H-PCTE – A High-Performance Object Management System for System Development Environments*

Udo Kelter
FernUniversität Hagen, Praktische Informatik V
Postfach 940, D-5800 Hagen, Germany

Abstract
H-PCTE is an object management system (OMS) for distributed, open, and integrated system development environments. H-PCTE performs up to several thousand simple operations per second. H-PCTE is intended to be a basis of environments in which tools operate directly on fine-grained data stored in the object base. Fine-grained data modelling has several important implications for the architecture of environments, for tool design, and for the necessary functionality of the OMS. H-PCTE's performance is due to main-memory-oriented implementation techniques and due to a careful adaptation of the OMS services to the data management needs of tools; as a result, tools need not pay for OMS services which they do not really need. The latter applies in particular to recovery and the external view facilities.

1 Introduction and Overview
Today the development of complex technical systems such as machines, VLSI chips, computer hardware and software, etc. is achieved in large part with the aid of computer-assisted tools (“CAD systems”, “design environments”, etc.). The term “system development environment” (SDE) will be used throughout this paper to refer to integrated sets of such tools.

An SDE has to handle a very large amount of complex data. The handling of data of this kind in many respects presents a real challenge to a database management system designed for this purpose: it must offer a semantically rich data model, support distribution and nested working groups by suitable transaction concepts and access controls, and must deliver a very high performance. These requirements have spawned a large number of new data models, almost all of which are object-oriented in one way or another (for comprehensive surveys and bibliographies, see [UnSSO, Vo91]).

Since there is a wide variety of different data models, they are not easy to classify; a frequently used classification distinguishes between
- **structurally** object-oriented data models, which offer hierarchically structured complex objects; objects, attributes and relationships are accessed using generic operations such as copy, read or write;
- **behaviourally** object-oriented data models, which allow type-specific operations to be defined.

This paper will be entirely concerned with structurally object-oriented data models; in fact, we shall concentrate on models which provide navigating access to objects and which at least partially satisfy the requirements mentioned (unlike non-conventional data models designed for conventional applications). A large number of DBMSs still fall into this restricted class, hitherto termed **object management systems (OMS)**.

The data models of most OMSs are derived from the entity relationship model and have a number of features in common (complex objects, versions, type hierarchies, etc.), but there are many differences with regard to detail, especially in their application programming interface (API).

From the perspective of designers of SDEs, many differences in detail between the various models are of secondary importance. What is of more immediate relevance is the performance of the OMS, which is intimately connected with the effort required to develop particular tools and with the degree of integration of different tools. Unfortunately, the performance of existing systems is still one order of magnitude below the required level.

The H-PCTE project involves a high performance implementation of the data model of PCTE [PCTE90]. The existing prototype, in spite of its relatively "expensive" functionality (complex data model, external schemata, object-specific access controls, forward and backward recovery), attains high or very high levels of performance (several thousand simple data manipula-
tion operations per second).

The aim of this paper is to report on results obtained so far; we will focus on certain aspects which turned out to be particularly relevant to performance and which have not received much attention in the literature so far. H-PCTE is realised as a main-memory DBMS; its high performance is, however, made possible only by a special design of the programming interface, main-memory-oriented internal data structures, a recovery concept geared to the requirements of SDEs, and the extensive use of caches. These implementation techniques are largely independent of the specific data model, i.e. they can also be applied for the implementation of other data models. Another aim of this paper is to discuss implications for the architecture of open development environments.

Section 2 is an introduction to the requirements on SDEs, including integratedness and openness. To fulfil these requirements, the SDE should be based on an open integration framework. Section 3 discusses several levels of granularity on which the data of an SDE can be modelled in the OMS. Section 4 gives a brief outline of the main implementation techniques and features of H-PCTE, as well as the results of benchmarks obtained using the current prototype.

2 System Development Environments
An SDE typically consists of several components which we refer to as tools. These provide support to a variety of activities that occur during the development of a technical system. An SDE has to:
- support all activities that occur during the development of a technical system
- integrate tools in several respects, especially with regard to the data
- be open in the sense of enabling existing tools (possibly written in different languages) to be exchanged or completely new tools to be integrated into the environment
- support distribution, in particular the distributed data storage in a network.

Ideally, it should be possible for an environment to be "plugged together" using either components available on the open market or self-made ones. This approach requires a suitable integration framework [RM91]. An integration framework is a set of interfaces which typically offer services in the areas of process execution and control, data management, input / output, graphics and network functions. An integration framework can also be conceived as a distributed, dedicated operating system.

PCTE. PCTE is the first internationally standardised integration framework for SDEs [PCTES0]. PCTE provides services for data management (basically a structurally object-oriented DBMS), process execution (basically the process model of POSIX), and distribution of data and processes. All the services of PCTE are available through its API. The standard does not comprise a command or query language; textual and/or graphical human user interfaces may be realized as tools which use the API of PCTE.

In what follows, we shall only be concerned with the data management component of PCTE. The data model realised in the PCTE-OMS contains concepts such as complex objects, versions of objects, typing of objects, relationships (which are called links) and attributes, inheritance of object type definitions, and the like. Owing to the assumptions concerning distribution, PCTE has no relational interfaces; instead, objects are localised by navigating along relationships. Tools always operate on the object base via an external schema. The metabase, a special part of the object base, contains all meta (or schema) data, i.e. data about the types in the object base. The extensive access controls are a special feature of PCTE.

3 Modelling the Data of an SDE
This section will investigate the impact the performance of an OMS has on the data integration of an environment. As will be shown, insufficient performance makes it necessary to model the SDE data on a coarse-grained level, thus reducing the tools' data independence and making it significantly more difficult to switch between different tools and to check non-local consistency criteria.

3.1 Coarse-Grained Data Modelling
This kind of data modelling corresponds to the ways in which conventional tools based on file systems manage their data. Typically, the data is divided up in such a way that relatively comprehensive data such as a drawing or a program module are stored within one file. The contents of the files must obey a particular syntax. Typically, tools operate according to a parsing / unparsing schema:
- In a first step, the data to be processed is read from one or more files using a parser which is part of the tool, and is transformed into internal data structures in main memory. The functions of the tool, which, for instance, may correspond to the user commands, then proceed to operate on these internal structures.
- After some time users will write the documents, which have been modified in the meantime, back onto the original file or will write derived data onto com-
completely new files. This is done using an unparsable,
which converts internal data structures into a linearised form that can be stored in files.

This kind of data integration is well known from environments based on file systems; it presents a number of drawbacks, only some of which can be dealt with here for reasons of space.

First, the data independence of the tools is weak. If a new tool is to be integrated into an environment and if this tool uses special additional data within already existing document types, the syntax of the files must be appropriately extended. Existing tools which process such documents would have to be modified, which may not be feasible for a number of reasons.

Another disadvantage, immediately relevant to practical use, is that this architecture does not allow an incremental checking of "non-local" consistency criteria, i.e. consistency criteria between fine-grained data contained in different files. Assume, for example, a software system whose module structure is described in a structure diagram, which is stored in one file, with the source program for each module being stored in another file. If the type of a parameter is modified in the structure diagram using an editor, this modification must also apply to the corresponding source programs: in each call of this procedure a check needs to be carried out to ensure the actual parameter is still compatible with the new type of the formal parameter. For checks of this kind we assume a different tool.

In the case of coarse-grained data modelling, the structure diagram has to be linearised again and written back onto the file; the checking tool then proceeds to read and analyse the contents of this file and, if necessary, those of dozens of other files. This is because, typically, all consistency criteria have to be checked, not just those relating to the modified parameter type.

Since environments constructed in this way cannot operate incrementally, they are often extremely inefficient. In practice, therefore, checking of non-local consistency constraints is often deferred or not carried out at all. It is thus quite possible that errors are spotted only at a later stage, if at all.

3.2 Fine-Grained Data Modelling

In the case of fine-grained data modelling, the complete structure of the data is modelled in the OMS. Rather than constructing internal data structures, the tools operate directly on the OMS.

The problems observed in coarse-grained data modelling no longer arise here. The schema facilities of the OMS make tools data-independent to a fairly large extent. Many tools become smaller as parsers and unparsers are no longer needed. Tools also become more consistent because functions provided by the OMS, such as versioning, transactions, integrity conditions etc., can be exploited. Standardized schemata make it possible to optimize openness and data integration. We assume that non-local consistency constraints are also modelled by data (typically relationships) in an OMS; thus, incremental checks and fast switching between tools are made possible.

Performance Requirements. The obvious architectural advantages of fine-grained data modelling raise the question of why such architectures have hardly been realised until now. The crucial problem here is the performance of the OMS. Performance requirements depend significantly on the kind of tool involved and vary between 500 and over 10,000 simple operations (such as the reading and writing of an attribute) per second. Two major implementation problems arise in the implementation of an OMS which achieves more than 10,000 operations per second:

- One disc access will take around 10 ms. If the amount of time used for disc access is to occupy only a fraction of the entire operation time, then only one disc access may occur during several hundred operations. In practice this means that there are to be no disc accesses at all; thus, the OMS has to be realised as a main-memory DBMS.

- For a message to be exchanged between two processes, currently available workstations with a performance of about 20 MIPS will require between 0.2 and 2.0 ms. If the OMS kernel and application are executed in separate processes, as is common practice with conventional DBMSs, interprocess communication alone will generally cause performance to fall below 1000 operations per second. It follows that, at the present time, if very high levels of performance are to be achieved, the OMS kernel and applications will have to be executed in the same process and, thus, in the same virtual memory.

Impact on the Environment Architecture. The second point mentioned has important implications for the notion of openness of an environment: while the classic interpretation of this notion is that tools are available as executable programs, fine-grained data modelling requires that tools be available as link modules. The complete environment must then be generated by linking these link modules with the OMS kernel and is executed as one process. (The "front end" of a tool, which may contain a graphical user interface, can be run in a separate process.) For a secure implementation, the OMS kernel which is bound to the tools must actually work as a large cache which accesses the
The main aim of the H-PCTE project is a high-object base via a background process. If several tools are to be used in parallel then this must be realized using threads, i.e. light-weight processes.

4 H-PCTE

The main aim of the H-PCTE project is a high-performance implementation of the OMS of PCTE as standardized in [PCTE90], i.e. a performance of roughly 10,000 simple data manipulation operations per second. The design of H-PCTE is based on certain assumptions about its "work load", that is how designers access data and about what sorts of data used by a tool need to be distinguished. These assumptions are exploited in three different ways in order to achieve higher performance:

- in internal optimizations of the H-PCTE kernel, which are invisible to the designers of H-PCTE-based tools. Several caches are examples of this.
- in a careful adaptation of the OMS services to the data management needs of tools, so that tools need not pay for OMS services which they do not really need. This applies, e.g., to recovery.
- in the "pragmatics" of H-PCTE, i.e. rules about how H-PCTE-based tools should be designed and how H-PCTE installations should be managed.

In the following sections, we will discuss specific assumptions about the work load and their consequences on segmentation, recovery and the API of H-PCTE. The discussion requires some knowledge of the implementation techniques applied in the H-PCTE kernel, which are therefore presented first.

4.1 General Implementation Techniques

When designing tools, the performance characteristics of the underlying OMS must be taken into account. This requires a degree of understanding of how the OMS kernel is designed. For reasons of space, we do no more than present the most important implementation techniques in broad outline and illustrate them by means of some examples.

H-PCTE is a main memory DBMS for SDEs. Avoiding disc accesses or clustering is irrelevant to the design of H-PCTE. Instead other optimization aims are given prominence, especially saving main memory space and CPU time. The recovery requirements of SDEs and conventional applications differ substantially. This is why implementation techniques basically different to those in disc-oriented (standard or non-standard) DBMS, or main memory DBMSs for conventional applications, have been applied.

Dynamic Tables. Tables (arrays) are "search structures" for indexed sets with excellent performance; however, they are not appropriate to scattered keys and dynamically growing sets. The problem of scattered keys can be solved in a main-memory DBS by reorganizing internally used keys whenever individual segments of the database are loaded into main memory: as a result, the array is almost always completely filled. The other problem can be solved by using dynamic tables: these are essentially arrays of pointers; each pointer points to an independently allocated main memory area, which contains the actual data value for this position in the table. If a dynamic table overflows, only the pointer array has to be copied, and not the data values. Moreover, space for the data values needs to be allocated only when required. In H-PCTE dynamic tables occur many times. One example is the central object table of a segment; objects are identified by a number within a segment. Several dynamic tables occur in value directories (see below).

Numbering Values. Numbering values means that rather than actually store the value itself, only a number of this value is stored. This makes it possible to save considerable amounts of space when "long" data values occur several times. To convert between value numbers and full values, a value directory is needed, which is typically based on a dynamic table. There are two noteworthy examples where value numbering is used:

- Each link has a link name which identifies this link among the links which lead off from the same object. In the case of fine-grained data modelling, many links (leading off from different objects) will have the same name. Therefore, link name numbers are used internally.
- Each atomic and each complex object in H-PCTE has an individual access control list (ACL), which typically requires between 30-100 bytes. Only an ACL number is stored at the objects internally (rather than a complete ACL). The ACL directory can efficiently convert ACL numbers into full ACLs, and conversely (see [Ke91] for details).

Caches. If complex calculations have to be performed repeatedly, it often makes sense to use a cache. Examples of caches in H-PCTE are:

- Applications access the object base via an external schema. The visibility and other properties of types in the external schema can be queried from a cache which consists of several tables (for object types, link types etc.). Elsewhere, types are referred to by type numbers, which are used as index to these tables.
- Evaluating ACLs is relatively complicated, since H-PCTE supports hierarchical user groups and a three-valued logic for authorizations (see [Ke90] for de-
Therefore, the result of the evaluation of the ACL with number $i$ is cached in position $i$ of a dynamic array, the evaluation cache. Evaluation via the cache takes only a few microseconds and is faster by a factor of 10 to 20 than the normal evaluation.

### 4.2 Segmentation

Designers will typically work with only a few documents at a time; only these documents need to be loaded in main memory. However, these documents may have relationships with other documents not currently operated upon. In order to avoid loading unused data into main memory and in order to allow the object base to be distributed in a network of workstations, an H-PCTE object base is partitioned into segments. Each segment contains a subset of all objects. Segmentation and location of segments are transparent. (In this respect, H-PCTE differs from other systems which are conceptually based on a client-server architecture and which require client processes to explicitly check-out data from the server.) The referential integrity of relationships between objects in different segments is guaranteed. Of course, tools can query on which segment an object resides. H-PCTE allows users to move individual objects between segments (typically in order to have them locally available).

Dynamically, an H-PCTE system (as implemented in the current prototype) consists of one central background process and several tool processes, which, apart from the tool, also contain an H-PCTE kernel (see figure 1). All processes can load segments into their main memory. Tool processes can only access the segments loaded by themselves (with high performance), or segments in the background process (with considerably weaker performance).

![Figure 1: H-PCTE processes](image)

Individual segments are stored in files; the size of these files depends on the amount of user data, i.e. very small segments are also possible and are even recommended. Objects should be distributed among the segments in such a way that the private data of individual designers or user groups are to be found in different segments, so that at least that version of the documents which a designer is working with can be loaded into a tool process. Segments should remain small (typically 1 to 10 MB of user data) with a view to avoiding loading data not required.

### 4.3 Recovery

H-PCTE provides, by default, full backward recovery (i.e. transaction rollback) and forward recovery after a system crash (i.e. reconstruction of the latest consistent state of the object base). Since the required logging tends to degrade performance, it should be avoided if it is not really required. H-PCTE enables tools to selectively switch on or off the recovery features as described below.

**Backward Recovery Options.** Tools which do not need backward recovery can switch it off entirely; then, each OMS operation is immediately committed.

On the other hand, in order to support the realisation of undo functions in editors, arbitrarily many save points can be set within a transaction. A tool can roll back the object base to an arbitrary save point.

**Forward Recovery Options.** Some of the data upon which tools operate are temporary in one way or another. As a result, these data need may not need forward recovery. Examples of such data are position data of graphical representations of documents, e.g. diagrams or unparsed syntax trees, or data generated by simulators. Typically, such temporary data structures are initialised by the tool. A tool will operate on both temporary and non-temporary data. We assume that both sorts of data can be distinguished on the basis of object or relationship types or attributes. Logging of the modifications of such temporary data need not take place and can be switched off in H-PCTE. In order to guarantee normal forward recovery for non-temporary data, certain restrictions are imposed; e.g. a relationship between a temporary and another object must also be temporary.

**Implementation.** To reduce the amount of log data, logging is performed on a relatively “low level”, that is using internal references to objects, types or other data items, rather than their external textual representations. The amount of log data for simple operations could thus be kept as small as 20 - 30 bytes (in contrast to this, the overall length of the parameters typically ranges between 30 and 100 bytes).

1Note that backward recovery of such data still makes sense since this facilitates the realization of undo functions. However, a full discussion of the conditions under which one should store position data in the OMS is beyond the scope of this paper.
4.4 Designing the Programming Interface

Where operations in [PCTE90] require types as (input) parameters, an external name of the type must be passed as a text. This text must obey a specified syntax and must be translated into an internal type identifier within each data manipulation operation. In the case of fine-grained data modelling, only a set of types which are fixed by the tool will emerge. That is why the same external names have to be translated again and again. H-PCTE enables external names to be pre-translated and opaque variables whose value identifies a type to be introduced. In H-PCTE such variables can be defined for:
- object types
- link types
- attributes and
- link names (link names contain the external name of the link type; the above considerations also apply to them).

There exist operations for setting type variables and link name variables, and variants of the normal data manipulation operations, in which types or link names are passed as a parameter using a variable rather than text - these are referred to as "high speed operations". The performance gains are described in section 4.5. All variables are externally identified by a positive integer; internally, the data belonging to the type variables are organised as dynamic arrays.

When setting type variables, checks are carried out to ensure that the specified name is syntactically correct and visible, and direct references are generated to corresponding data in the external schema, showing for example whether the process has the right to generate objects of this type. These checks are not repeated in the high speed operations, or can be simplified significantly.

Link Name Numbers. The value numbering technique is applied to link names internally. Each link name has a segment-specific number. When setting a link name variable, this number is determined and stored as part of the corresponding data structure. This, in turn, enables navigation to be speeded up considerably: for each object, there is an AVL tree containing the links which lead off from this object. The search key for this tree is not, as might be assumed, the full link name (i.e. a text), but a link name number (i.e. an integer value), thus allowing a much more efficient comparison of keys.

Quite generally, this example illustrates that the introduction of opaque variables in the API creates a variety of possibilities for optimizing internal search structures and for caching results of checks or other parts of the execution of operations.

4.5 Benchmark Results

Using the benchmark published in [De+91] the real time taken by important operations was measured on a 28 MIPS workstation. Using the standard interfaces (with external names of types), the most frequent operations take 1 - 2 ms on local data, incl. recovery. Switching off forward recovery speeds up write operations on attributes to about 0.6 ms, pre-translation of type names and link names further saves about 80 % of this time, leading to total execution times around 0.13 ms. Non-local access to data in the server process is around 4 ms slower.

Thus, it seems quite possible to realize SDEs which directly operate on fine-grained data stored in an OMS, and which are open as well as highly integrated.

Acknowledgements. The results presented here have been developed in the context of the H-PCTE project. The author would like to acknowledge the contribution of all members of the team. Particular thanks are due to F. Lindert, D. Platz, M. Roschewski, W. Seelbach and B. Sonderkötter.

References


[PCTE90] Portable Common Tool Environment - Abstract Specification (Standard ECMA-149); European Computer Manufacturers Association, Geneva; 1990

