# PERDIS: PERSISTENT DISTRIBUTED STORE FOR VIRTUAL ENTERPRISE CONCURRENT ENGINEERING[1]

Fadi Sandakly* Sytse Kloosterman** Paulo Ferreira*** Patrice Poyet*

*CSTB, BP 209, 06904 Sophia-Antipolis Cedex, France.
**INRIA, B.P. 105, 78153 Le Chesnay Cedex, France.
***INESC, Rua Alves Redol 9 - 6, 1000 Lisboa Cedex, Portugal.

## ABSTRACT

*Software infrastructure still one of the major difficulties in concurrent engineering environment especially in large scale projects where participants are geographically dispersed and belonging to different organizations. In this paper, we propose a new approach to build Virtual Enterprise software infrastructure that offers persistence, concurrent access, coherence and security on a distributed-shared data store.*

*Keywords: Virtual Enterprise, Concurrent Engineering, Data Sharing, Cooperative Work, STEP, Interoperability, Distributed Object Systems.*

## INTRODUCTION

After the Second World War the industrial competition focused on innovation and marketing. Studying the market, targeting the consumers needs and creating new products were the main factors of company's competitiveness. Time to market and production cost were not major elements for competitive advantage. During the seventies and eighties, industrial competition became harder and companies turned their efforts towards the time to market and cost reduction. This led to a wide acceptance and use of *Concurrent Engineering* (CE) practices in the design of products and their related process including manufacturing and support.

*Concurrent Engineering* is a methodology defining a framework for resolving the problem arising from the sequential approach where loosing time in the design phase leads to delayed production and increased time to market. CE changes deeply the internal organization and the way to work in enterprises from management to production. It encourages teamwork, consisting of multi-disciplinary personnel from all the major departments like engineering, manufacturing, purchasing, sales, support, etc. One of the major issues arising from the practice of CE is the intern information access and handling [5].

Today, the global open and fast-changed market push companies to react more and more quickly to adapt and modify their products. To do this, suppliers as well as contractors corresponding to different companies, have to be tightly involved in the design and production cycles. The CE techniques have been generalized giving a new form of *Collaborative Work* employed at companies' level instead of persons' level. The term *Virtual Enterprise* (VE) refers to this kind of companies consortium. We can define a VE as *a temporary alliance of independent organizations that come together to quickly exploit a*

[1] This work has been supported by PerDiS Esprit Project N° 22533.

Construction Informatics Digital Library http://itc.scix.net/
paper w78-1998-40.content

*product manufacturing opportunity.* They have to develop a working environment to manage all or part of their different resources toward the attainment of their common goal. Obviously, common information definition and sharing is the key problem of the VE. Actually, partners of a VE usually have different business rules and information infrastructure. Being part of a VE means that a company have to adapt its information system (or part of it) to the VE common information infrastructure in order to share and exchange project data with other partners.

During the last few years an important effort has been undertaken in different research projects to define the software infrastructure of the future VE. The main issues of such infrastructure are its *scalability*, its *evolution* and its *easiness to be adapted* by the different members of a VE. In fact, for scalability, the architecture of a VE infrastructure has to be independent of the number of partners, the projects they work on and the type and the quantity of data they manipulate. Furthermore, due to the fast evolution of Information Technologies (IT), this architecture has to be open and easy to evolve. At the same time it has to be simple to use in order to reduce the cost of adapting the IT infrastructure of each partner to it. The main challenges of developing this kind of infrastructure are:

- Definition of a *common data model* representing the information to be exchanged and shared between partners,
- Definition of a *common sharing software infrastructure* that can insure:
  - Data storage,
  - Data integrity,
  - Data security.
- Adapting the existing IT of each partner that has to be involved in the VE to work with the *common data model* and the *common sharing software infrastructure*.

Among the significant research efforts in this domain we can cite the NIIIP project [16] that aims *to develop open industry software protocols that will make it possible for manufacturers and their suppliers to effectively interoperate as if they were part of the same enterprise.* NIIIP bases its architecture on emerging standards (formal and *de facto*) like *STEP ISO-10303* [6] for data modeling and *Corba* [18][15] as a middleware for interoperation of different applications. NIIIP promotes a wide use of the World Wide Web (WWW) as an encapsulation of VE communication protocol to organize and access data. Influenced by the OMG and Corba formalism, NIIIP views VE activity as a set of *Services* offered by partners with *interfaces* defined with IDL language and laid on Corba standard services [17] like *Naming, Persistence, Security, Transaction*, etc. and used on a LAN or a WAN like the WWW. NIIIP architecture is used as a basis for other domain dependent research projects like SMART [21] for the interoperability of Manufacturing Execution Systems (MES), SHIIP [20] for shipbuilding domain, LITE [14] for electronic commerce, AMMPLE [1] for tactical missile manufacturing, etc.

At the European level, an equivalent effort is currently under realization with the VEGA Esprit project which aims *to establish an information infrastructure to support the technical and business operations of Virtual or Extended Enterprises*. This information infrastructure relies on COrba Access to STep models (the COAST) architecture [13] which allows applications to access data using an API that extends the Standard Data Access Interface (SDAI) [7] of STEP. Like in NIIIP, the VEGA COAST platform provides *Services* that can be used by partners as *Conversion* service allowing the mapping between different data schemas or *Workflow* service to manage projects. The COAST is also based on standard Corba services mainly naming and transaction; its final version is still under development.

While ISO STEP seems to offer a modeling language and methodology as well as data models that are largely accepted and used in different manufacturing sectors (Aerospace, Building & Construction, Electronics…), the Corba approach presents, from the practical point of view, some difficulties to build VE software infrastructure. Those difficulties can be summarized as follows:

- Currently, none of the commercial object request brokers (ORB) implements together all services needed for a real VE like security, transaction, concurrency control and persistence.
- Corba is based on *remote method invocation*. With this approach, objects used by an application reside in a remote host and can only be used via their functional interface. This is a disadvantage when objects are frequently accessed (which is the case in most design tools like CAD for instance) because an important amount of processing time is wasted in communications. This increases considerably a non-useful network traffic.
- Most of the Corba services like concurrency access, persistence and security are defined at the level of objects. This means that important problems like data concurrent access, security and data distribution which is a major issue in the application performance because of the remote method invocation mechanism, have to be solved in early phases of the conception of an application. Furthermore, implementing policies for concurrent access and distribution at the level of individual objects (which is a *fine grain* resolution) make them difficult to modify and evolve during the application development phase.
- VE infrastructure has to integrate *existing applications* of different partners. Those applications have to access common data stores and thus have to be interfaced using Corba mechanisms, which require a deep modification in their data structure (the inheritance graph has to be changed to access some Corba services) and sometime a new code restructuring to offer their processing as *service*.

In this paper, we present a new approach to solve some of these problems mainly the remote method invocation, the distribution and concurrent access granularity and the porting of existing applications. This approach is based on a platform of **persistent, distributed and shared memory** called PerDiS. In PerDiS, memory is shared between all applications, even located at different sites or running at different times. This shared memory represents the *shared store* of a VE. Coherent caching of data improves performance and availability, ensures that applications have a consistent view of data, and free developers from manually managing object location. Persistence by reachability, based on a distributed garbage collector frees programmers from dealing with explicit data storage. PerDiS offers transactional mechanisms and checkpointing as well as notification when data is *modified*. Applications can define their security policy and data access rights based on a **task-role** model. Distribution granularity can be tuned at application level and/or PerDiS platform level. This allows an easy porting of existing applications where default distribution granularity can be adopted, which shifts this delicate problem from programming/object level to the applications/VE level.
To open the PerDiS platform to the ISO STEP formalism, we are currently porting an implementation of the Standard Data Access Interface (SDAI) [7] on the top of it. This work consists on extending the SDAI definition to deal with data distribution, ownership and security problems.

In the next section we present an overview of the PerDiS platform. Then we discuss the issues of developing a distributed SDAI using PerDiS. Finally we show some preliminary results and we summarize our work.

## PERDIS

In this section we outline the PerDiS architecture and we show how it offers coherent data sharing, persistence and security.

### Architecture

A PerDiS system [19] consists of machines running two kinds of processes: application processes and a single PerDiS daemon (PD). Applications use PerDiS through an API, connecting the application with the PerDiS user level library (PULL). A PULL communicates with its corresponding PD locally, and PDs communicate with each other over the network. The PULL deals with application-level memory mapping, data transformations and management of objects, locks and transactions. When the application, through the API and the PULL, needs locks or reads or writes data, it makes requests to the local PD, which deals with caching, issuing locks and data, storage, security and communication with remote machines. A typical configuration is shown in Figure 1. Note that application processes are optional. In fact, the PerDiS architecture is a symmetrical client-server architecture in which each node behaves both as a client (requesting data) and as a server (providing data). A machine with just a PD running behaves as a pure server.

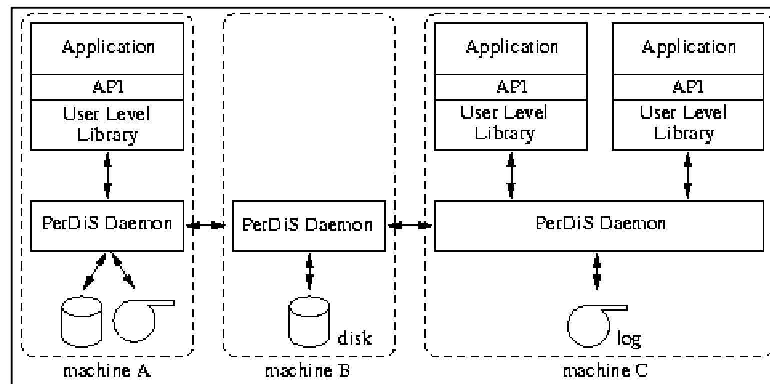


*Figure 1: PerDiS Architecture*

### Objects and clusters

Objects in PerDiS are sequences of bytes representing some data structure. They are not limited to e.g. C++ [22] objects. An application programmer allocates objects in a cluster which is a physical grouping of logical related objects. In contrast with current technology like Corba [18], clusters are the user-visible unit of naming, storage and security, allowing efficient and large-scale data sharing applications. In fact, a cluster combines the properties of a heap (programs allocate data in it) and of a file (it has a name and attributes, and its contents are persistent). Programmers use URLs [3] to refer to clusters, e.g. pds://perdis.esprit.ec.org/clusters/floor1.

An object in some cluster may refer, using normal pointers, to some other object in the same cluster or in another one, even when the current machine does not actually hold the pointed-to object. While navigating through an object structure, an application may implicitly access a previously "unknown" cluster. For instance, Figure 2 shows two clusters located at two different machines. Starting from the entry point *start* in the leftmost cluster, one can navigate through the objects e.g. `start->a->b->print()`, implicitly accessing the remote cluster. The same function could have been called by first opening the remote cluster and then calling `first->print()`.

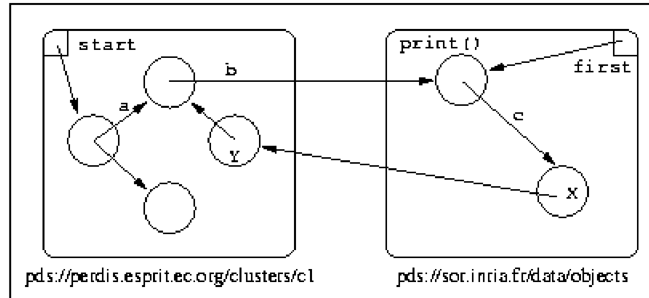


*Figure 2: Clusters, objects and remote references*

## Caching and replication

Data distribution in PerDiS is based on lazy replication and cooperative caching. Lazy replication means that the system only makes copies of data when an application accesses it. Cooperative caching means that caches of different PDs interact to fetch the data an application accesses. Replication and caching avoid the remote access bottleneck because all data access is local. In addition, since replicas are kept in several cooperative caches, data requests are spread over several nodes preventing data access bottlenecks. A potential drawback of this approach is false sharing. False sharing occurs when two applications access different data items that happen to be stored in the same unit of locking. Each application has to wait until the other has unlocked the data before it can access it, even though there is no real sharing. We cope with this by using small locking granularities, which may even vary over time.

## Transactions

Due to replication, several replicas of some object may exist at the same time. A transaction-based consistency control keeps the data globally consistent. PerDiS supports several transaction models that may differ along the following dimensions: explicit vs. implicit locking, optimistic vs. pessimistic locking, serializable vs. non-serializable. The need for these transaction models is driven by the application domain, where interactive applications may issue multiple long transactions during which they may want to access data at the same time.

Transactions using implicit locking automatically lock data accessed by the application. When an application reads data, it will be protected by a read lock. When an application modifies data it will be protected by a write lock. Modifying data that is already protected by a read lock causes an upgrade of the lock from *read* to *write*. Implicit locking is mainly provided to make porting existing applications to PerDiS easy. However, it reduces concurrency and increases the risk of data access dead locks. Therefore, when developing new applications, explicit locking is preferred. Explicit locking transactions do not use automatic locking, but rely on explicit object intent requests, using functions like `try_lock()`, `lock()`, and `unlock()`.

An optimistic transaction defers the locking of data to commit time. If it then turns out that some objects have been modified, the optimistic transaction aborts. On the other hand, pessimistic transactions take locks before they access data. Serializable intent modes ensure that different transactions don't interfere, even if their implementation is interleaved. A long lasting serializable transaction may block other transactions and this may not be acceptable

for some applications. Therefore non-serializable intent modes, e.g. based on versions of data, are also supported.

When starting a transaction, a programmer may indicate that the application must be notified when the involved data is accessed or modified by other transactions. This allows the development of reactive applications. For instance, the system can generate an event when the value of a data item changes. Applications that registered to this event can then update their cached value using a "refresh" function.

### Persistence

Persistence in PerDiS is based on persistence by reachability (PBR) [2]: an object is persistent if and only if it is transitively reachable from a persistent root. A persistent root is a distinguished, named reference which is persistence by default. To illustrate PBR, reconsider Figure 2. All objects are reachable from the roots *start* and *first*, and thus they are persistent. If the pointer `first->c` would be set to `NULL`, objects X and Y would no longer be reachable, and would automatically be deleted. Destroying the root objects *start* and *first* would delete all objects in both clusters.

The PBR model has two main advantages: it makes persistence transparent and frees programmers from memory management. Persistence is transparent since it is deduced from the reachability property. It frees the programmer from memory management because programmers only allocate memory; deallocation is performed by the system if necessary. This prevents dangling pointers and memory leakage.

### Security

Protecting data in VEs is important for two reasons. First, data often represents a large amount of money due to labor intensive tasks. Losing data means losing money. Second, data often represents knowledge and provides companies with a competitive edge. Due to the nature of a VE, partners cooperating in one project may be competing in another. Therefore, PerDiS provides security means [4] to protect data in a collaborative environment. Security in PerDiS consists of two parts: data access control and secure communication. Data access is controlled by groupware-oriented access rights based on users' tasks and roles. This means that one can specify access rights for a user having a specific role in a specific task. Access rights can be assigned on a cluster basis to reduce management overheads. Secure communication uses public key schemes for signed data access requests, shared keys for encryption and a combination of the two for authentication of the originators of messages.

## DISTRIBUTING STEP SDAI

The international Standard for the Exchange of Product Model Data (STEP) provides a basis for communicating product information at all stages in the product life cycle. The key word of data exchange in STEP is *Data Model Sharing*. In fact, STEP defines tools like EXPRESS language [8] to develop data models that can be used in different applications allowing interoperability and common data structures for data sharing. EXPRESS is an OO-like language providing mechanisms to model constraints on data like *Global Rules* and the *Uniqueness* of object values. STEP provides also a definition for data exchange [9] which is an ASCII format that can be used to exchange data defined with EXPRESS. Finally, for data storage and access, STEP specifies an application programming interface (API) called SDAI [7] for STEP Data Access Interface that defines the way applications can store and retrieve instances in databases. So applications sharing the same data model can share databases.

SDAI does not deal with *concurrent access* of data nor with *distribution* of databases. Data security is not defined in the SDAI neither.

In the scope of PerDiS project we decided to implement a distributed version of the SDAI to allow the development of concurrent STEP applications. In the rest of this section we describe briefly the SDAI architecture and the way we port it to the PerDiS platform as well as extensions added to its API. This work is still on progress, we show in the Experiments section some preliminary results.

## Architecture of SDAI

The SDAI is defined in four different schemas written in EXPRESS (see Figure 3):

- The SDAI **Dictionary Schema**: It includes definitions of entities needed to represent a meta-model of an EXPRESS schema. Instances of these entities that correspond to a given schema constitute the SDAI Data Dictionary.
- The SDAI **Session Schema**: It includes the definition of entities needed to store the current state of a SDAI *session* started by an application. Information stored are mainly the list of repositories opened by the application and their access mode, current transactions, events and errors.
- The SDAI **Population Schema**: It defines the organization structure of a SDAI *population*. A SDAI population is the set of instances stored in SDAI repositories. Three main entities to store instances are defined in this schema (see Figure 4):
    1. *SDAI Model*: is a grouping mechanism consisting of a set of related entity instances based upon one schema,
    2. *Schema Instance*: is a logical collection of SDAI Models based upon one schema. A schema instance is used as a domain for EXPRESS *Global Rules* validation or as a domain over which references between entity instances (in different models) are supported or as domain for EXPRESS *Uniqueness* validation.
    3. *Entity Extent*: it groups all instances of an EXPRESS entity data type that exists in a SDAI Model.
- The SDAI **Parameter Data Schema**: This schema describes in an abstract terms the various types of instances that are passed and manipulated through the API. It provides definitions for EXPRESS *simple types*, EXPRESS *entity instance*, EXPRESS entity *attribute value*, EXPRESS *aggregations* and *iterators*, etc.

Those schemas are independent of any implementation language. The SDAI language bindings (SDAI Implementations) are specified for computing languages like C, C++, Java and IDL. Two types of SDAI language bindings are identified: **late** bindings and **early** bindings. Late bindings are applied to any application data model (EXPRESS Schema) using the same set of functions. This set of functions is available in an API independently of the application schema. A SDAI late binding implementation implies an implementation of the dictionary schema. Early bindings describe the generation of a data access interface based upon a specific application data model. In this approach the complete set of functions available in the API depends on the application schema underlying the SDAI implementation. A SDAI early binding implementation does not include a dictionary representation.

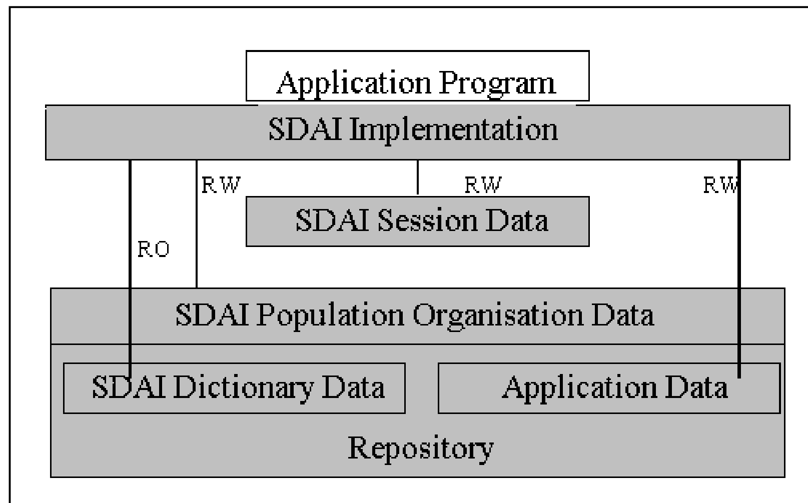


*Figure 3: SDAI Architecture*

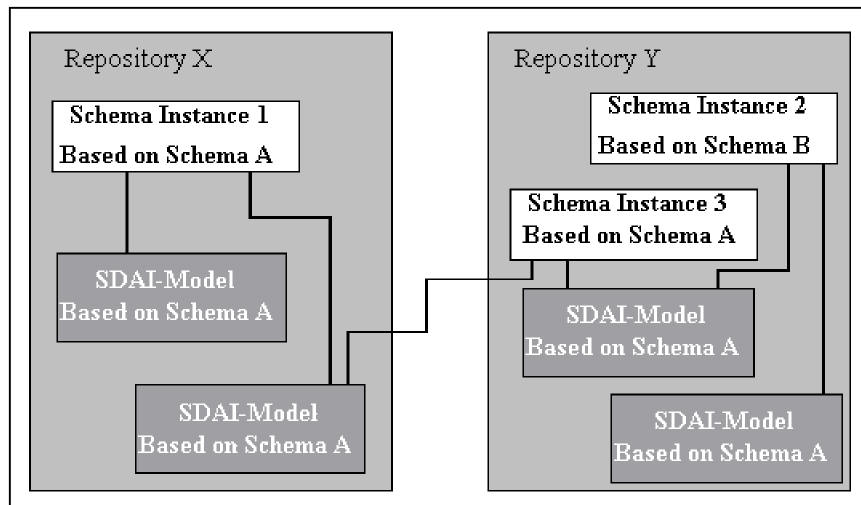


*Figure 4: SDAI Storage Structure*

## Porting SDAI on PerDiS platform

We discuss in this section the main issues of developing a persistent distributed version of a SDAI which allows concurrent data access. Some of our design decisions have been influenced by one of our principal goals in PerDiS which is the fast porting of existing applications that were neither developed to work in a concurrent environment nor in a transactional way. To do this we preserved as far as it was possible the SDAI API described in [10] by introducing *implicit* concurrent and transactional behavior based on some of PerDiS features. However, we extended the SDAI with a specific distribution and concurrence API that can be used by new applications.

### Persistence

In the SDAI, application data can be either persistent or not. A specific interface is defined to create and manipulate persistent instances by their unique identifiers (PID). Unique identifiers are persistent labels attached to persistent application instances. An identifier is unique within the repository containing the SDAI model that contains its application instance. This approach

is adapted to traditional database persistence implementation. As we saw in the previous section, PerDiS is based on *Persistent Shared Memory* where no difference exists between persistent and volatile object manipulation. No need to use unique identifiers to access persistent objects; this is made through normal pointer navigation. Furthermore, we assume from our application domain that lifetime of objects created by an application is longer then its execution duration (objects are used by other applications and users). For those reasons we decided to make all application instances created during a SDAI session persistent. However, applications can explicitly delete objects from persistent stores. At the same time the persistence by reachability mechanism takes care of removing unused objects created by applications.

## SDAI Storage Structure

In a SDAI implementation, application instances are stored in a logical structure. These structures will lay on the PerDiS persistent distributed store (PDS). The main object *collector* in the SDAI is the *Model*. From the application point of view, a model is a set of related instances. Another important collector is the *Repository*. A repository is a collection of models. Those two objects are part of the application data organization and have to be persistent. *Schema Instances* are also logical collectors that contain several models related to the same schema. All these objects are mapped to PerDiS clusters which are a logical collection of application objects.

## Distribution Granularity

In the PerDiS approach, the distribution granularity as seen by the application is a cluster. Physical distribution is hidden and the PerDiS platform can manage this granularity automatically or semi-automatically to optimize performance. From the SDAI point of view, the smallest collector structure is the model. Implementing SDAI models with clusters give applications fine-grain distribution granularity. This granularity is adjustable because models can contain any number and kind of application instances. Inside a cluster, instances can be grouped using entity extent objects.

## Concurrent data access

There is no concurrent access specification in the current definition of the SDAI. The main issue of concurrent access is the data coherence. Locking mechanisms are used by applications to ensure the validity of data. The notion of locking has been added to our SDAI implementation in two different ways:

*(i)* *Implicitly* by using the implicit locking mechanism offered by PerDiS each time a persistent data is accessed in a read or a write mode. Implicit locking allows the porting of applications that don't deal with concurrency without modifications.

*(ii)* *Explicitly* by extending the SDAI API to offer locking primitives on application instances. This allows the development of new applications. Three different locking modes are defined: *read-only*, *read-write* and *no-guarantee*. The last mode allows an application to access to a data in read mode without blocking applications that need to modify this data. However, it is not guaranteed that the obtained data is globally coherent.

We adopt the notion of *notification* when data changes introduced by PerDiS. In fact, an application can declare its interest in a piece of data (that it can access in a no-guarantee mode). PerDiS notifies the application when the data is changed by another one. The application can then use the *refresh* function to update the data.

### Transactions

Once an SDAI session is started, application can manipulate instances in stores in transactional manner: the application *starts* a transaction, defines the *access mode* (*read-write* or *read-only*), and then *access* data, and finally it can *commit* the transaction or *abort* it. We extend this behavior by adding the *no-guarantee* access mode. By default, transactions are mapped to PerDiS *pessimistic transactions*. The standard API is extended to support the *optimistic* transactions.

### Security

The security model in PerDiS is defined at two levels:

*(i)* security of communications which is transparent to the application,

*(ii)* security of stored data expressed with *access right* given to different users in a VE according to a *task-role* model [4]. This model separates the security attributes from the data model definition. It allows the management of *legal* and *data ownership* aspects in a VE. No major modifications have to be done to port application that don't deal with security except the handling of access right violation which can be done at the highest level of an application.

Access right are defined using security tools that will be developed during the project. The mapping of the SDAI storage structure to the PerDiS clusters allows us to use those tools with the SDAI implementation without modifications.

## EXPERIMENTS

The PerDiS project is in its second year. A first version of the PerDiS platform has been released at the beginning of 1998[2]. Several test applications have been ported on this version like the XFig interactive drawing program. To start working with STEP applications, we ported a reduced version of a SDAI early binding based on ISO STEP AP225 format [11]. This format is an explicit shape representation of building models.

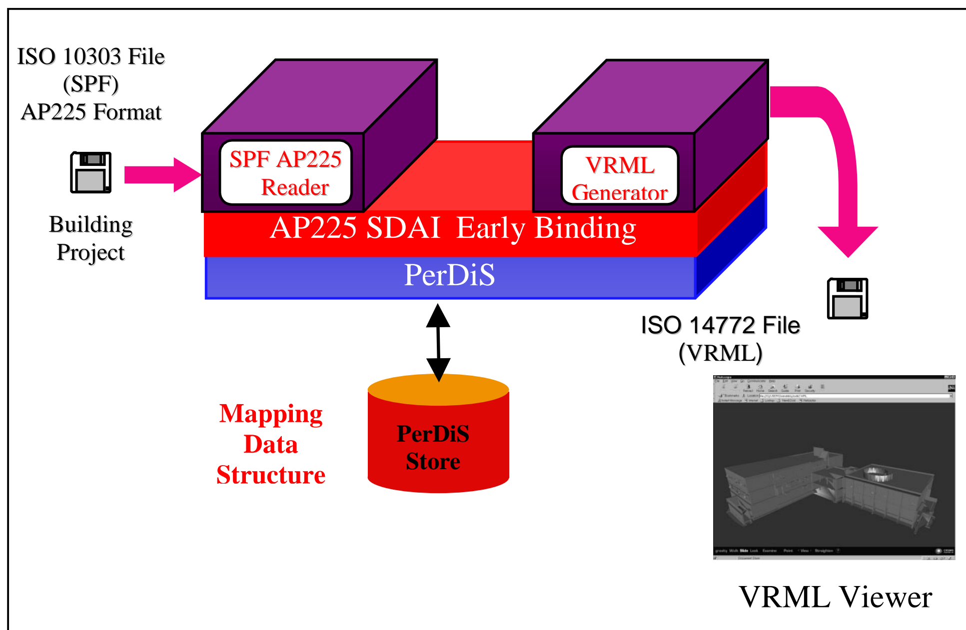


*Figure 5: Mapping AP225 Format to VRML*

[2] This release and some test programs can be found at http://www.perdis.esprit.ec.org/download/.

Two application modules have been ported on this version of the SDAI: the first one is a SPF format reader used to populate SDAI models i.e. PerDiS clusters and the second one is a mapping application that access SDAI model corresponding to the building project (from a local or a distant machine) and generates a VRML [12] file representing a 3D view of the building (Figure 5). Porting these applications has been done in a relatively short time thanks to the small number of modifications that had to be made in the original code.

## CONCLUSION

Today, the major difficulty to build a Virtual Enterprise consortium is the lake of a software infrastructure which can integrate persistence, distribution, concurrency control, coherence and security as well as an efficient interface allowing a fast porting of existing applications to reduce the development effort needed to insure the interoperability of different tools from different partners. In this paper we presented PerDiS, a new approach to develop VE software infrastructure using a persistent distributed store which is based on lazy replication and a global cooperative and coherent caching. This approach reduces the performance problem by avoiding the remote access techniques used by traditional distribution approaches. Furthermore, PerDiS includes features such as implicit locking and optimistic and pessimistic transactions to allow a simple and fast porting of applications that were not developed to work in a transactional and concurrent environment.

Another major difficulty of VE is the definition of common data models. For this purpose, we propose the standardization efforts as a solution. In fact, STEP techniques and methodologies start to be widely accepted in different manufacturing sectors. To open the PerDiS platform to a wide range of application domains we are implementing a distributed version of the Standard Data Access Interface (SDAI) to develop concurrent STEP tools. This work is still under development, we hope be able to implement a stable version of PerDiS before the end of the project and to compare performance in our approach to other traditional ones.


## ACKNOWLEDGMENTS

We would like to think all partners of the PerDiS Esprit project 22533 who contributed to the design and the development of the platform and the test applications. Members of PerDiS consortium are (in alphabetical order): CSTB (France), IEZ (Germany), INESC (Portugal), INRIA (France) and QMW (Great Britain).


## REFERENCES


[1] Affordable Missile Manufacturing Pilots - Linking Enterprises (AMMPLE). URL http://ammple.npo.org.

[2] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, P.W. Cockshott and R. Morrison. *An approach to persistent programming*. In The Computer Journal, 26(4):360-365, 1983.

[3] T. Berners-Lee, L. Masinter, M. McCahill (eds.), *Uniform Resource Locaters*. December 1994, RFC 1738.

[4] G. Coulouris, J. Dollimore and M. Roberts. *Security Services Design*. PerDiS deliverable PDS-R-97-008. URL http://www.perdis.esprit.ec.org/deliverables/docs/T.D.1.1/A/T.D.1.1-A.html.

[5] S. M. Fohn, A. Greef, R. E. Young and P.O'Grady. *Concurrent Engineering*. In Lecture Notes in Computer Science, volume 973, 1995.

[6] J. Fowler. *STEP for data management, Exchange and Sharing*. Great Britain: Technology Appraisals, 1995. ISBN 1-871802-36-9.

[7] ISO 10303-22. *Industrial automation system and integration-Product data representation and exchange- Part 22. Implementation methods: Standard Data Access Interface specification*. 1996.
[8] ISO 10303-11. *Industrial automation system and integration-Product data representation and exchange- Part 11. Description methods: The EXPRESS language reference manual.* 1994.
[9] ISO 10303-21. *Industrial automation system and integration-Product data representation and exchange- Part 21. Implementation methods: Clear Text Encoding of the Exchange Structure.* 1994.
[10] ISO 10303-23. *Industrial automation system and integration-Product data representation and exchange- Part 23. C++ programming language binding to the standard data access interface.* ISO TC184/SC4/WG11 N004. Jan. 1997.
[11] ISO 10303-225. *Industrial automation system and integration-Product data representation and exchange- Part 225. Application Protocol: Building Elements Using Explicit Shape Representation.*
[12] ISO 14772-1. *The Virtual Reality Modeling Language.*
[13] M. Köthe. *COST Architecture: the Corba Access to STEP Information Storage - Architecture and Specification.* Deliverable D301 of ESPRIT 20408 VEGA project. January 1997.
[14] NIIIP Lite Project. URL http://niiplite.iti-oh.com.
[15] T. J. Mowbray, R. Zahavi. *The Essential CORBA - System Integration Using Distributed Objects*. 1996. John Wiley and Sons.
[16] The National Industrial Information Infrastructure Protocols (NIIIP) Consortium. *The NIIIP Reference Architecutre* (Revision 6). *URL* http://www.niiip.org/public-forum/NTR96-01/NTR96-01-HTML-PS/niplongd.html.
[17] Object Management Group. *Corba Services Specification.* URL http://www.omg.org/corba/csindx.htm.
[18] Object Management Group. *The Common Object Request Broker Architecture and Specification* (CORBA) Revision 2.1. September 97. URL http://www.omg.org/corba/corbaiiop.htm.
[19] M. Shapiro *et al.*, *The PerDiS Architecture*, PerDiS deliverable PDS-R-97-002, URL http://www.perdis.esprit.ec.org/deliverables/docs/architecture.
[20] The Shipbuilding Information Infrastructure Project (SHIIP). URL http://shiip.npo.org.
[21] Solution for MES-Adaptable Replicable Technology (SMART project). URL http://smart.npo.org.
[22] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986.