rank_basis){
LET
   float non_par_code = summary(r,e).sums.non_parallelized_code>0;
IN
   CONDITION: non_par_code>0;
   CONFIDENCE: 1;
   SEVERITY: non_par_code/duration(rank_basis,e);
}
```

A region has the following *synchronization* property if any synchronization overhead occurs during its execution. One of the obvious reasons for high syn-

chronization cost is load imbalance, which is an example of a more specific property: an application suffering the *load_imbalance* property is, by implication, suffering *synchronisation*.

```
Property synchronization(Region r, Experiment e, Region rank_basis){
   CONDITION: sync(r,e)>0;
   CONFIDENCE: 1;
   SEVERITY: sync(r,e)/duration(rank_basis,e);
}

Property load_imbalance( Region r, Experiment e, Region rank_basis) {
   CONDITION: summary( r, e ).sums.region_wait >0;
   CONFIDENCE: 1;
   SEVERITY: summary( r, e ).sums.region_wait/duration(r,e);
}
```

When work is unevenly distributed to threads in a region, this manifests itself in *region_wait* time (time spent by threads waiting on the region exit barrier). If the *region_wait* time cannot be measured, the property can also be derived as the execution time of the thread with the longest duration minus the average thread duration.

The *synchronization* property defined above is assigned to regions with an aggregate non-zero synchronization cost during the entire execution of the program. 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. The *irregular_sync_across_instances* property identifies this case. The severity is equal to the severity of the synchronization property since the *irregular_sync_across_instances* property is only a more detailed explanation.

```
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);
}
```

An important property for code executing on ccNUMA machines, *remote accesses*, arises from access to data in memory located on nodes other than that of the requesting processor. Remote memory access involves communication among parallel threads. Since, usually, only the number of accesses can be measured, the severity is estimated based on the mean access time.

```
Property remote_accesses(Region r, Experiment e, Region rank_basis){
   CONDITION: summary(r,e).nr_remote_accesses>0;
   CONFIDENCE: 1;
   SEVERITY: remote_access_time(r,e) / duration(rank_basis,e);
}
```

A previous property (*remote_accesses*) identifies regions with remote accesses. The next property, *remote_access_to_variable*, is more specific than this 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 those provided by the SGI Origin 2000 on a page basis. 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, rather than with respect to the whole program, during a more detailed analysis.

```
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);
}
```

6 Conclusions and Future Work

In this article we describe a novel approach to the formalization of performance problems and the data required to detect them with the future 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.

We applied the ASL to OpenMP by successfully formalizing several OpenMP performance properties. ASL has also been used to formalize a large variety of MPI and HPF performance properties which is described in [ApartWP2 99].

Two extensions to the current language design will be investigated in the future:

1. The language will be extended by templates which facilitate specification of similar performance properties. In the example specification in this paper some of the properties result directly from the summary information, e.g. *synchronization* is directly related to the measured time spent in synchronization. The specifications of these properties are indeed very similar and need not be described individually.
2. Meta-properties may be useful as well. For example, synchronization can be proven based on summary information, i.e. synchronization exists if the sum of the synchronization time in a region over all processes is greater than zero. A more specific property is to check, whether individual instances of the region or classes of instances are responsible for the synchronization due to some dynamic changes in the load distribution. Similar, more specific properties can be deduced for other properties as well. As a consequence, meta-properties can be useful to evaluate other properties based on region instances instead of region summaries.

ASL should be the basis for a common interface for a variety of performance tools that provide performance-related data. Based on this interface we plan to develop a system that provides automatic performance analysis for a variety of programming paradigms and target architectures.

References

[Apart 99] *APART – Esprit Working Group on Automatic Performance Analysis: Resources and Tools*, Dr. Hans Michael Gerndt, Forschungszentrum Jülich, Zentralinstitut für Angewandte Mathematik (ZMG), D-52425 Jülich, http://www.kfa-juelich.de/apart, Feb. 1999.

[ApartWP2 99] T. Fahringer, M. Gerndt, G. Riley, J. Träff: *Knowledge Specification for Automatic Performance Analysis*, APART Technical Report, Workpackage 2, Identification and Formalization of Knowledge, Technical Report FZJ-ZAM-IB-9918, D-52425 Jülich, http://www.kfa-juelich.de/apart, Nov. 1999.

[Bates 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

[DaMe 99] L. Dagum, R.Menon: *OpenMP: An Industry-Standard API for Shared-Memory Programming*, IEEE Computational Science & Engineering, Vol.5, No.1, pp 46-55, 1998.

[EsMaLu 98] A. Espinosa, T. Margalef, E. Luque: *Automatic Performance Evaluation of Parallel Programs*, Sixth Euromicro Workshop on Parallel and Distribued Processing, 1998

[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

[GKO 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

[KaLeObWa 98] R. Hockauf, W. Karl, M. Leberecht, M. Oberhuber, M. Wagner: *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

[MuRiGu 00] N. Mukherjee, G.D. Riley, J.R. Gurd: *FINESSE: A Prototype Feedback-guided Performance Enhancement System*, Accepted for PDP2000, to be held in Rhodes in January 2000.

[RVSR 98] R. Ribler, J. Vetter, H. Simitci, D. Reed: *Autopilot: Adaptive Control of Distributed Applications*, Proceedings of the 7th IEEE Symposium on High-Performance Distributed Computing, 1998

[WoMo 99] 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