Tool Construction for the British Airways SEE with the O_2 ODBMS*

Wolfgang Emmerich
Interoperable Systems Research Centre, City University, London EC1V 0HB, UK

Jim Arlow
LogOn Technology Transfer, Burgweg 14, 61476 Kronberg, Germany

Joëlle Madec
$O_2 Technology$, 7 rue du Parc Clagny, 78000 Versailles, France

Mark Phoenix
British Airways Plc, TBE (E124), Viscount Way, Hounslow, UK

**Software engineering environments (SEE) support the construction and maintenance of large-scale software systems. They integrate tools for the production and maintenance of documents such as requirements definitions, architecture definitions or user manuals. Very few SEE tools meet all the developers' requirements. Some requirements that are important in practice have not been appropriately addressed. These are inter-document consistency handling, version and configuration management and co-operative work. We claim that the reason for current tools not meeting these requirements is the fact that the required database support for maintaining documents is only now becoming available. The British Airways SEE meets these new requirements. Its tools were constructed using the O_2 object database management system, which has been extended to become a database management system for software engineering. We discuss the experiences we made during tool construction for this SEE.
© 1997 John Wiley & Sons**

1. Introduction

A software development process that develops and maintains a software system consists of a number of different *tasks*. Examples are *requirements analysis* tasks through which the requirements of future customers of a software system are elicited or *architectural design* tasks through which the different components of software systems and relationships among them are identified. The implicit assumption of the Waterfall model [41], that these tasks be performed in mutual exclusion, has proved to be unsound [9]. Tasks are, in fact, performed in an incremental and intertwined manner [49, 23].

The purpose of each task is to produce a set of *documents*, such as use cases during requirements analysis or Booch class diagrams [10] during architectural design. Such documents are written in formal graphical or textual languages. These languages determine *document types* and the purpose of each task of a software process is to create, analyse and maintain *documents* of the types identified for that task. Document types are defined in terms of syntax and static semantics of the underlying specification languages.

Apart from static semantic constraints of the formal languages, there are also consistency constraints between different documents. These *inter-document consistency constraints* are not confined to documents of the same type but frequently exist between documents of different types. Such a constraint may require, for instance, that each class identified in an analysis model should be refined by a class of the same name in the Booch design model. A major problem is that the mix

of document types is process specific. A process developing a safety critical real-time application will use document types induced by languages such as Z [44] and Ada [17] which differ considerably from those used in a traditional information system development, i.e. entity relationship diagrams and fourth generation languages.

The need arises to assist software developers in the production of documents that meet inter-document consistency constraints. Software developers, which we refer to as *users* hereafter, require a *tool* for each document type. Such a tool should then implement the language associated with the document type and offer commands to edit documents. During editing the tool should be supportive in achieving syntactic and static semantic correctness of documents, browsing of semantically related documents and it must check for inter-document consistency. Checking for constraints with documents of other types requires tools to be integrated into a *software engineering environment* (SEE). Due to the fact that inter-document consistency constraints depend on the process-specific mix of document types, a need arises to support SEE construction and customisation.

A *process-centred software engineering environment* is an SEE that also has a component called a *process engine*. The process engine maintains knowledge about the software process, the particular state a development project currently has and sometimes even the evolution of development states over time. In doing so, it guides developers through tasks they are obliged to perform, automates particular tasks and controls the way multiple users cooperate. Examples of such environments are Merlin [37], Melmac [16], and Marvel [7].

The main contributions of the GOODSTEP project to solving the problems of construction and customisation of process-centred SEEs are the following:

- The O_2 database system [3] has been extended with object-level concurrency control, a version manager, object-oriented views [42], triggers [13] and schema updates [25] so as to become a database suitable for the construction and customisation of process-centred software engineering environments.
- The process modelling language SLANG [5] was defined and a process engine interpreting SLANG models was implemented. It controls tool execution and therefore guides the overall development process. An account on the use of O_2 for the SLANG interpreter implementation is given in [4].
- The GOODSTEP Tool Specification Language (GTSL) [20] was defined and *GENESIS*, a compiler for this language has been implemented. GENESIS generates object database schemas that implement the structure of documents and the commands offered by the respective tools.

SPADE, GENESIS and the O_2 database system with its extensions were evaluated during the construction of an SEE for British Airways. British Airways is one of the largest software developers in the U.K. currently with some 2,000 IT staff. In order to simplify maintenance of their applications and increase productivity during their development, for instance through corporate reuse, a number of projects at BA are being developed using object-oriented techniques and C++ [18] as the programming language. The IT division has established a group that is in charge of development and maintenance of libraries containing reusable classes that are considered of importance for the carrier.

The use of SPADE, GENESIS and the O_2 database for the construction of the BA SEE is outlined in FIG. 1. The purpose of the SEE is to support the development and maintenance of reusable class libraries. A SLANG process model defines different roles for BA software engineers and defines the way developers at British Airways cooperate. GTSL specifications define a number
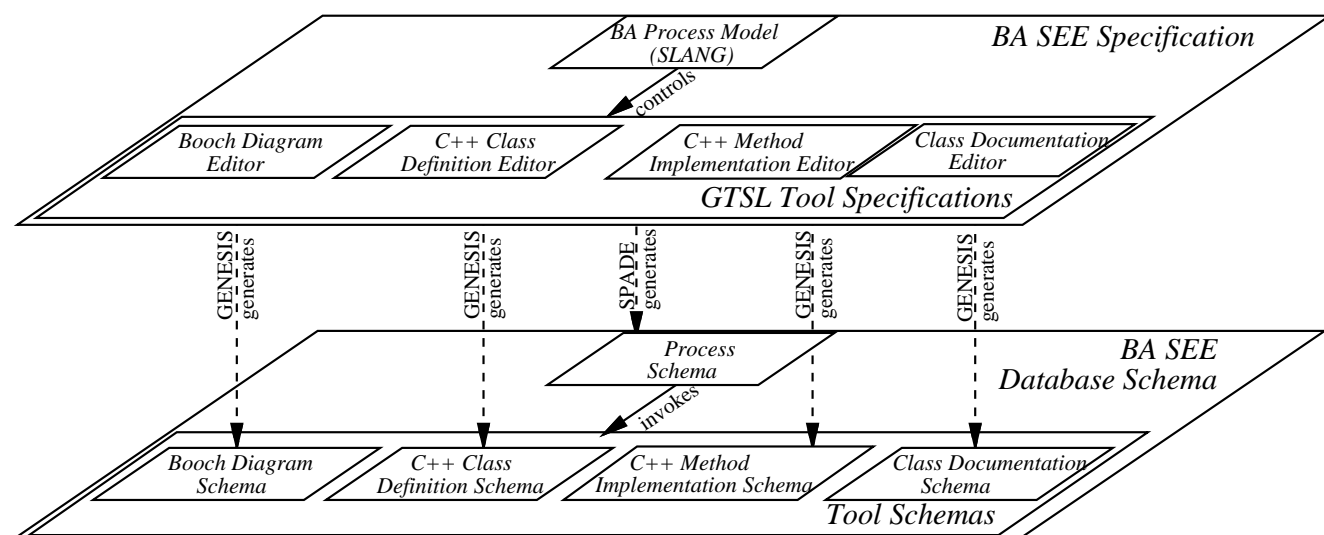

FIG. 1. Construction of the British Airways SEE

of tools that BA software engineers can use to develop class libraries. SPADE transforms the process model into a schema that is used to store the state of the actual process. GENESIS is used to generate schemas for the different tools. These schemas define the structure of documents and implement the tool commands available for document creation and modification.

Our experiences when defining the BA process model and its integration with the generated tools are the subject of a companion paper [21]. The focus of this paper is on how O_2 was exploited for the construction of BA SEE tools.

This paper is structured as follows. In Section 2, we discuss major problems software developers at British Airways are faced with in their daily work. These problems lead to the requirements definition for tools for the BA SEE. Section 3 discusses how documents should be represented so as to meet these requirements. In Section 4 we discuss extensions to the O_2 system that were done within GOODSTEP and focus on how they were exploited for the construction of the BA SEE tools. In Section 5 we present an evaluation of the BA SEE and trace results to their origin in the O_2 system. Section 6 relates our work to the literature and we conclude the paper in Section 7.

2. Requirements for BA SEE Tools

2.1. Inter-Document Consistency

The British Airways reuse group has adopted the Booch methodology [10] for the design of reusable classes. It has developed corporate programming guidelines that define a subset of C++ that is approved for use. This subset excludes a number of C++ constructs, such as ellipses, friends and inlines, whose application might create problems such as statically uncheckable parameter lists or the broken encapsulation of classes. Moreover, each class of a library has to be accompanied with appropriate documentation. Without documentation, client projects were not able to effectively reuse these classes. In summary, documents of four types have to be produced and maintained. These are Booch class diagrams, C++ class interfaces, C++ method implementations and appropriate documentation for C++ classes. Examples of these document types are displayed in FIG. 2 at the user interface of the BA SEE.

Documents of these types must meet a number of inter-document consistency constraints. A class identified in a Booch diagram must be refined in a C++ class interface definition. 'Inheritance' and 'has' relationships included in the diagram must be properly im-
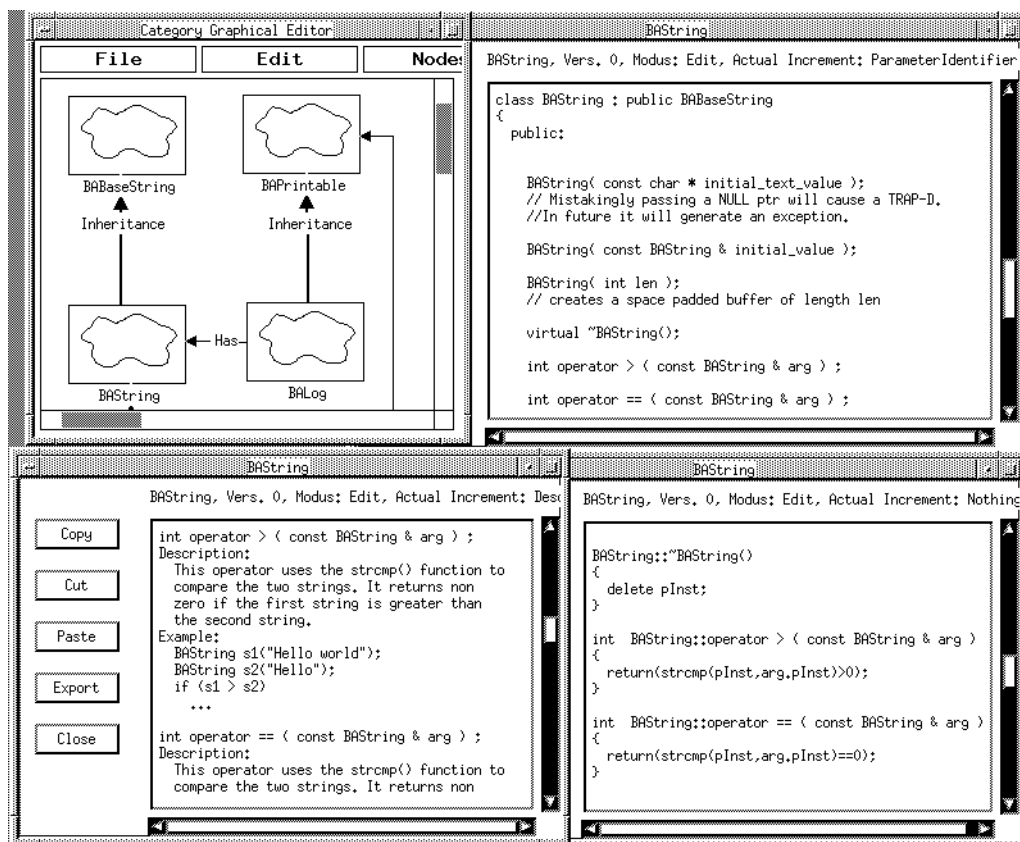

FIG. 2. User Interface of the BA SEE

plemented in the class interface and vice versa. Member functions declared in a class interface should be implemented in the implementation document of the class. The member function must be described in the documentation document of the class. In both cases, the signatures for the member function must match the ones in the class interface. There are also inter-document consistency constraints between different documents of the same type. Class forward declarations, which can be considered as imported classes, should be declared in another class interface document. `#include` preprocessor statements should denote filenames that have been defined for another class interface document. These constraints are only a few examples and in reality a much higher number of constraints exist between these four document types.

BA software engineers are not able to maintain all these constraints manually. Some constraint violations that involve C++ class interfaces and method implementations can be detected by the C++ compiler. Some are only detected by the linker (e.g. class forward declarations that have not been implemented). Both compiler and linker, however, are not particularly supportive in resolving inconsistencies, leaving these to the user. Moreover, the compiler can only be used if all imported classes are available and the document is complete. We agree with [49] that constraint violations should be detected at much earlier stages. Inter-document constraints beyond those implemented by the C++ compiler/linker are not checked at all. Tools for developing Booch diagrams, already on the market, can generate an initial code fragment for a C++ class, but after the generation has been completed, consistency between the Booch diagram and the C++ class interface is no longer maintained by these tools. Changes to a C++ class interface will not be reflected in the Booch diagram and, vice versa, incremental changes to a Booch diagram cannot be incorporated into the class interface without losing work spent on the generated interface. The situation for documentation is even worse, because there are no tools at all that support the BA documentation standard. Hence keeping the documentation of a class consistent with its interface definition is highly labour intensive and often not done, which renders the documentation useless. What is required is an environment, containing four integrated tools for the above document types that support the checking and preservation of the inter-document consistency constraints identified. There are several strategies to be considered for constraint handling.

CONSTRAINT ENFORCEMENT:
The straightforward strategy to handle constraints is to simply prevent users from introducing violations by rejecting erroneous input. Although this is inappro-

priate in many cases (see below), there are constraints that ought to be handled this way. As an example, consider names of class icons in Booch diagrams that represent class libraries. These class names should be unique within the scope of not only one but any diagram. If there were two classes with the same name contained in different libraries, these libraries could never be reused together. Typically different developers are in charge of different diagrams. Considerable communication between the developers would be required to resolve a constraint violation. In particular, a developer might not understand why his or her class name all of a sudden ceases to be unique. This overhead cannot be justified compared to just choosing another name when the tool has detected a duplicate. In cases like this, tools should therefore follow a strategy of *constraint enforcement* and reject erroneous input.

CHANGE PROPAGATIONS:
The above constraint enforcement is inappropriate when a part of a document must be changed that is already consistent with other parts in other documents. As an example, consider changing a name of a class icon in a Booch diagram. If the name is consistent with the C++ class interface it will become inconsistent after the change. We cannot enforce the constraint, because then we could never change the name of an existing class. To solve this dilemma, consistency can be retained by explicit actions of the interface tool, if it is informed about a change by the Booch tool. The interface tool might then take the necessary action and in the above example change the class name, the names of constructors and destructors and make appropriate modifications to all the types that were using the class. This relieves developers of the mundane and error-prone task of implementing the change manually in the interface. The class interface tool might then even inform tools working on transitively dependent documents, which are the implementation and documentation tools of the BA SEE, so that they can deal with the change. We refer to this mechanism as *change propagation*. It preserves consistency during changes by informing related tools to take explicit action.

Propagations can be seriously damaging if tools cannot ensure atomicity and durability of changes. As with many other tool commands, users clearly expect that propagations are either done completely, or not at all. Moreover, they expect that the effect of complete propagations are stored persistently so that they will not be affected by future failures.

VIOLATION TOLERATION:
Constraints will not be violated if tools only implement the constraint enforcement and change propagation strategies. There are, however, constraints that de-

velopers might want to violate temporarily. An example in the BA SEE is a class forward declaration. If developers are always forced to meet the constraint that the forward declared class must be declared in some other class interface, then they could not use classes to be designed in the future. As a consequence library design would have to be done strictly bottom-up. A top-down or hybrid strategy would be inhibited by the constraint handling strategy.

Again, we agree with [49] that this is undesirable. A *violation toleration* strategy seems more appropriate for constraints upon declarations and their applied occurrence. In general, any inconsistency between a declaration and an applied occurrence that can be rectified at the declaration should be tolerated. Hence, other violations that need to be tolerated include provision of the wrong number of actual parameters in a method call or the assignment of incompatible types. Tools should highlight temporary inconsistencies e.g. by using another colour or another font, so as to draw the user's attention to their existence.

2.2. Version and configuration management

In our experience, the management of reusable class libraries is impossible without proper version and configuration management. Library component documents are improved during library maintenance. Sometimes new components are added to meet new requirements or obsolete components are deleted. It is, however, essential that any release that has been given to a client project can be restored in the workspace of the reuse group. This is required, for instance, if a project detects a fault in a library that needs to be fixed urgently. Then the reuse department must be able to locate the fault within the release that the project is using. In general, it is not feasible to force projects to use the most recent release and, therefore, the reuse group may be asked to produce releases of a library, fixing a particular fault specifically for the project that has encountered it.

Available version and configuration management systems do not precisely meet the reuse group's configuration management needs. Systems such as SCCS [40] and RCS [45] can manage versions of files but they fail to support library releases, i.e. configurations of versioned files. Stand-alone configuration management systems, such as CCC [43] or PVCS [31] suffer from two main problems. Firstly, these CM systems are not aware of interdocument consistency constraints at all and put the burden of selecting consistent components completely on the user. Secondly, they demand maintenance of a redundant *component model*, which might be embedded in a physical design document that is produced anyway.

To solve these problems, version and configuration management should be tightly integrated with the tools of the BA SEE. The library architecture that is de-

fined within a Booch diagram serves as the component model for configuration management. Any class icon in the Booch diagram is considered to represent three component documents: the C++ class interface, the C++ method implementation and the class documentation. Each of these documents has a *version label*, that uniquely identifies the current version of the document. We attach the document's version labels as attributes to the respective class icons in the Booch diagram. The environment will interpret these version labels as version selection rules. Hence, a version of the Booch diagram represents a configuration of the library, and a diagram version must be kept for each release.

The Booch tool should then provide a variety of version and configuration management commands: It must support a means to freeze a consistent component. It should also be able to freeze all components identified by the diagram if they are consistent with each other. The other tools should respect this status and prevent users from modifying a frozen component. Users should be able to derive a successor version from a frozen component version. A new version label should be automatically computed during derivation of the successor version and be associated with the class icon. Navigation to components should be supported by the Booch tool in terms of opening commands. During execution of such a command it should pass the component's version label as a version selection rule to the interface, implementation or documentation tools. These tools should, in turn, respect the version semantics and apply changes only to the version identified by the label. In this way version labelling becomes fully transparent to the user. Restoration of a particular release then only requires the loading of the Booch diagram version that corresponds to the release.

2.3. Cooperative Work

Most software development processes are performed by multiple cooperating software engineers. This is also the case for the development of reuseable libraries at BA, where a group of six engineers cooperates. The cooperation model for these engineers is defined in a process model using the SLANG [5] formalism. The model is enforced by the SPADE process engine [4]. A full account of this modelling exercise is provided in [21].

No matter what cooperation policy is defined by a process model, tools have to implement it. The four editing tools of the BA SEE should, therefore, avoid imposing restrictions on the process model. In this subsection we argue that editors requiring tools to lock documents during whole editing sessions are too restrictive for the BA reuse process. Hence, the four editing tools need to be constructed in a way that tighter cooperation can be modelled and controlled by the process model.

The different libraries provided by the reuse group are not totally independent of each other. Classes defined in one library are used by classes contained in other libraries. Hence, developers responsible for the different libraries cannot work in complete isolation from each other, but have to cooperate on the development of the overall set of libraries maintained by the group. In particular, before a set of related library configurations can be released, the developers have to share their library configurations and must reach a state of inter-document consistency, so that the libraries can be compiled and tested. During this phase, developers want to use versions of classes from each other's working library configuration. They then expect to see the effect of each other's changes immediately.

The reuse group is, for instance, maintaining a library `BASQLXX` that contains classes to send SQL queries to relational databases. Some of these classes use class `BAString` contained in library `BALIBXX`, which is displayed in FIG. 2. Now assume that an extension of `BASQLXX` requires the introduction of a new function `upcase` in `BAString`. Then the developer in charge of `BASQLXX` asks his colleague responsible for `BALIBXX` to introduce this function. He, however, starts using `upcase` right away and the applied occurrences of `upcase` are, therefore, marked erroneous. He expects, however, the error marks to disappear as soon as his colleague has defined `upcase`. Similarly changes to existing member functions of a class, such as `BAString`, should be propa-

gated to classes in `BASQLXX`, even though affected classes might be edited concurrently. These change propagations retain inter-document consistency and relieve the developer responsible for `BASQLXX` from defining the required changes manually.

This style of computer supported cooperation is not available at present because tools apply strict locking policies that lock complete documents while they are being edited, or even worse lock them while they are in a working configuration, i.e. have been checked out from a common repository. On the other hand, some sort of concurrency control scheme must be implemented so as to avoid lost updates or inconsistent analysis problems, known from database systems [15], that can occur in software development tools as well [19]. Tools should, therefore, apply a smart concurrency control scheme that only locks those parts of documents that are accessed and releases locks as soon as possible.

3. Document Representation

Documents should be represented as *project-wide attributed abstract syntax graphs* (ASGs), as discussed in [22, 20]. We discuss how the requirements delineated above can be met using ASGs in this section. ASG nodes are attributed. Attribute values represent lexemes or semantic information such as references to a string table, symbol table or error list. We distinguish between *aggregation edges* in the graph, which
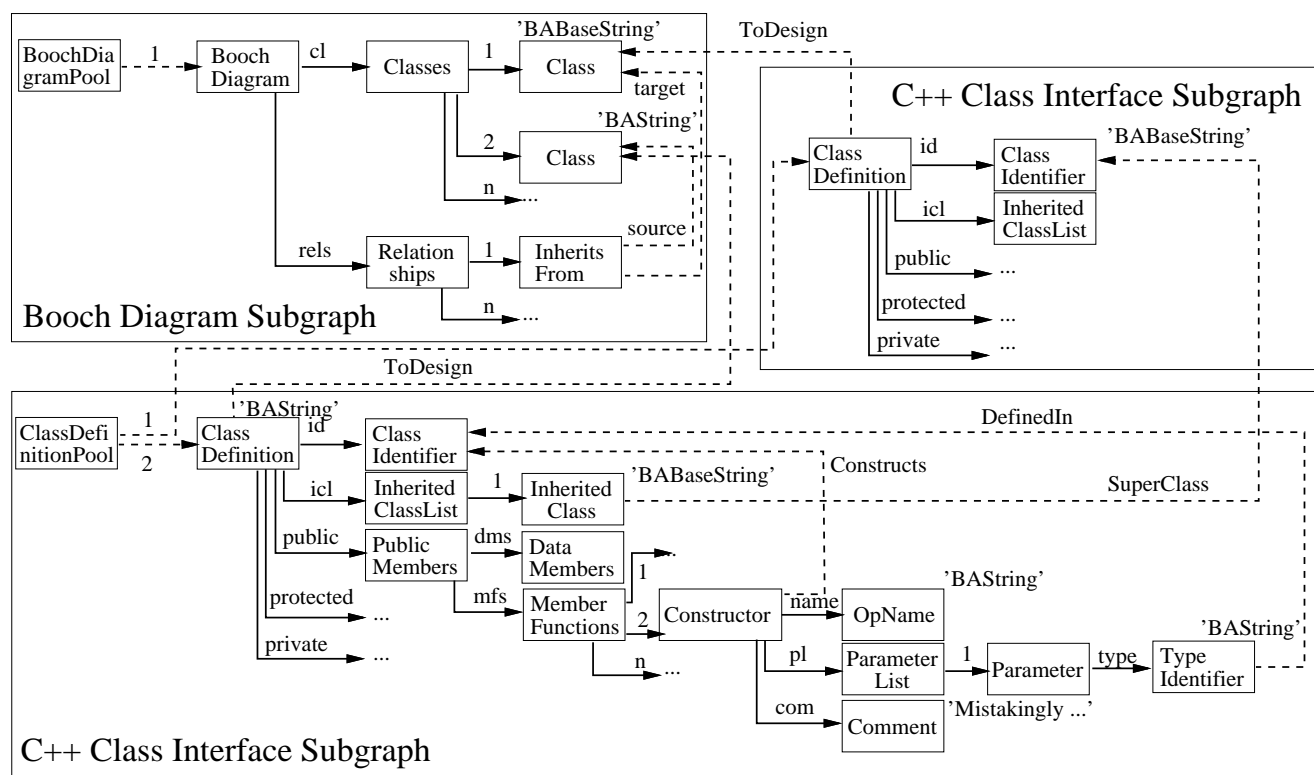

FIG. 3. Fragment of an Abstract Syntax Graph

implement syntactic relationships, and *reference edges*, which arise from semantic relationships. Documents are identified by the subgraph whose node-set is the transitive closure of nodes reachable by aggregation edges from document root nodes. Nodes that cannot have out-going aggregation edges are called *terminal nodes*, for they are derived from terminal symbols of the underlying grammar. Those nodes that may have out-going aggregation edges shall be called *non-terminal nodes*.

As an example, consider FIG. 3, which displays an excerpt of the ASG that represents documents displayed in FIG. 2. The figure shows three excerpts of document subgraphs. The subgraph in the upper left corner represents the graphical Booch diagram. The subgraph below represents the class interface of the C++ class `BAString` and the subgraph in the upper right represents the class interface of `BABaseString`, the super class of `BAString`. Attributes given in quotes at the upper right corner of a node representation represent lexemes. A number of attributes, like positional coordinates, error lists and symbol tables, are omitted for reasons of brevity. Aggregation edges are drawn with solid arrows and reference edges are displayed as dashed arrows. Although they are directed, edges are considered traversable in both directions. Edges are directed in order to determine the navigation direction that is designated by the edge name. For aggregation edges, the reverse direction has the implict name 'father' and for reference edges, names for the reverse direction can be explicitly defined. The two nodes of types `BoochDiagramPool` and `ClassDefinitionPool` serve as directories for the respective document types and are the starting point for queries that need to lookup a particular document of that type.

CONSTRAINT HANDLING STRATEGIES:

The various inter-document consistency constraint handling strategies identified above can be implemented efficiently on the basis of this document representation scheme. Implementation of constraint enforcement can be achieved by adoption of techniques known from compiler construction. Implementing checks for uniqueness of class identifiers, for instance, is a typical *name analysis problem* [32] and should be treated as such. Hence we attach a symbol table attribute to the node of type `BoochDiagramPool`. During insertion of a new class, the Booch tool can then perform a look-up in this symbol table in order to see whether the new name has been introduced before. If this is not the case, it can add a new class node to the list of class nodes, attach the designated class name to the lexeme attribute of that node and insert a new association under the key of the new class name into the symbol table. In the other case, it can reject the user request and perform no changes to the graph at all.

The implementation of change propagations exploits the reference edges of the graph. We first have to consider how these edges come into existence. Let us, therefore, continue the above example. After the Booch tool has successfully created a new class, the tool would request the class interface tool to create a new class interface subgraph. It would pass a reference to the class node in the Booch diagram subgraph to the interface tool. The interface tool would use this reference to establish the referenced edge `DesignedIn` between the new `ClassDefinition` node and the `Class` node in the Booch diagram subgraph. This reference is then exploited to implement the change propagations to be done during the changing of a class name. If such a change is done by the Booch tool, it would exploit the reverse direction of `DesignedIn` to find the affected class interface. If the change is initiated by the class interface tool, it would follow the original direction to find the affected node in the Booch diagram subgraph. Note that in both cases only constant time is required to traverse along an edge and find the affected node.

In order to achieve atomic and durable change propagations, the graph traversal and modification operations should be clustered together into atomic and durable execution units that we refer to as transactions. If a failure occurs during such a transaction, the ASG state should be recovered to the state that it had after the last completed transaction.

The constraint violation toleration strategy is the most difficult to achieve. It requires the maintenance of constraint violations that nodes are causing. From a structural point of view, each node must have an error set as an attribute. The elements of this set are error descriptors denoting particular constraint violations. The error set is exploited during computation of external document representation: The part of a document corresponding to a node with a non-empty error set is marked as erroneous. From a behavioural point of view, we will then have to maintain the attribute values of these error sets depending on the overall state of the ASG. This is particularly complicated to specify and implement efficiently.

We suggest a rule-based formalism to specify dependencies. We have described in [19] a way to generate a dedicated rule interpreter that exploits the dependencies between rules to perform incremental rule evaluation and thus achieves a better performance than a generic rule interpreter.

For reasons of brevity, we will have to restrict the discussion here to a sketch of the main idea based on an example. Consider the constraint in C++ that each constructor of a class must have the same name as the class itself. This constraint can be ex-

```
SPECIFICATION Constructor; ...
ON CHANGED(name.value) OR
   CHANGED(Constructs.value) DO
ACTION
  IF name.value == Constructs.value THEN
     Errors.remove(#ConstructorNameMismatch)
  ELSE
     Errors.insert(#ConstructorNameMismatch)
  ENDIF
END ACTION;
```

FIG. 4.   GTSL Rule Checking Constructor Names

pressed in our rule-based formalism in the form displayed in FIG. 4. It defines that whenever the name of the constructor (`name.value`) or the name of the class (`Constructs.value`) have been changed the error set attribute `Errors` will have to be modified. If the names match, the error descriptor `#ConstructorNameMismatch` will be removed from the set, otherwise it will be inserted. Note how the specification of this rule is based on attributes and edges of the ASG.

A locally controlled two-phase evaluation algorithm is generated from the dependencies determined by references in the `ON` clauses of the rules. The idea of the algorithm is as follows. Whenever an attribute is modified, a propagation phase marks all those attributes as 'dirty' whose nodes have rules that reference the modified attribute. Before a dirty attribute is accessed, an evaluation phase brings the attribute back into 'clean' state. This requires all attributes which the accessed attribute transitively depends on, to be brought into the clean state first. This is done by executing the action parts of the rules that modify the attributes. Then the attribute is itself brought back into the clean state by executing all rules that modify the attribute.

In the above example, the `Errors` attribute of a constructor node would be marked as dirty as soon as the lexeme attribute `value` of the constructor name, or the lexeme attribute value of the `ClassIdentifier` node, were modified. Before the `Errors` attribute is accessed the next time, e.g. during the computation of an external document representation, the attribute is brought into the clean state by determining the attribute value on the basis of the computations defined in the action part.

VERSION MANAGEMENT:
The granularity for version management is defined by the subgraphs of the ASGs that represent documents. The identification of these subgraphs is enabled by the distinction between aggregation and reference edges. In order to freeze such a subgraph we have to prevent the creation or deletion of nodes and aggregation edges as well as changes to the values of lexeme attributes. Note that other attributes, like error sets, as well as reference edges might have to be modified, even in a frozen version of a subgraph. As an example, consider the in-

troduction of a new subclass of a frozen class. This will require the creation of a reference edge `SuperClass` between an `InheritedClass` node and the `ClassIdentifier` in the frozen version of the subgraph representing the interface of the super class. A copy of a subgraph has to be created to implement the derive operation. We note that this copy should actually not be a physical copy, because that would be too inefficient in both space and time.

COOPERATIVE WORK:
When several users are cooperating on the development of related libraries, they are concurrently accessing a shared ASG representing all the libraries. The concurrency control scheme that we are looking for should avoid imposing restrictions on developers' work as far as possible and it must at the same time ensure the integrity of the graph in the face of concurrent updates. We claim that conventional ACID transactions [28], if they are used to implement single tool commands rather than complete editing sessions, are well suited to implement the required concurrency control.

Executing a tool command, such as insertion of a new member function template or expansion of a member function identifier template, will require a few hundred milliseconds [19]. Redisplaying changes done after such a command again requires a few hundred milliseconds. If we start a transaction before the command execution begins and commit it after the changes have been redisplayed, it will most likely last for less than one and a half second. Integrity preservation for concurrent updates is achieved by the locking that the transaction performs. It locks nodes that are being modified in exclusive mode and nodes that are being accessed in read mode. Read locks are compatible with each other while any other combination reveals a concurrency control conflict. Note that the number of nodes that will be locked during a transaction is very limited and the duration for which locks are being held is very short. In addition the tool will hold no locks at all when it is idle. Because different users are rarely working on the same libraries, the chances of concurrency control conflicts are fairly remote. If they occur, they can be resolved by delaying one conflicting transaction or even by aborting a conflicting transaction and restarting it again.

4. Managing Abstract Syntax Graphs in O_2

In this section, we discuss how ASGs as introduced in the previous section are implemented using the O_2 ODBMS as extended in GOODSTEP. Before we discuss these extensions and their exploitation for ASG implementation, we briefly sketch why we have cho-

sen ODBMSs rather than conventional data stores. A much more detailed review of available database support required for software engineering environments can be found in [6].

4.1. Choosing a DBMS

The ASGs introduced in the previous section must be stored persistently because they will have to survive editing sessions. Moreover, different developers will have to access the ASGs from different workstations, or the graph itself might even be distributed over several workstations. While these concerns could be addressed by storing graphs as files in a network file system, further considerations lead us to use database systems instead. Any full-blown database system supports conventional ACID transactions that are required to implement cooperative work, while network file systems lack this support. Moreover, database systems can protect the integrity of ASGs against hardware or software failures, while network file systems do not provide this. Finally, databases efficiently manage the transfer between higher-level data structures in main memory and their physical representation on secondary storage media, while file systems lack this secondary storage management.

We have chosen object database management systems (ODBMS) [2] rather than relational database management systems (RDBMS) for the storage of ASGs for the following reasons. The data definition languages available in ODMBSs can directly express graph structures, while graphs would have to be decomposed into tables for management by RDMBSs. The schema definition gets more complicated and the run-time performance suffers because RDBMSs have to compose the graph during external documentation computation by as many joins as there are node types to be accessed. Moreover, inheritance and the encapsulation of data definitions supported by the concept of classes in ODBMSs help to keep the complexity of ASG definitions manageable and also contributes to their maintenance. Finally, some object databases support version management, which we will use for versioning those subgraphs of ASGs that represent documents. There are no comparable mechanism available in RDBMSs.

Currently, there are about ten ODBMS products in the market. The most relevant of these are GemStone [14], ObjectStore [34], Versant [27], Ontos [1] and O_2 [3]. A significant step towards the standardisation of ODBMSs has been achieved by the object database management group (ODMG-93) [12]. The first ODBMSs that comply with this standard have been released. The standard defines a common object model, an object definition language (ODL), an object query language (OQL) and programming language bindings to C++ and Smalltalk so as to use them as object manipulation languages (OML). As we shall see later, these ODL and OMLs provide the expressiveness that is needed for expressing graph structures and graph traversal and modification operations. Capabilities that are neither standardised nor commonly available in ODBMSs include version management of composite objects and efficient object-level locking. These were added to the O_2 system during the course of the GOODSTEP project and are briefly discussed now.

4.2. Extensions to the O_2 System

4.2.1. Versions of Composite Objects

Two principles guided the design of the version management facility: generality and efficiency. As a component to be incorporated into a database system, the version manager should be of general applicability for a large variety of database applications that require version management. This implies that the design of the version manager should avoid imposing particular version or configuration management policies, but only provide the basic mechanisms that applications would use to define their own policies. The second guiding principle was efficiency in both time and space. The version management operations to be provided should avoid slowing down the overall application performance and at the same time the storage space of versioned application data should not increase linearly with the number of versions of application data to be maintained, but linearly with the number of differences among the different versions.

The version manager provides versioning operations that can be applied to a collection of objects, rather than to single objects. These collections are dynamically defined at run-time by insertion of single objects into a *version unit*, rather than statically in the schema. The slight performance overhead of this dynamic definition is out-weighed by the greater flexibility that it provides. In particular, the decision on the granularity for versioning can be postponed until after the schema has been defined and the same schema definitions can be used to manage versioned and non-versioned objects.

The version manager implements a *lazy object duplication strategy*, which imporantly contributes both to decreasing the time required for creation of a new version and reducing the space the new version occupies. Following this strategy, different versions of a version unit share the physical representation of any object included in the version unit as long as that object does not differ in the different versions. Hence, creating a new version does not require to create objects anew. Objects are rather considered as *multi-versioned objects*. As soon as a shared object is modified in one version,

however, a new copy of the object is created and registered within the multi-version object.

We note that this lazy object duplication strategy interferes with the concurrency control protocol of the database. Assume that an object is shared by two versions V1 and V2. If one transaction modifies the object in version V1, while some other transaction accesses the object in version V2, a split has to be made. This split, although being a modification to the multi-version object that is shared between the concurrent transactions, must not cause a concurrency control conflict. In that case, one of the major objectives for having versions, namely to isolate concurrent development with different versions of the unit for a certain period of time, cannot be achieved. Instead the transaction manager has to be integrated with the version manager so as to not consider splits as modifications. This leads to the general observation that version management with lazy object duplication can never be implemented on top of a database system that does not support version management.

The version manager is available to application programmers as a pre-defined class o2_Version. It offers operations to modify the contents of the version unit associated to an instance of the class by inserting or deleting objects. It provides operations for deriving, comparing, merging and selecting particular versions of this version unit. Moreover, the class maintains a version history graph and provides operations for querying and navigating through the graph.

4.2.2. Object-level Concurrency Control

O_2 has a client/server architecture, where the server is in charge of concurrency control and storing pages on disk, while the mapping of objects to pages as well as the execution of schema and version management are performed by the client. Clients and server exchange pages that contain, in the case of the BA SEE, multiple objects because objects implement fairly small-sized ASG nodes. A clustering mechanism [8] can be adjusted in a way that related objects reside on the same page and consequently the network overhead involved in transferring objects to the client is minimised.

Originally not only client/server communication, but also concurrency control, which implements ACID transactions, was page-based since the server was not aware of the objects that resided on the pages it was managing. We refer to this page-level concurrency control as mode C_A_R hereafter. With a number of small objects residing on a page, situations might occur where the page-level concurrency control reveals conflicts though the concurrent transactions were accessing disjoint sets of objects. The conflicts occurred just because there were objects in the sets which by chance resided on the same page.

O_2 has been extended in GOODSTEP with object-level concurrency control to resolve this undesirable behaviour. In the new mode OC_A_R concurrency control is, by default, still page-based. As soon as a conflict occurs, however, the concurrency control switches to object-level locking that is then implemented jointly by server and client. It only reveals conflicts if the locks acquired by transactions are really incompatible. The extension is, therefore, done in a way that the benefits of smaller network overhead, achieved by page-level client/server communication and page-level concurrency control, are retained as far as possible, but concurrency control conflicts are restricted to concurrent transactions that actually do conflict.

4.3. Exploitation of the Extended O_2

4.3.1. Schema Generation

The object database schemas implementing ASGs for complicated languages, such as Booch diagrams, C++ class definitions and method implementations, tend to become rather complex. One reason for this is that even powerful object definition and manipulation languages as they are provided by ODBMSs do not offer the right level of abstraction for the highly application specific problem of defining ASGs. Rather they have to be considered as persistent object-oriented programming languages that serve general purposes. The approach taken in GOODSTEP was, therefore, to design the GOODSTEP tool specification language (GTSL) [20] as an application specific schema definition language that can appropriately express ASG structures and operations. The desired object-database schemas for ASGs are then derived by the GTSL compiler from these high-level specifications during code generation. The GTSL compiler itself has been generated with relatively little effort [19] using the Eli compiler construction toolkit [29].

GTSL is a multi-paradigm language. It combines object-orientation, rules and patterns to appropriately address the different concerns that arise during specification of ASGs. An environment specification is structured into a number of *tool configurations*. Each of these consists of a number of *classes* that define the different node types that occur in the subgraph corresponding to the document type the tool is intended for. Different *sections* are provided to define properties of a class. GTSL provides *attribute, abstract syntax* and *semantic relationship sections* to define structural properties, which are node attributes, out-going aggregation edges and out-going reference edges. The lexical syntax that lexeme attributes attached to terminal nodes must obey is defined in the form of regular expression patterns in *regular expression sections*. An *unparsing section* is available to define the mapping between external document representation and the ASG. This mapping is

defined in terms of patterns. Three behavioural properties can be defined for a class. The available operations to modify graph nodes are defined in a *method section*. The invocation of operations from commands and their availability are specified as patterns in *interaction sections*. Finally, inter-document consistency constraints are defined in a rule-based manner in *semantic rule* sections, which we have, in fact, discussed above already. *Multiple inheritance* is supported so as to facilitate reuse of properties.

The tools for the BA SEE have been specified with GTSL. Let us consider a small excerpt from this specification. We will then discuss the ODL and OML code that has been derived from the specification as the object database schema for the BA SEE tools. FIG. 5 displays fragments of GTSL classes, some of which specify node types that were used in FIG. 3.

GTSL supports the concept of abstract classes, which specify common properties of all their subclasses. Abstract classes cannot be instantiated. The most general class is `Increment`. It defines four attributes, which any other node inherits. The first of these is the error set that we discussed already. Moreover, it defines an attribute for deciding whether a node represents a placeholder or is expanded. Furthermore, it defines the attribute `father` which refers to a node's father node in the abstract syntax tree and, finally, it defines an attribute `doc_ver`. This refers to the root node of the sub-

```
ABSTRACT INCREMENT INTERFACE Increment;
 ATTRIBUTES
  Errors:SET OF ERRORS;
  expanded:BOOLEAN;
  father: Increment;
  doc_ver: DocumentVersion;
 END ATTRIBUTES;
END ABSTRACT INCREMENT INTERFACE Increment.
```

```
TERMINAL INCREMENT INTERFACE ClassIdentifier;
 INHERIT Increment;
 REGULAR EXPRESSION
  {[A-Za-z][A-Za-z0-9_]*}
 END REGULAR EXPRESSION;
 SEMANTIC RELATIONSHIPS
  IMPLICIT Constructors:SET OF Constructor.Constructs;
  IMPLICIT UsedBy : SET OF Type.DefinedIn;
 END SEMANTIC RELATIONSHIPS;
 METHODS
  IMPLICIT METHOD scan(Str:STRING):BOOLEAN;
  METHOD ChangeName(Str:STRING):BOOLEAN;
 END METHODS;
END TERMINAL INCREMENT INTERFACE ClassIdentifier.
```

```
NONTERMINAL INCREMENT INTERFACE MemberFunctions;
 INHERIT Increment;
 ABSTRACT SYNTAX
  l : LIST OF MemberFunction;
 END ABSTRACT SYNTAX;
 UNPARSING SCHEME
  (NL), l DELIMITED BY (NL) END;
 END UNPARSING SCHEME;
 METHODS
  IMPLICIT METHOD expand();
  IMPLICIT METHOD parse(Str:STRING):MemberFunctions;
  IMPLICIT METHOD unparse():STRING;
 END METHODS;
END NONTERMINAL INCREMENT INTERFACE MemberFunctions.
```

```
ABSTRACT INCREMENT INTERFACE MemberFunction;
 INHERIT Increment;
 ABSTRACT SYNTAX
  pl : ParameterList;
  com : Comment;
 END ABSTRACT SYNTAX;
 METHODS
  DEFERRED METHOD expand();
  DEFERRED METHOD parse(Str:STRING):MemberFunction;
  DEFERRED METHOD unparse():STRING;
  METHOD ExtendParamList();
 END METHODS;
END ABSTRACT INCREMENT INTERFACE MemberFunction.
```

```
NONTERMINAL INCREMENT INTERFACE Operator;
 INHERIT MemberFunction;
 ABSTRACT SYNTAX
  virt : OptVirtual;
  type : Type;
  os   : OperatorSign;
  const: OptConst;
  ass  : OptAssign;
 END ABSTRACT SYNTAX;
 UNPARSING SCHEME
  virt, type, "operator", (" "), os, (" "), "(",
  pl, ")", (" "), const, ass, ";", (NL),
  com, (NL)
 END UNPARSING SCHEME;
 METHODS
  IMPLICIT METHOD expand();
  IMPLICIT METHOD parse(Str:STRING):Operator;
  IMPLICIT METHOD unparse():STRING;
 END METHODS;
END NONTERMINAL INCREMENT INTERFACE Operator.
```

```
NONTERMINAL INCREMENT INTERFACE Constructor;
 INHERIT MemberFunction;
 ABSTRACT SYNTAX
  name : OpName;
 END ABSTRACT SYNTAX;
 UNPARSING SCHEME
  name, "(", pl, ")", ";", (NL),
  com, (NL)
 END UNPARSING SCHEME;
 SEMANTIC RELATIONSHIPS
  Constructs : ClassIdentifier;
 END NON SYNTACTIC EDGES;
 METHODS
  IMPLICIT METHOD expand();
  IMPLICIT METHOD parse(Str:STRING):Constructor;
  IMPLICIT METHOD unparse():STRING;
 END METHODS;
END NONTERMINAL INCREMENT INTERFACE Constructor.
```
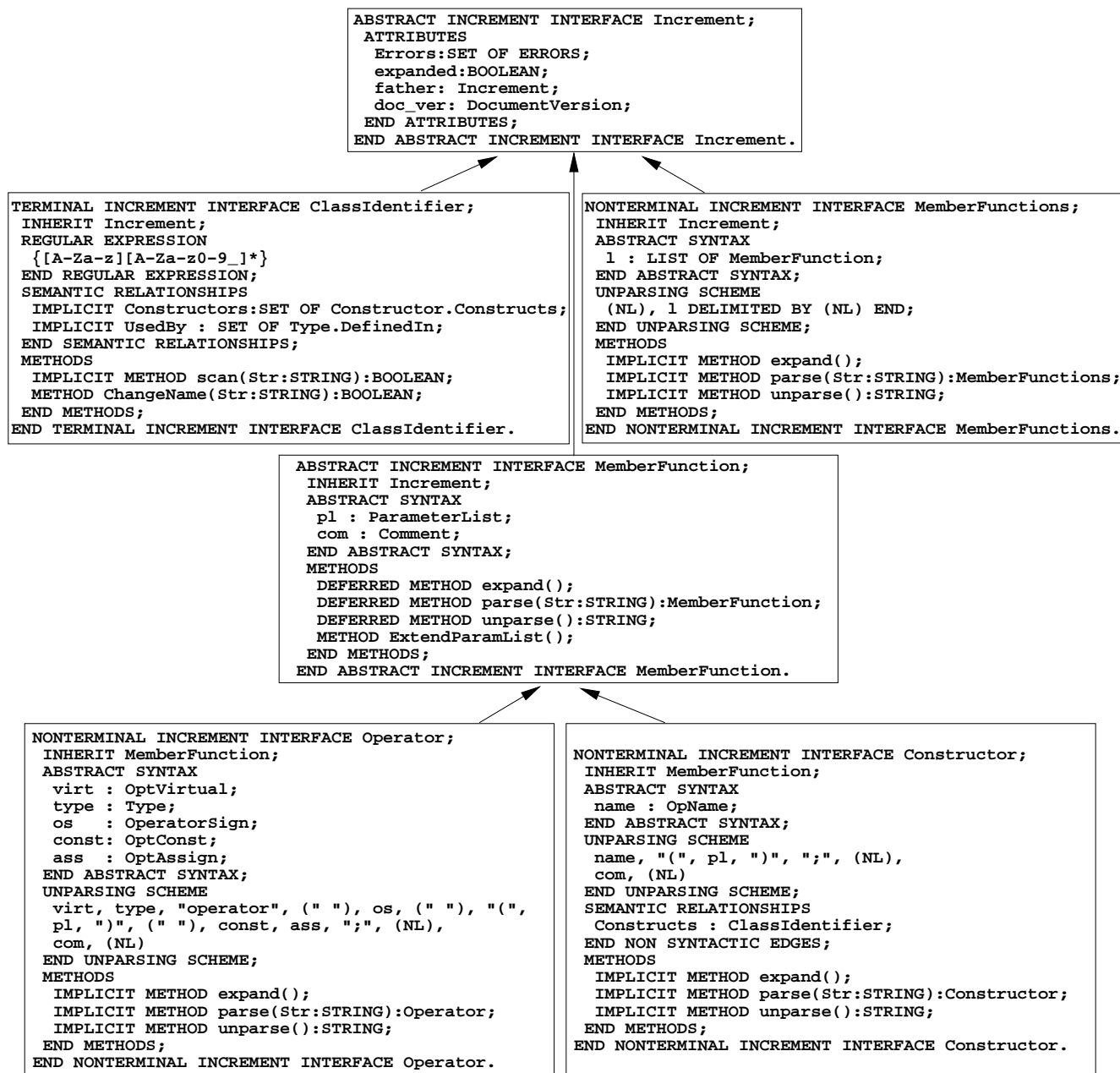
FIG. 5. Fragments of GTSL classes

```
SPECIFICATION ClassIdentifier; ...
METHOD ChangeName(Str:STRING):BOOLEAN;
BEGIN
  IF SELF.scan(Str) THEN
    FOREACH con:Constructor IN Constructors DO
      con.name.react_to_change(Str)
    ENDDO; ...
    value:=Str;
    RETURN(TRUE)
  ELSE
    RETURN(FALSE)
  ENDIF;
END ChangeName;
```

FIG. 6.   GTSL Method Performing Change Propagation

graph that represents the document the node belongs
to.

Class `MemberFunctions` defines the node type of the
same name in FIG. 3. Its abstract syntax section spec-
ifies an ordered multi-valued aggregation edge directed
to member function nodes. In C++ these can be op-
erators, constructors, destructors and ordinary meth-
ods. We define this heterogeneity with polymorphism.
Instances of any subclasses of `MemberFunction` may be
inserted into the list `l`. The abstract syntax section of
class `MemberFunction` defines two aggregation edges `pl`
and `com`. They are defined in class `MemberFunction` be-
cause any C++ member function can have a parameter
list and a comment. Thus, its subclasses inherit these
definitions. They add specific abstract syntax defini-
tions in their abstract syntax sections.

As an example for the specification of a reference
edge, consider the edge between constructor and class
identifier nodes. It is specified as a pair of *links* in the
semantic relationship section of the two classes. The
explicit link `Constructs` denotes the original direction
and the *implicit link* `Constructors` is used to address
the reverse direction of the edge.

The unparsing sections of classes specifying non-
terminal node types define the mapping between nodes
and the external representation of the nodes. The pre-
cise semantics is of no concern here and we refer the
interested reader to [19].

Methods are the means to modify the graph and can
be *implicit, explicit* or *deferred*. A deferred method is
only declared in an abstract class and has to be re-
defined in all subclasses, either by implicit or explicit
methods. An implicit method is a programming inter-
face to the other sections declared for a class. Their
bodies are generated by the GTSL compiler. The `scan`

```
void Constructor::expand() {
  if (name == NULL) name = new OpName(this);
  if (pl == NULL) pl = new ParameterList(this);
  if (com == NULL) com = new Comment(this);
  expanded = true;
};
```

FIG. 7.   OML Code Generated for GTSL Implicit Methods

method in class `ClassIdentifier`, for instance, checks its
argument for conformance with the regular expression
defined for a terminal node type. The `expand` method
performs a placeholder expansion and creates nodes for
all abstract syntax children and assigns them to the
children defined in the abstract syntax section. The
`parse` method tries to construct a subgraph from its
character string argument according to the unparsing
section. It returns a reference to the new subgraph if
the string parsed as argument conforms to the grammar
that is induced by the unparsing sections. The `unparse`
method performs the inverse operation and returns a
textual representation of a subgraph as defined by the
unparse section. The tool builder may then apply these
implicit methods in explicit methods. As an example
for an explicit method consider method `ChangeName` of
class `ClassIdentifier` in FIG. 6. It uses `scan` to check
for conformance of the new class name to the regular
expression and then propagates the change to applied
occurrences, for instance constructor names.

The translation of GTSL classes into ODL class in-
terfaces is straightforward. Each GTSL class is trans-
lated into an ODL interface definition. ODL sup-
ports multiple inheritance, as GTSL does. GTSL at-
tributes of atomic types are translated into ODL class
attributes. Attributes whose types are other classes,
abstract syntax children and links of semantic relation-
ships are translated into ODL relationships. Implicit,
explicit and deferred methods are translated into ODL
operations. In addition to the defined methods, each
ODL class interface defines a constructor `init`, which
initialises objects upon creation. An *extent* is defined
for each class that will include references to all persis-
tent objects of that class. The result of the application
of this translation process to the classes displayed in
FIG. 5 is shown in FIG. 8.

A more complicated problem is the generation of
implementations for the implicit methods, those which
implement placeholder expansion, parsing and unpars-
ing as well as the semantic rule evaluation algorithm,
we discussed in the previous section. The target lan-
guage for all these methods is the object manipulation
language (OML). To describe this in detail is beyond
the scope of this article; we refer the interested reader
to [19].

As an example, consider the implementation of the
`expand` method of class `Constructor` given in the C++
OML language binding in FIG. 7. It creates a new ob-
ject for each abstract syntax child that has not been
created before and modifies the status of attribute
`expanded` to reflect the state that the node no longer rep-
resents a placeholder. The OML code for the explicit
methods is generated by syntax-directed code genera-
tion techniques that are appropriately supported by the
compiler construction toolkit Eli and their implementa-
tion is straightforward.

```
interface Increment {
  extent __Increments;
  attribute set<ERRORS> Errors;
  attribute boolean expanded;
  relationship Increment father;
  relationship DocumentVersion doc_ver;
  init(in Increment);
}
```

```
interface ClassIdentifier:Increment {
  extent __ClassIdentifiers;
  attribute string value;
  relationship set<Constructor> Constructors
              inverse Constructor::Constructs;
  relationship set<Type> UsedBy inverse Type::DefinedIn;
  init(in Increment);
  boolean scan(in string)
  boolean ChangeName(in string);
}
```

```
interface MemberFunctions : Increment {
  extent __MemberFunctionss;
  relationship list<MemberFunction> l;
  init(in Increment);
  expand();
  MemberFunctions parse(string Str);
  string unparse();
}
```

```
interface MemberFunction : Increment {
  extent __MemberFunctions;
  relationship ParameterList pl;
  relationship Comment com;
  init(in Increment);
  expand();
  MemberFunction parse(in string);
  string unparse();
  ExtendParamList();
}
```

```
interface Operator:MemberFunction {
  extent __Operators;
  relationship OptVirtual virt;
  relationship Type type;
  relationship OperatorSign os;
  relationship OptConst const;
  relationship OptAssign ass;
  init(in Increment);
  expand();
  Operator parse(in string);
  string unparse();
}
```

```
interface Constructor : MemberFunction {
  extent __Constructors;
  relationship OpName name;
  relationship ClassIdentifier Constructs;
  init(in Increment);
  expand();
  Constructor parse(in string);
  string unparse()
}
```

FIG. 8.   Class Interface Definitions in ODL

4.4. Version and Configuration Management

GTSL has a number of pre-defined classes. Their implementations are part of the GTSL run-time environment. The GTSL class `DocumentVersion` is such a pre-defined class. Its purpose is to serve as a superclass for all GTSL classes whose instances represent root nodes of those subgraphs of the project-wide abstract syntax graph that represent versionable documents.

The implementation of GTSL class `DocumentVersion` in ODL defines a relationship to an instance of class `o2_Version` which always refers to the current version. It also defines an attribute to store the information that the document version is considered frozen, or not. It then defines a number of operations for version management. These are inherited by all subclasses. FIG. 9 displays the ODL class definition for the implementation of GTSL class `DocumentVersion`.

The inclusion of objects that represent nodes of versionable subgraphs in the version unit, which is associated by `DocumentVersion` to the subgraph, is implemented in the constructor of the most general class `Increment`, as displayed in FIG. 10. Additional initialisations have to be done to initialise the root node

properly. These are implemented in the constructor of class `DocumentVersion`. It creates a new instance of class `o2_Version`, thereby creating a new version history graph, then defines the name of the root version to be the parameter `verName`, then identifies that it is not frozen and finally inserts the newly created root node in the version unit so that it will be itself under version control.

The operations of classes `DocumentVersion` are implemented merely by calling the respective operations of `o2_Version`. As an example, consider the implementations of `DeriveVersion` and `SelectVersion` below. They are implemented by using the `retrieve`, `set_label`, `select`, `set_default` and `derive` operations provided by `o2_Version`. `DeriveVersion`, displayed in FIG. 11, creates a new successor version of the current version and gives the new version the name passed as parameter. The aim of `SelectVersion` is to explicitly select a new version. All changes done after a version selection to the subgraph representation in terms of OML operations will be applied only to that selected version.

In this way, the implementation of version management of subgraphs in the GTSL run-time environment is achieved in about 800 lines of OML code. In addi-

```
interface DocumentVersion : Increment {
  readonly attribute boolean Stable;
  relationship o2_Version CurrentVersion;

  init(Increment f,string doc, string root_ver);
  o2_Version GetRootVersion();
  string GetRootVersionName();
  string GetVersionName();
  string GetDefaultVersionName();
  SetDefaultVersion(string ver_name);
  SelectVersion(string ver_name);
  SelectDefaultVersion();
  MergeVersions(string nameToMerge, string newName);
  boolean DeleteVersion(string name);
  DeriveVersion(string new_ver);
  FreezeVersion();
  set<string> AllVersions();
  set<string> GetChildren(string ver_name);
  set<string> GetParents(string ver_name);
  set<string> GetChildrenClosure(string ver_name);
  set<string> GetParentsClosure(string ver_name);
  boolean AreOnDifferentPaths(v1:string, v2 : string);
  boolean IsVersionStable(string ver_name);
  boolean IsStable();
}
```

FIG. 9.   Interface of Class `DocumentVersion`

```
Increment::Increment(Increment f) {
  father = f; expanded = false;
  if (f!=NULL) {
    doc_ver= f->doc_ver;
    doc_ver->CurrentVersion->append(this);
  }
}
DocumentVersion::DocumentVersion(string name,
                 string verName):Increment(NULL) {
  CurrentVersion = new o2_Version("Document: "+name);
  CurrentVersion->set_label(verName);
  Stable = false;
  CurrentVersion->append(this);   ...
}
```

FIG. 10.   Appending ASG Nodes to the Version Unit

tion, this code is generic and can be reused in any tools that access documents stored as subgraphs of ASGs in the O_2 ODBMS.

4.5.   Concurrent Tool Commands

As argued above, the granularity of concurrency control should be that of tool commands rather than that of editing sessions. GTSL provides the concept of *interactions* to specify commands. The definition of an interaction encompasses an internal and an external name, a selection context, a precondition and an action. The external name appears in context sensitive menus or is used to invoke a command from a command-line. The internal name is used to determine the redefinition of an inherited interaction. The selection context defines which increment must be selected so that the interaction is applicable. It is actually included in a context-sensitive menu if the precondition that follows the ON

```
void DocumentVersion::DeriveVersion(string new_ver) {
  o2_Version new,current;
  current=CurrentVersion;/* save selection in current*/
  new=CurrentVersion->derive();/* create new version */
  new->select                 /* select it and change*/
  CurrentVersion=v;          /* value of CurrentVersion*/
  new->set_label(new_ver);   /* label the new version*/
  current->select;       /* and restore old selection*/
}
void DocumentVersion::SelectVersion(string ver_name) {
  o2_Version v;
  v=self->RootVersion->retrieve(ver_name)
  if (v!=NULL) {              /* if ver_name exists */
    v->select();              /* select it as current*/
  }
}
```

FIG. 11.   Version Derivation & Selection

```
SPECIFICATION Class; ...
INTERACTION ChangeClassIdentifier;
NAME "Change Class Name"
SELECTED IS SELF
ON expanded
VAR new_name:TEXT;
    errors:TEXT_SET;
BEGIN
  new_name:= NEW TEXT(SELF.value);
  WHILE (new_name.LINE_EDIT("Change identifier:")) DO
    IF (SELF.ChangeIdentifier(new_name.CONTENTS())) THEN
      errors:=NEW TEXT_SET(SELF.get_set_of_errors());
      errors.DISPLAY();
      ABORT;
    ENDIF
  ENDDO;
END ChangeClassIdentifier;
```

FIG. 12.   Interaction of Booch Diagram Editor

clause evaluates to TRUE. The action is a list of GTSL statements that is executed as soon as the user chooses the command from the menu. An example showing the definition of a command offered by the Booch diagram editor that is used to change the class name is shown in FIG. 12.

The command defined by that interaction will be added to the menu of applicable commands if a class icon has been selected in a Booch diagram and if the class name has been expanded before. If the user chooses the command from the menu, the body will be executed. Then a new text dialog object is created and this object is used to display a line edit window. This prompts the user to change the class identifier, with the previous identifier as the default. If the user completes the dialogue, the method ChangeIdentifier will be executed. This method checks the new identifier for lexical correctness, performs all the required inter-document consistency checks and returns TRUE if everything is correct and false otherwise. Then a detailed error message is computed and shown to the user, the command is aborted and all changes done during the command execution are undone.

GTSL interactions are implemented as conventional O_2 transactions to meet the requirements of cooperative work discussed above. If the user has chosen a command, a new transaction is started. We note that locking need not be defined in the tool specification, nor be generated by the GTSL compiler, but is performed transparently by O_2. A transaction commit will be executed if the last statement of the interaction has been executed. GTSL `ABORT` statements are implemented as transaction aborts.

The change from page-level to object-level concurrency control proves particularly appropriate for the implementation of concurrent tool commands. ASG nodes are implemented as objects which are of fairly small granularity. Hence quite a number of ASG nodes reside on the same server page. Without page-level locking, concurrent tool commands reveal concurrency control conflicts, even though they are accessing disjoint sets of nodes. This is remedied with object-level locking. As the concurrency control strategy is determined during server startup, not a single change was needed to the GTSL compiler or the GTSL run-time environment in order to implement the transition from page to object-level locking.

5. Evaluation

5.1. Environment Generation

The full GTSL specification of the BA SEE consists of some 120 GTSL classes. The overall size of the specification is 12,000 lines of GTSL. The ODL/OML schema implementation for Booch diagrams, C++ class interfaces, implementations and documentation that has been generated from this specification consists of some 105,000 lines of ODL/OML code. Moreover, the command interpreters and the tool-specific parts of the user interface, which have also been generated from the GTSL specifications cover further 70,000 lines of C++ code. In addition, each tool uses a tool kernel, which consists of reusable classes that do not vary from tool to tool. This kernel contains a further 40,000 lines of C++ code. Hence, this environment is composed of 215,000 lines excluding the user interface management system and the object database system.

The commercially available Opus environment [24] is of similar complexity to the BA SEE, since it also integrates tools for three textual and one graphical document type. The functionality of Opus, however, is less than the BA SEE since it does not support version management, cooperative work and lacks support of free textual input and constraint violation toleration. Opus has been hand-coded and uses the GRAS database [35], which must be considered less power-

ful than object databases. Although less powerful, the amount of code in Opus is larger than that generated for the BA SEE, let alone the BA SEE specification. Opus consists of 280,000 lines of C code. This comparison provides evidence that the approach of using an application specific schema definition language and a powerful object database system simplifies the problem of tool construction considerably.

5.2. Run-Time Performance

To assess whether the generated BA SEE meets the end-user performance expectations, we performed a controlled experiment with the BA SEE. The experiment was performed using a single processor Sun SparcStation 10/40 with 64 MBytes of main memory that operates with SunOS 4.1.3. The database was stored on a local 2 GBytes disk. It contained the schema and the production configuration of the `BALIBXX` library with some 80 classes, their interfaces, implementation and documentation. We gave O_2 a caching allowance of 2 MBytes for each tool client process and another 1 MByte for the database server process. The clients and the server were executed on the same machine and the database was in a warm state, i.e. all objects accessed during the experiment had been accessed before. We used the experiment also to compare the performance of the two concurrency control scheme implementations. The experiment consisted of six activities:

Dumping: The action that dominates the response time while opening a document is the time required to perform a traversal along the transitive closure of the aggregation edges from a document root node so as to compute the external representation of the document. The subgraph that we use in this activity represents the largest class interface definition of the the `BALIBXX` library. Its textual representation has 220 lines of code and the respective ASG consists of 507 nodes.

Version Derivation: This activity measures the time required for the derivation of a new version. This new version is the only successor version of the root version. The activity also covers committing the change to the database.

Version Selection: This activity measures the time required for the selection of the version derived during the last activity.

Template Insertion: During this activity we measure the time for template insertion. As an archetypical example we measure the time that the tool needs to insert a parameter template into a list of parameters that currently includes three parameters. The time not only includes the required ASG modifications, but also the time taken to insert new ASG nodes into the version unit, the time for redisplaying the contents of the affected window and the time for transaction start-up and commit.

Static Semantic Check: During this activity we explore the performance of commands that perform static semantic checks. As an example, we expand the name identifier of a previously expanded parameter. The time we measure includes checking the lexical correctness of the identifier, storing the value in a lexeme attribute, checking the uniqueness of the identifier in the parameter list, incrementally redisplaying the affected window and transaction start-up and commit.

Inter-Document Consistency Check: The purpose of this activity is to measure the performance of the creation of a new dependency relation between two different documents. For that purpose we have chosen a forward declaration of a class, that already exists. The measured time includes the check as to whether the referenced class exists, the creation of a reference edge, the class identifier node and the forward declaration, the incremental redisplay of the affected document parts and transaction start-up and commit.

FIG. 13 depicts the performance figures for the two concurrency control implementations in milliseconds of elapsed real-time. To retrieve the figures, we have repeated each activity 10 times and the figures represent the mean values of these 10 executions. The observed deviations were neglectable. The performance of **Dumping** is about 1,400 milliseconds for page-level and 1,800 milliseconds for object-level concurrency control. This is a reasonable performance that means that processing each node required on average less than four milliseconds with object-level locking. If all error attributes are in clean state, this activity need not perform any changes. This is the more frequent case and we could exploit O_2's read-only transactions, which do not perform locking at all. A further experiment with these transactions has shown that they reduce the time to 1,050 milliseconds because these do not acquire read-locks. The **Version Selection** and **Version Derivation** activities are performed in both concurrency control modes for the complex class interface definition in less than 300 milliseconds, which users will hardly ever recognise as a delay. **Template Insertion** is performed with page-level concurrency control in 900 milliseconds, while 1,250 milliseconds were required with object-level locking. This is about the time that they require to move the hand from the mouse to the keyboard and users will find the performance acceptable. The **Static Semantic Check** activity required 1,300 milliseconds with page-level concurrency control, while 1.650 milliseconds were required with object-level locking. 1,500 milliseconds were required with page-level locking for the **Inter-Document Consistency Check**, while 1,800 milliseconds were required with object-level locking. The performance of 1,800 mil-
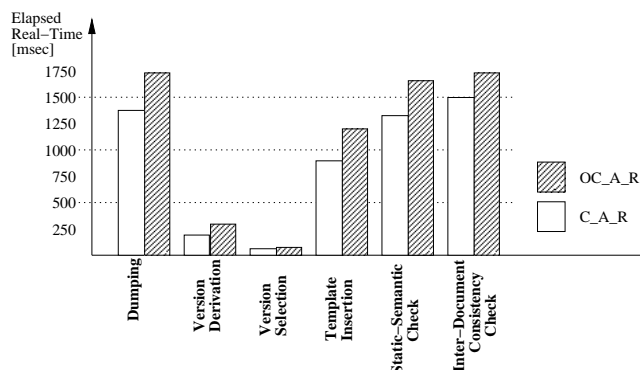

FIG. 13. Elapsed Real Time during Activities [msec]

liseconds for a tool command that involves an inter-document consistency check, however, is too slow.

The experiment was conducted on a rather old machine with only one processor, a SCSI-I disk and a slow clock rate. With performance of hardware and operating systems increasing by 100% every two years, we expect to have acceptable response times of tools available shortly. Moreover, we note that there is an overhead of 20-30% in object-level locking compared to page-level locking, even though no conflicts occurred at all during this experiment. This overhead traces back to the prototype implementation of OC_A_R that is not yet optimised to the case when no page conflicts occur. In the future product version, this optimisation will be in place and object-level locking without page conflicts will perform as fast as page-level locking.

5.3. Acceptance in Industry

British Airways did not deploy the BA SEE developed in GOODSTEP. One of the reasons was that the practical impact of the deployment was too radical. The BA SEE contains completely new development tools, such as the Booch editor and the C++ class interface editor, and explicit process constraints, e.g. library code cannot be modified if the corresponding Booch diagram is not modified first. In addition, the environment could not be purchased from a trusted vendor; it was just a prototype developed by a research consortium that lacked, for instance, stability, a sophisticated user interface with keyboard short cuts, facilities for macros and the like.

Another drawback of the architecture of the environment is that it does not properly address integration with foreign tools. British Airways was using the Rose product from Rational and they were quite happy with its user interface and its editing capabilities for designing classes in the Booch notation. They would have preferred an integration between Rose and the C++ and documentation tools. As Rose does not store its documents in the same object database that we use to

store ASGs, an integration has to be achieved using non-database mechanisms.

6. Related Work

The idea of tool generation came up during the early eighties in a number of projects including Gandalf [30], Centaur [11] and the Cornell Synthesizer Generator [39]. The tools generated by these systems are tools for programming environments, that means they were intended to support documents in one language only. Therefore, they initially did not address the problem of inter-document consistency constraints. [26] suggested use of different views to represent different documents, but this inhibits inter-document consistency constraint violations. Moreover, none of these early tools provide sufficient support for concurrency control, but mostly store documents in a flattened representation in the file system. In view of concurrent tool execution, this could result in loss of changes. Neither version nor configuration management is explicitly supported by any of these tools.

The IPSEN environment [23] was among the first environments that considered inter-document consistency. The specification of this environment was based on graph grammars. Recently, a graph grammar interpreter and compiler have been completed [50]. These simplify environment construction in the same way as our GTSL compiler. IPSEN provides facilities for revision control [48], though configuration management has not yet been addressed. The most serious drawback of IPSEN, compared to the BA SEE, is its lack of support for cooperative work. The reason is that it has been built using the home-grown database system GRAS [35], which does not yet support the required fine-grained concurrency control protocol, but applies strict locking to a complete graph, rather than locking only those nodes that are being accessed.

Tools contained in the Field environment [38] are integrated using a broadcast message server. This integration technique can achieve inter-document consistency constraints and even change propagations. Version management and concurrency control required to support cooperative work, however, are not supported in Field.

The UQ editor family [47, 33] supports different document types as different views of the same conceptual representation. These views implicitly implement change propagations. The user interaction paradigm is recognition-based as opposed to the structure-oriented paradigm in the BA SEE. We acknowledge that this recognition-based interaction paradigm provides better user support. The UQ editors, however, do not yet address version and configuration management and team cooperation support.

Documents managed by the Software through Pictures environment [46] are stored in the relational database Sybase and inter-document consistency constraints are managed by relations between these documents. Specific tools can be defined on the basis of a query and reporting language, which is interpreted by the StP/Core environment. Sybase supports ACID transactions, which could be exploited for concurrency control. However, as Sybase does not support version management and due to the fact that version management cannot reasonably be added on top of a database, Software through Pictures, does not provide any support for managing different versions of documents. Sybase may be appropriate for storage of graphical documents, whose syntax graphs tend to be fairly small. We, however, doubt that the management of large syntax graphs as they occur in textual documents, like for instance programming languages, can be achieved with a relational database with an acceptable performance. Linton [36] reports a time of about 200 minutes required for computing an external representation of a programming language syntax tree for a 1,000 line document that was stored in Ingres. Though our own investigations [19] revealed that since 1984 relational database technology, operating system and hardware performance have improved considerably, the performance will still be too slow if syntax graphs are stored in third normal form tables.

7. Summary

Using the case study of constructing the British Airways SEE, we have explored the degree to which software engineering environments can be enhanced on the basis of object database systems. We have discussed a number of advanced requirements, namely inter-document consistency constraints, version and configuration management and cooperative work, that are not addressed by environments that are in the market. We have discussed how these requirements can be addressed on the basis of documents that are represented as ASGs. We have shown how these ASGs can be defined in a dedicated specification language and how an object database schema can be derived from these specifications. The schema then enables documents to be managed by object databases in an ASG representation. We have sketched the extension of a particular object database, namely the O_2 system, with facilities for version management of collections of objects and object-level concurrency control. We have also shown how these extensions are exploited to implement version management of documents and cooperative work. The evaluation of the BA SEE has shown the advantage of specifying the environment at a level of abstrac-

tion higher than that allowed by manual construction. The environment generated from this specification has a performance that is in general acceptable.

When reviewing the related work, we observed that other systems lack support for at least one of our central requirements. We assume that an implementation was too difficult to achieve without the powerful basic mechanisms that object databases provide. The file systems used for document storage purposes lack support for secondary storage management, error recovery and the fine-grained concurrency control required to support cooperative work. In our experience the implementation of an efficient and reliable concurrency control mechanism that could be used for fine-grained concurrency control is so difficult to achieve that it can be hardly expected in any home-grown database systems, be it an academic prototype or a system developed in-house by an SEE vendor. In addition, there are dependencies between version management and concurrency control that preclude the implementation of a powerful version management mechanism for subgraphs of an ASG on top of, say, a relational database system. This leads to the conclusion that the way ahead is to exploit the powerful basic mechanisms now emerging in object database products. The case study outlined in this article has given evidence that object databases can be used for improving the functionality of software engineering environments considerably.

Acknowledgments

We are grateful to Roberto Zicari, who insisted on constructing the BA SEE as a case study in the GOODSTEP project. A particular thanks goes to Werner Beckmann, Jörg Brunsmann, Ralph Mertingk and Matthias Kurth, who implemented most of the BA SEE. We also thank Claude Delobel, Sabine Sachweh, Wilhelm Schäfer and Jim Welsh for their suggestions on the version manager and Sabine Habert for her effort in getting the object-level concurrency control in place.

References

[1] T. Andrews, C. Harris, and K. Sinkel. Ontos: A Persistent Database for C++. In R. Gupta and E. Horowitz, editors, *Object-Oriented Databases with Applications to CASE, Networks, and VLSI CAD*, pages 387–406. Prentice-Hall, 1991.

[2] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The Object-Oriented Database System Manifesto. In W. Kim, J.-M. Nicholas, and S. Nishio, editors, *Proc. of the 1st Int. Conf. on Deductive and Object-Oriented Databases, Kyoto, Japan*, pages 223–240. North-Holland, 1990.

[3] F. Bancilhon, C. Delobel, and P. Kanellakis. *Building an Object-Oriented Database System: the Story of O_2*. Morgan Kaufmann, 1992.

[4] S. Bandinelli, L. Baresi, A. Fuggetta, and L. Lavazza. Experiences in the Implementation of a Process-centered Software Engineering Environment Using Object-Oriented Tech-

nology. *Theory and Practice of Object Systems*, 1(2):115–131, 1995.

[5] S. Bandinelli, A. Fuggetta, and S. Grigolli. Process Modeling-in-the-large with SLANG. In *Proc. of the 2nd Int. Conf. on the Software Process, Berlin, Germany*, pages 75–83. IEEE Computer Society Press, 1993.

[6] N. S. Barghouti, W. Emmerich, W. Schäfer, and A. H. Skarra. Information Management in Process-Centered Software Engineering Enviroments. In A. Fuggetta and A. Wolf, editors, *Software Process*, number 4 in Trends in Software, chapter 3, pages 53–87. Wiley, 1996.

[7] N. S. Barghouti and G. E. Kaiser. Modeling Concurrency in Rule-Based Development Environments. *IEEE Expert*, pages 15–27, December 1990.

[8] V. Benzaken, C. Delobel, and G. Harrus. Clustering Strategies in O_2: An Overview. In *[3]*, pages 385–410. Morgan Kaufman, 1992.

[9] B. W. Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, pages 61–72, May 1988.

[10] G. Booch. *Object Oriented Design with Applications*. Benjamin/Cummings, 1991.

[11] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: The System. *ACM SIGSOFT Software Engineering Notes*, 13(5):14–24, 1988. Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Boston, Mass.

[12] R. Cattell, editor. *The Object Database Standard: ODMG-93*. Morgan Kaufman, 1993.

[13] C. Collet, T. Coupaye, and T. Svensen. NAOS Efficient and modular reactive capabilities in an Object-Oriented Database System. In *Proc. of the 20th Int. Conf. on Very Large Databases, Santiago, Chile*, 1994.

[14] G. Copeland and D. Maier. Making Smalltalk a Database System. *ACM SIGMOD Record*, 14(2):316–325, 1984. Proc. of the ACM SIGMOD 1984 Int. Conf. on the Management of Data, Boston, MA.

[15] C. J. Date. *Introduction to Database Systems, Vol. 1*. Addison Wesley, 1986.

[16] W. Deiters and V. Gruhn. Managing Software Processes in MELMAC. *ACM SIGSOFT Software Engineering Notes*, 15(6):193–205, 1990. Proc. of the 4th ACM SIGSOFT Symposium on Software Development Environments, Irvine, Cal.

[17] Department of Defense. Reference Manual for the Ada Programming Language. Technical Report ANSI/MIL-STD-1815A, United States Dept. of Defense, 1983.

[18] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison Wesley, 1990.

[19] W. Emmerich. *Tool Construction for process-centred Software Development Environments based on Object Database Systems*. PhD thesis, University of Paderborn, Germany, 1995.

[20] W. Emmerich. Tool Specification with GTSL. In *Proc. of the 8th Int. Workshop on Software Specification and Design, Schloss Velen, Germany*, pages 26–35. IEEE Computer Society Press, 1996.

[21] W. Emmerich, S. Bandinelli, L. Lavazza, and J. Arlow. Fine grained Process Modelling: An Experiment at British Airways. In *Proc. of the 4th Int. Conf. on the Software Process, Brighton, United Kingdom*, pages 2–12. IEEE Computer Society Press, 1996.

[22] W. Emmerich, W. Schäfer, and J. Welsh. Databases for Software Engineering Environments — The Goal has not yet been attained. In I. Sommerville and M. Paul, editors, *Software Engineering ESEC '93 — Proc. of the 4th European*

Software Engineering Conference, Garmisch-Partenkirchen, Germany, volume 717 of *Lecture Notes in Computer Science*, pages 145–162. Springer, 1993.

[23] G. Engels, C. Lewerentz, M. Nagl, W. Schäfer, and A. Schürr. Building Integrated Software Development Environments — Part 1: Tool Specification. *ACM Transactions on Software Engineering and Methodology*, 1(2):135–167, 1992.

[24] R. Fehling and W. Schäfer. OPUS: Konzept und Werkzeug für die verteilte, modulare Softwareentwicklung. In U. Kelter and W. Lippe, editors, *Software-Technik Trends — Proc. Softwaretechnik '93*, pages 73–80, November 1993.

[25] F. Ferrandina, T. Meyer, and R. Zicari. Implementing Lazy Database Updates for an Object Database System. In *Proc. of the 20th Int. Conference on Very Large Databases, Santiago, Chile*, pages 261–272, 1994.

[26] D. Garlan. *Views for Tools in Integrated Environments*. PhD thesis, Carnegie Mellon University, 1987.

[27] K. E. Gorlen. An Object/Oriented Class Library for C++ Programs. *Software – Practice and Experience*, 17(12):181–207, 1987.

[28] J. N. Gray. Notes on Database Operating Systems. In R. Bayer, R. Graham, and G. Seegmüller, editors, *Operating systems – An advanced course*, volume 60 of *Lecture Notes in Computer Science*, chapter 3.F., pages 393–481. Springer, 1978.

[29] R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane, and W. M. Waite. Eli: A Complete, Flexible Compiler Construction System. *Communications of the ACM*, 35(2):121–131, 1992.

[30] A. N. Habermann and D. Notkin. Gandalf: Software Development Environments. *IEEE Transactions on Software Engineering*, 12(12):1117–1127, 1986.

[31] Intersolve, 3200 Tower Oaks Blvd, Rockville, Maryland. *PVCS Version 4.0 Version Manager Users Reference Guide*, 1991.

[32] U. Kastens and W. M. Waite. An abstract data type for name analysis. *Acta Informatica*, 28:539–558, 1991.

[33] D. Kiong and J. Welsh. Incremental Semantic Evaluation in Language-based Editors. *Software – Practice and Experience*, 22(2):111–135, 1992.

[34] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore Database System. *Communications of the ACM*, 34(10):51–63, 1991.

[35] C. Lewerentz and A. Schürr. GRAS, a management system for graph-like documents. In *Proc. of the 3rd Int. Conf. on Data and Knowledge Bases*, pages 19–31. Morgan Kaufmann, 1988.

[36] M. A. Linton. Implementing Relational Views of Programs. *ACM SIGSOFT Software Engineering Notes*, 9(3):132–140,

1984. Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Penn.

[37] B. Peuschel, W. Schäfer, and S. Wolf. A Knowledge-based Software Development Environment Supporting Cooperative Work. *International Journal for Software Engineering and Knowledge Engineering*, 2(1):79–106, 1992.

[38] S. P. Reiss. Interacting with the FIELD environment. *Software – Practice and Experience*, 20(S1):S1/89–S1/115, 1990.

[39] T. W. Reps and T. Teitelbaum. *The Synthesizer Generator – a system for constructing language based editors*. Springer, 1988.

[40] M. J. Rochkind. The Source Code Control System. *IEEE Transactions on Software Engineering*, 1(4):364–370, 1975.

[41] W. W. Royce. Managing the Development of Large Software Systems. In *Proc. WESCON*, 1970.

[42] C. Santos, S. Abiteboul, and C. Delobel. Virtual Schemas and Bases. In M. Jarke, J. Bubenko, and K. Jefferey, editors, *Proc. of the 4th Int. Conf. on Extending Database Technology, Cambridge, UK*, volume 779 of *Lecture Notes in Computer Science*, pages 81–94. Springer, 1994.

[43] Softool. *CCC: Change and Configuration Control Environment. A Functional Overview*, 1987.

[44] J. M. Spivey. *The Z Notation - A Reference Manual*. Prentice Hall, 1989.

[45] W. F. Tichy. RCS – A System for Version Control. *Software – Practice and Experience*, 15(7):637–654, 1985.

[46] A. I. Wassermann and P. A. Pircher. A Graphical, Extensible Integrated Environment for Software Development. *ACM SIGPLAN Notices*, 22(1):131–142, 1987. Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Palo Alto, Cal.

[47] J. Welsh, B. Broom, and D. Kiong. A Design Rational for a Language-based Editor. *Software – Practice and Experience*, 21(9):923–948, 1991.

[48] B. Westfechtel. Revision Control in an Integrated Software Development Environment. *ACM SIGSOFT Software Engineering Notes*, 17(7):96–105, 1989.

[49] A. L. Wolf, L. A. Clarke, and J. C. Wileden. The AdaPIC Tool Set: Supporting Interface Control and Analysis Throughout the Software Development Process. *IEEE Transactions on Software Engineering*, 15(3):250–263, 1989.

[50] A. Zündorf. *PROgrammierte GRaphErsetzungsSysteme – Spezifikation, Implementierung und Anwendung einer integrierten Entwicklungsumgebung*. PhD thesis, University of Aachen, 1996.