Specification of Tools with GTSL*

Wolfgang Emmerich
University of Dortmund
Informatik 10
D-44221 Dortmund
Germany
emmerich@ls10.informatik.uni-dortmund.de

Abstract

The definition of software development methods encompasses the definition of syntax and static semantics of specification languages. These languages determine documents to be produced during the application of a method. Developers demand language-based tools that provide document production support, check syntax and static semantics of documents and thus implement methods. A number of methods are used in the different tasks of software construction and a need arises for their integration. This method integration must determine inter-document consistency constraints between documents produced in the various tasks. The various tools that are used during the process must, therefore, be integrated in a way that they implement the required method integration. Unfortunately, the particular mix of methods that is used in one software process need not be appropriate for another. Method integration must, therefore, become part of process modelling and tool integration must be adjusted for each different process. The focus of this paper is on the specification of integrated tools. We outline the main concepts of a dedicated, object-oriented tool specification language, namely GTSL. The language was defined, implemented and evaluated in the GOODSTEP project. Part of the evaluation was the construction of a set of integrated tools for C++ class library maintenance for British Airways, an industrial project partner.

1 Introduction

A software process that develops and maintains a software system consists of a number of different *tasks*. Examples are *requirements analysis* tasks where requirements of future customers of a software system are elicited or *architectural design* tasks where the different components of software systems and relationships among them are identified. The suggestion of the Waterfall model [Roy70] that these tasks be performed in mutual exclusion has been proved infeasible [Boe88]. Instead the tasks are often carried out in an intertwined manner.

Tasks are performed using particular *methods*, like *structured analysis* [dM78] for requirements analysis or object-oriented methods (for instance the Booch methodology [Boo91] or the Object Modelling Technique [RBP+91]) for architectural design. *Method definitions* have to determine formal graphical or textual languages. Examples are data flow diagrams or class hierarchies.

*This work has been partly funded by the CEC as part of ESPRIT-III project 6115 (GOODSTEP).

These languages then determine *document types* and the purpose of each task of a software process is to create, analyse and maintain *documents* of the types identified. Hence, the definition of a method encompasses the precise definition of *document types*. Document types are defined in terms of syntax and static semantics of the underlying specification languages.

A particular mix of methods that is appropriate in one process need not be appropriate for another. A process developing a real-time application, for instance, should use a requirements definition language that can express response time constraints, but such a language might be unnecessarily complicated for customers of a banking application where response time constraints need not be expressed. Likewise, the language used for module interface design will have to be directed towards the programming language used in the implementation task. This means that it is impossible to find **the** mix of methods that could be used in arbitrary software processes. One of the main goals during *software process modelling* is, therefore, to identify those methods and document types that are most appropriate for the tasks to be carried out during the process being modelled.

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. A need for *method integration* arises whose aim is to define the consistency constraints that documents must obey. An important factor for the quality of a software system is then whether these constraints have been defined properly and are respected by the documents produced during the process. Due to the fact that a method mix is process-specific, method integration must become part of process modelling and deserves appropriate attention.

Methods are implemented by tools that software developers can use to apply the method. To implement method integration then requires tools to be integrated. The main contribution of this paper is the presentation of the *GOODSTEP*[1] *tool specification language (GTSL)*, an object-oriented language dedicated to the specification of integrated tools. A compiler for this language has been implemented and enables tools to be constructed and to be adapted to particular processes efficiently. The rest of this paper will be structured as follows. In the next section, we discuss the need for method and tool specification in more detail. Section 3 suggests a representation for documents as a basis for the specification of tools and document types and relates it to the literature. In Section 4 we present the main concepts of GTSL. Section 5 describes the main results of an evaluation that has used GTSL to construct tools in order to implement methods for the development and maintenance of class libraries within British Airways, an industrial partner of the GOODSTEP project. We conclude the paper in Section 6 with work that remains to be done.

2 Method and Tool Customisation Required

As an illustrating example that we will use throughout this paper consider Figure 1. It displays four different documents of four different types. Starting from the bottom left, there are in clock-wise order an entity relationship diagram, an architectural definition that identifies different types of modules as components of a software system[2], a module interface specification that identifies exported types and operations as well as an import interface, and a module

[1]The purpose of GOODSTEP is to enhance a general object database for software engineering processes.
[2]The detailed notion is of no concern here and we refer to [ES94].

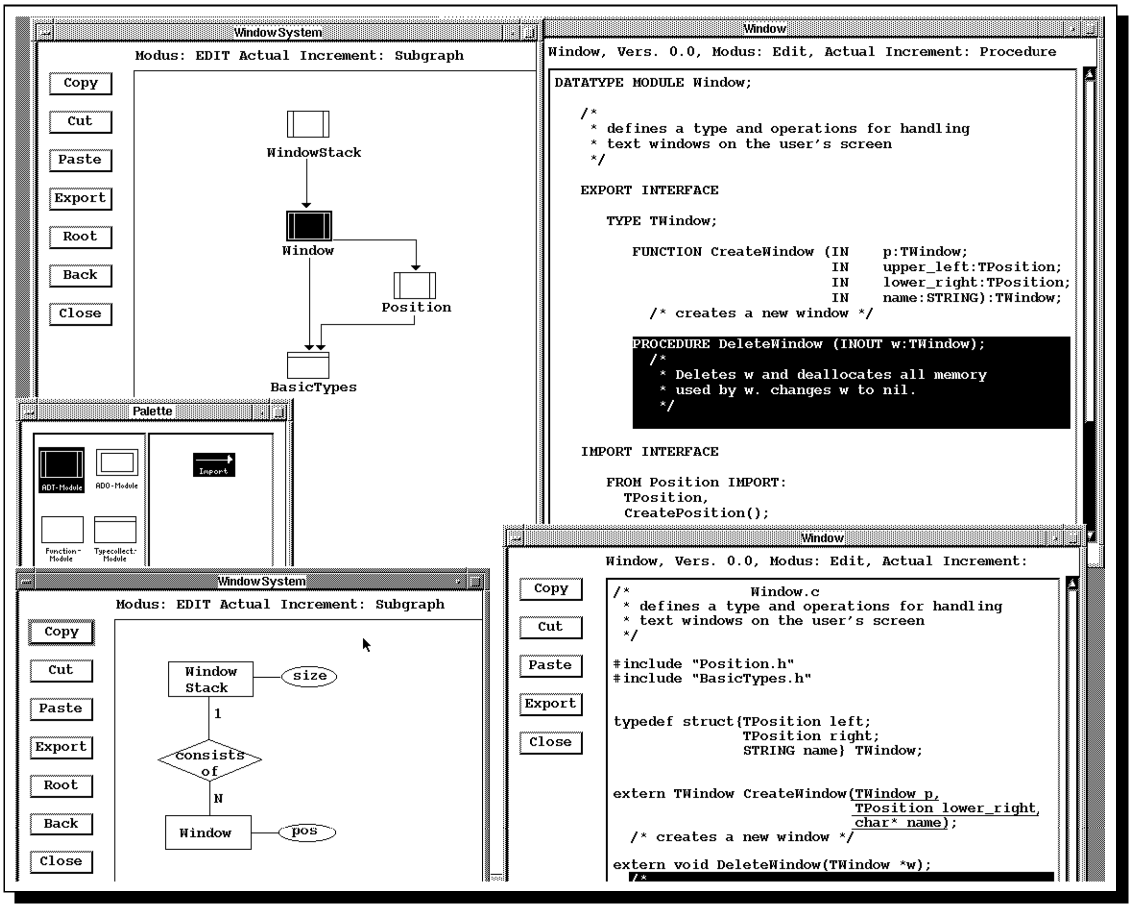

Figure 1: Inter-Document Consistency Constraints

implementation that implements the exported types and operations of a module in the C programming language. The integration of the underlying methods requires a number of inter-document consistency constraints to be defined between the respective document types. Entities of the entity relationship diagram, for instance, must be refined in terms of abstract data type modules in the architecture diagram. Modules in these diagrams, in turn, must be refined by a module interface definition, that defines the export and import interface in detail. Each arrow of the architecture diagram should appear as an entry in the import interface of the module interface definition. Operations and types that have been identified in the export interface must be properly implemented in the C document. Therefore, parameter lists in the interface design and C should match as well as the result types. Moreover, import interfaces are refined by pre-processor `#include` statements. Vice versa, there should be no such statements when the design does not include the respective entry in the import interface, otherwise there would be dependencies among source code components that are not properly reflected in the design.

The need arises to assist software developers in the production of documents that meet inter-document consistency constraints like the ones outlined above, and thus to implement the methods and their integration. Users[3], therefore, require a *tool* for each document type. Such a tool should then support the methods and offer commands to edit multiple documents of that type. It should be supportive in achieving syntactic and static semantic correctness of

[3]The software developers that use tools are referred to as *users* hereafter.

documents, browsing to semantically related documents and most important, it must check for inter-document consistency. This requires that tools be aware of the syntactic structure of documents. We refer to the syntactic units of documents as *increments* hereafter. The example of Figure 1 displays the user interfaces of tools contained in the *Groupie* environment [ES94]. They are used to edit, analyse and check documents of the types identified above.

To implement method integration, tools have to check for inter-document consistency constraint violations. Different strategies can be considered how a tool should react to a constraint violation. It might handle a violation in a lazy way and only visualise an inconsistency to the user when it has been introduced. This visualisation might be achieved by the use of colours or by underlining. In the example of Figure 1, a parameter list in the C implementation is underlined because it does not match the parameter list determined in the interface design. Detailed error messages should be provided on demand in order not to overload the document representation. A tool might also follow an eager approach and reject the execution of commands that would violate an inter-document consistency constraint. A tool might even automatically correct erroneous increments. Upon a change of one increment, it can, for instance, automatically modify related increments in other documents in such a way that consistency is retained. We refer to these automatic modifications of related increments as *change propagations*.

The implementation of method integration might depend on the process state. A constraint violation that is tolerable in an early stage of a software process might become intolerable when the process reaches a certain deadline where documents must become consistent. As an example, consider again the above scenario. An import from a non-existent module in a module interface definition is quite tolerable during the design. If the tool did not tolerate such errors bottom-up design would be enforced, which is not always appropriate. The constraint may, therefore, be violated temporarily. If implementation of the module has started, however, the constraint should not be violated because the implementation of the module might depend on a type that might never be implemented and if that is detected too late a significant amount of effort is wasted.

Most software processes are conducted by multiple rather than single developers. This implies that we also have to consider the concurrent use of tools by multiple users. Different *versions* of documents must be managed to facilitate independent document development [SS95]. The methods defined in terms of document types, therefore, also have to identify the granularity for version management. Versions are then used to allow users to edit documents in *isolation* for a certain period of time. However, due to inter-document consistency constraints, the development cannot be performed in complete isolation. At some point in time, the documents produced by one developer must become consistent with documents produced by other developers. Users must then share their document versions. Consider in the above example, that a requirements engineer uses the entity relationship tool to define the information model of a software system, while a system architect is in charge of the architectural design of the system. Their document versions should become consistent with each other before implementation begins, otherwise significant effort might be wasted during implementation if, for instance, wrong names are used or it turns out that an implemented module is obsolete. They, therefore, have to edit the same versions of the entity relationship and the architecture diagram concurrently.

To reach a state of consistency, users then want to see the impact of concurrent document updates as soon as possible. Tight cooperation then requires updates to a document version to be done in such a way that all tools concurrently displaying a document version are informed of the update as soon as possible. They should then redisplay the document version in order to reflect the update as well. In the above example of inconsistent parameter lists, a designer might remove the inconsistency by deleting the additional parameter. If a programmer is concurrently accessing the implementation document that corresponds to the interface, he or she should see, as soon as possible, that the inconsistency has been resolved and requires no further attention. The shared and cooperative updates of document versions must, therefore, not be disabled by exclusive locking of complete documents by long transactions but transactions must be short and locking must be done with a more fine-grained granularity.

Different methods and languages are used in a software process, depending on the process model. Companies often do not use methods and languages as defined originally, but have their own guidelines to use only a particular subset. Hence, there is a need for *specification languages* that are devoted to the definition and customisation of methods. As users require tools to effectively apply methods, the specification languages must define tools in a way that they implement the respective methods. Method integration depends on the process-specific mix of methods. Therefore there is a need to define, or at least to customise, method integration for each different process model. Since method integration must be implemented in terms of tools that check inter-document consistency constraints, the specification languages must be capable of defining the required tool integration. Our concern is thus to ease this tool construction and customisation as far as possible. We, therefore, focus on generating integrated tools that implement methods and their integration from appropriate high-level specifications.

3 Document Representation

Before we can identify concepts of a higher-level specification language, we will have to understand how documents should be represented. During this discussion we compare our considerations to related work. The common internal representation for documents manipulated by tools is an *abstract syntax tree* of some form [RT81, DGKLM84, HN86]. Nodes in the abstract syntax tree often have additional attributes whose values represent semantic information such as references to a string table, symbol tables or type information. Operations, like insertion of a new parameter list place holder can easily be implemented as operations on this abstract syntax tree. It can be implemented as subtree replacement. After free textual input, the abstract syntax tree can be established with parsing techniques well-known from compiler construction [ASU86].

Static semantic checking of a document that is represented as an abstract syntax tree can be done by *attribute evaluations* along parent/child paths in the document's attributed abstract syntax tree [Knu68, RT84]. The evaluation paths are computed at tool construction time based on attribute dependencies. If inter-document consistency checks between different documents are implemented by attribute evaluations, all inter-document consistency constraints must be checked at an artificial root node, which has sub-trees for each document. With respect to concurrent tool execution many concurrency control conflicts arise at these root nodes and decrease efficiency. Therefore, techniques based on the introduction of additional, non-syntactic paths for more direct attribute propagation have been developed [JF82, Nag85, Hoo87]. They generalise the concept of abstract syntax trees to *abstract syntax graphs*. Such non-syntactic paths implement *semantic relationships* that connect syntactically disjoint parts of possibly different documents even of different types. They can be used for consistency checking, change-propagation when the document is changed and even for implementing static semantic analysis and browsing facilities. To handle these semantic relationships in a consistent way, the obvious strategy is to view the set of documents making up a project as a single *project-wide abstract syntax graph*.

We note that this generalisation to a single project-wide graph does not necessarily undermine the concept of a document as a distinguishable representation component. If we distinguish between *aggregation edges* in the graph, which implement syntactic relationships, and *reference edges*, which arise from semantic relationships, then a document of the project is a subgraph whose node-set is the closure of nodes reachable by aggregation edges from a document node (i.e., a node not itself reachable in this way), together with all edges internal to the set[4]. The edges not included in this subgraph are then necessarily the inter-document relationships inherent in the project. Nodes that cannot have outgoing aggregation edges are called *terminal nodes*, for their origin lies in terminal symbols of the underlying grammar. Those nodes that may have outgoing aggregation edges shall be called *non-terminal nodes* accordingly.

As an example, consider Figure 2. It outlines how the different abstract syntax trees representing documents of Figure 1 are integrated to a project-wide abstract syntax graph. The refinement of entities defined in the entity relationship diagram in terms of modules of the architecture is reflected by inter-document reference edges labelled `ToArch`. Likewise, the refinement of modules of the architecture definition in terms of module interface documents is stored by means of reference edges labelled with `ToDesign`. Intra-type reference edges labelled `DefinedIn` represent the use/declare relationship between type increments of a module interface document. The parameter type of function `CreateWindow` with the attribute `STRING`, for instance, has an outgoing reference edge to the node where it is declared, that is to the `TypeImport` node with attribute `STRING`. This node represents an import that is itself connected via an inter-document reference edge labelled `ImpFrom` to another node contained in the subgraph of module `BasicTypes` where the type is exported.

Graph grammars have been suggested in [ELS87, Sch91, ELN+92] to specify the structure of such abstract syntax graphs. Productions of the grammar can then be considered as available operations to modify abstract syntax graphs. Graph grammars, however, do not appropriately specify concurrency constraints, lexical syntax, external document representations and dialogues between users and tools during command execution. Moreover, graph grammars do not impose a particular structuring paradigm and specifications of graphs that occur in practice tend to become so complex that they cannot be managed appropriately.

4 The Tool Specification Language GTSL

On the basis of the above concepts, we can now focus on the question how document types and tools are defined appropriately. We propose GTSL for that purpose. The language allows tool builders to define *static* and *dynamic properties* of syntax graphs as well as mappings between syntax graphs and *external document representations*. The static properties that are to be defined are the various node types, their attributes and the edges that may start at or lead to nodes. Dynamic properties are, firstly, the available tool commands and their definition on the basis of syntax graph access and modification operations and, secondly, the dependencies between attribute values and reference edges that define static semantics and inter-document

[4]What we call a subgraph here, is comparable to the notion of a *composite entity* in PACT VMCS (c.f. [Tho89]).

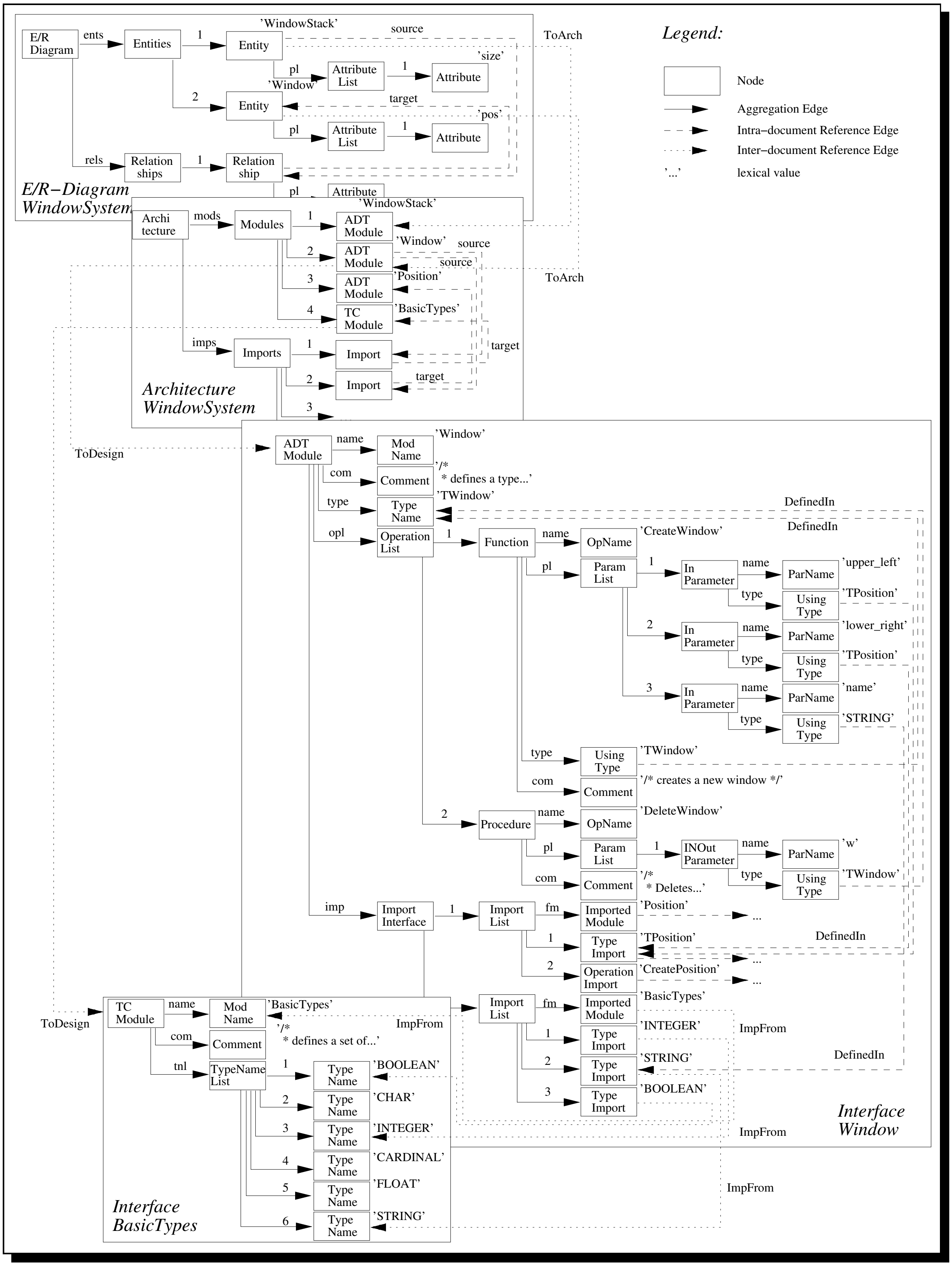

Figure 2: A Project-wide Abstract Syntax Graph

consistency constraints. The mappings to external document representations must define the appearance of abstract syntax graphs at the user interface of tools or in printed tool output.

For the definition of a project-wide abstract syntax graph, as many node types must be defined as there are productions in the underlying languages. The specification of document types and tools, therefore, becomes rather complex for methods and languages that occur in software engineering practice. The specification language must incorporate structuring facilities to keep this complexity manageable. The overall specification of a project-wide abstract syntax graph should, therefore, be decomposed into the specifications of the various subgraphs that represent different document types and each of these subgraph specifications should be decomposed into specifications for the node types that occur in the subgraph. GTSL supports this decomposition in an object-oriented way. Tools are specified in terms of *configurations* of *increment classes*. Increment classes determine node types of the project-wide syntax graphs and configurations determine the increment classes that belong to a tool.

The complexity of specifying a tool is significantly reduced if tool builders can reuse already existing tool specification components. GTSL, therefore, allows the tool builder to identify similarities among different increment classes and to specify the common structure and behaviour of increments in one class and reuse it in similar classes. The object-oriented paradigm is exploited and similarities are expressed by inheritance. Definitions inherited from super classes can be customised by redefining them. Reuse is then further supported since GTSL comes with a library of predefined classes defining, for instance version management, common tool commands, symbol tables or scoping rules.

Due to the heterogeneity of the different static and behavioural concerns, it will hardly be possible to find a unique formalism that will be appropriate for their specification. Instead, we will separate the different concerns and offer the most appropriate formalism for each of them. Following the principle of information hiding [Par72], the definition of a class will be divided into a public *class interface* and a private *class specification*. The class interfaces and specifications are, in turn, structured into different sections that offer different paradigms to specify the various concerns. We integrate these different formalisms into a domain-specific *multi-paradigm language* that uses rule-based, object-oriented and imperative concepts.

The node types in an abstract syntax graph definition play different roles. If we ask a tool builder to make these roles explicit, we will be able to define a domain-specific type system for GTSL that enables a number of specification errors to be detected. *Terminal classes*, which define leave nodes of the abstract syntax tree, must not have commands to expand child increments, whereas *non-terminal classes*, which define inner syntax tree nodes, require these commands. Non-terminal classes must also specify the unparsing scheme that defines the external representation of their instances. *Abstract classes* define common properties of classes that are inherited by its subclasses. We call instances of non-terminal classes *non-terminal increments* and instances of terminal classes *terminal increments*. We refer to them as *increments*, if their position in the syntax tree is not important. Besides increment classes, we additionally support *non-syntactic* classes that will be used for the declaration of non-atomic attribute types, such as error lists or symbol tables. Instances of these classes are referred to as *attributes*. If the distinction between attributes and increments is not important, we will denote instances of classes as *objects*.

4.1 Specification of Static Properties

Abstract Syntax: Aggregation edges that start from nodes of a type are defined in the increment class interfaces within an *abstract syntax section*. The abstract syntax section is available for abstract and non-terminal increment classes. It is not defined for terminal increment classes, because these must not have outgoing aggregation edges by definition. If a child is defined in an abstract class it is inherited by all subclasses. Children are specified in the abstract syntax section with the name and a formal type that restricts child increments to instances of the formal type or subtypes thereof. Multi-valued aggregation edges are defined by the `LIST` type constructor. Below there are several examples of abstract syntax sections that define node types used in the graph in Figure 2.

```
ABSTRACT INCREMENT INTERFACE Module;
  INHERIT DocumentVersion, ScopingBlock;
  ...
  ABSTRACT SYNTAX
    name:ModName;
  END ABSTRACT SYNTAX;
  ...

NONTERMINAL INCREMENT INTERFACE ADTModule;
  INHERIT Module, Commentable;
  ...
  ABSTRACT SYNTAX
    type:TypeName;
    opl:OperationList;
    imp:ImportInterface;
  END ABSTRACT SYNTAX;
  ...
```

```
ABSTRACT INCREMENT INTERFACE Commentable;
  INHERIT Increment;
  ...
  ABSTRACT SYNTAX
    com:Comment;
  END ABSTRACT SYNTAX;
  ...

NONTERMINAL INCREMENT INTERFACE OperationList;
   INHERIT Increment
  ...
  ABSTRACT SYNTAX
    ol:LIST OF Operation;
  END ABSTRACT SYNTAX;
  ...
```

The distinction between different types of classes enables us to exclude a number of potential specification errors. It does not make sense, for instance, to have a terminal increment class that inherits from an abstract class, which, in turn, defines abstract syntax children. In that case the terminal class would inherit these children and no longer be terminal. In addition, it is unreasonable to have attributes or atomic types as syntax children. The type system of GTSL, therefore, excludes these situations. Without the domain-specific distinction between different kinds of classes, such as if we used a conventional object-oriented programming language, we would not have been able to detect these specification errors.

Attributes: Node attributes are declared within the attribute section of increment classes. An attribute definition declares a *name* and a *type* of an attribute. Non-syntactic classes can also be used to impose a particular behaviour on attribute types. We do not address non-syntactic classes any further here. They provide the expressive power of an object-oriented language including multiple inheritance, construction of types and encapsulation with methods. As an example, consider the following example from the Groupie interface editor definition. It defines an attribute `DefinedNames` whose type is of class `SymbolTable`. It is used to maintain associations between names and increments.

```
ABSTRACT INCREMENT INTERFACE ScopingBlock;
  ...
  ATTRIBUTES
    DefinedNames:SymbolTable;
  END ATTRIBUTES;
  ...
```

Semantic Relationships: Reference edges are defined as pairs of unidirectional *links* in the *semantic relationship* sections of the two increment classes that are connected by the edge[5]. The *explicit link* denotes the direction from the source to the target increment class. The *implicit link* denotes the reverse direction.

Relationships are created and deleted during static semantics and inter-document consistency checks. Creation of a relationship is specified by assigning an expression that denotes an increment to an explicit link. Such an assignment first deletes the existing relationship, if any. Then the increment denoted by the expression is assigned to the explicit link. Finally the implicit link is established by including the source increment in the set that stores the implicit link. A relationship is deleted by assigning the undefined value `NIL` to the explicit link of the relationship. Thus creation and deletion of relationships is only controlled by the target increment class of a relationship or its subclasses. This further contributes to the enforcement of structured specifications. As an example, consider the relationships `DefinedIn/UsedBy` between `UsingType` and `TypeDecl` and `ImpFrom/ExpTo` between classes `TypeImport` and `TypeName` that are taken from the Groupie interface editor specification.

```
TERMINAL INCREMENT INTERFACE UsingType;  ABSTRACT INCREMENT INTERFACE TypeDecl;
 INHERIT UsingName;                       INHERIT DefiningName;
 SEMANTIC RELATIONSHIPS                   SEMANTIC RELATIONSHIPS
  DefinedIn: TypeDecl                      IMPLICIT UsedBy:SET OF UsingType.DefinedIn;
 END SEMANTIC RELATIONSHIPS;              END SEMANTIC RELATIONSHIPS;
 ...                                      ...
TERMINAL INCREMENT INTERFACE TypeImport; TERMINAL INCREMENT INTERFACE TypeName;
 INHERIT TypeDecl;                        INHERIT TypeDecl;
 SEMANTIC RELATIONSHIPS                   SEMANTIC RELATIONSHIPS
  ImpFrom:TypeName;                        IMPLICIT ExpTo: SET OF TypeImport.ImpFrom;
 END SEMANTIC RELATIONSHIPS;              END SEMANTIC RELATIONSHIPS;
 ...                                      ...
```

Increment class `UsingType` defines an explicit link `DefinedIn` to an abstract increment `TypeDecl`. Using that link, a `UsingType` increment can refer to the increment where its type is declared. Due to polymorphism, this can be a type import or an exported type name. The relationship is established by assigning an instance of classes `TypeName` or `TypeImport`, which are subclasses of `TypeDecl`, to the link. After that, `DefinedIn` may be used to navigate to the corresponding declaration increment. The implicit link `UsedBy` in class `TypeDecl` contains an implicit reference to all the increments that refer to one particular type. That link is exploited for the definition of change propagations or in a browsing command that visits all increments that use a particular type. Similarly, the link pair `ImpFrom/ExpTo` models the import/export relationship between imported and exported types.

4.2 Specification of Dynamic Properties

Semantic Rules: Attributes and semantic relationships are concepts that can be used for defining data structures for static semantics and inter-document consistency constraints. Changes of attribute values and the creation or deletion of semantic relationships will be defined in tool command definitions. These changes, however, usually require a number of follow-on activities in order to check static semantic constraints for related increments.

[5]The terminology follows the concepts for relationships that have been introduced in the PCTE data model [GMT87].

If tool builders have to use imperative concepts to define static semantics and inter-document consistency constraints they would have to find valid execution orders to perform the required follow-on actions for all potential attribute and semantic relationship changes. We strongly consider this to be at the wrong level of abstraction. Tool builders require instead a declarative concept for defining the correctness of the various static semantic and inter-document consistency constraints. This concept should, in particular, relieve them from worrying about the order in which evaluations are performed. The new concept should also support our structuring paradigm and be defined in terms of increment classes. In addition, the concept must enable the efficient evaluation of static semantic constraints to be carried out as this has to be done on-line, i.e. during the execution of user commands. We introduce *semantic rules* for that purpose.

Each semantic rule consists of a list of statements called *action* that is bound to a *condition*. The condition is specified after the `ON` clause and the action is defined between `ACTION` and `END ACTION` keywords. Temporal predicates may be used to specify conditions, namely `CHANGED` and `DELETED`. A `CHANGED` predicate becomes `TRUE` if its argument has been created or changed since the last execution of the semantic rule. The `DELETED` expression becomes `TRUE` if its argument is about to be removed. Arguments of a `CHANGED` or `DELETED` expression may be attributes or semantic relationships of any other increments. Path expressions are used to determine attributes or semantic relationships of remote increments. A name of an attribute may only occur as the last name in a path expression. Compound conditions can be built by using the `OR` operator. An `EXISTS` operator is used in the usual sense of first-order logic to specify that the rule has to be executed as soon as some other condition holds for an element in a multi-valued syntax child or a multi-valued semantic relationship.

As an example we now consider a solution to a problem that occurs during static semantics specification, namely the *name analysis problem* [KW91]. We solve it with three abstract classes, i.e. `ScopingBlock`, `DefiningName` and `UsingName`. The classes are independent of a particular target language and can thus be reused to define name analysis in multiple tools. `ScopingBlock` serves as super class for increment classes that start a new block. `DefiningName` serves as a super class for classes whose increments contribute to the declaration of new names. Finally, `UsingName` serves as a super class for all applied occurrences of names. The attribute `DefinedNames` in class `ScopingBlock` is used to maintain associations between names and references to increments where the respective names are declared. We then have to define that an association is included for those and only those increments that declare names. Hence associations are entered into the table when they are created, the table is updated when the increment name is changed and associations are deleted when the declaration is deleted. This is defined in the semantic rules in Figure 3.

To keep the solution language-independent, we define an auxiliary semantic relationship `Block/IncludedNames` that connects each declaring increment with its block. We exploit the implicit link `IncludedNames` of this relationship in the semantic rules above. The first rule fires whenever the lexical value attribute of a declaring name identifier is changed. Then the symbol table is updated to include an association between the new value and the increment declaring the value.

As displayed in Figure 4, the explicit link `Block` is established by method `myScope` whenever a new declaring increment is created. The scoping block is determined as the next increment on the way to the root whose class inherits from class `ScopingBlock`. This may be redefined in a language-specific subclass. The link is exploited in a semantic rule that checks for uniqueness of declarations within their scope. Whenever the symbol table of the scoping block is changed,

```
ABSTRACT INCREMENT INTERFACE ScopingBlock;
 INHERIT Increment;
 ...
 SEMANTIC RELATIONSHIPS
  IncludedNames:IMPLICIT SET OF
                DefiningName.Block
 END SEMANTIC RELATIONSHIPS;

 ATTRIBUTES
  DefinedNames:SymbolTable;
 END ATTRIBUTES;
 ...
END INCREMENT INTERFACE ScopingBlock;
```

```
INCREMENT SPECIFICATION ScopingBlock;
 INITIALIZATION
  DefinedNames := NEW DuplicateSymbolTable;
 END INITIALIZATION;

 SEMANTIC RULES
  ON EXISTS(name:DefiningName IN IncludedNames):
      CHANGED(name.value)
  ACTION
   SELF.DefinedNames.associate(name,name.value);
  END ACTION;

  ON EXISTS(name:DefiningName IN IncludedNames):
     DELETED(name);
  ACTION
   SELF.DefinedNames.deassociate(name,name.value);
 END ACTION;
 END SEMANTIC RULES;
END INCREMENT SPECIFICATION ScopingBlock.
```

Figure 3: Abstract Increment Class `ScopingBlock`

the semantic rule below fires and inserts or deletes an error descriptor into or from an error set. This error set is consulted during the computation of the external representation in order to visualise erroneous increments by underlining. The particular error descriptor to be inserted upon detection of an error is language-specific and is determined by redefinition of a deferred method in the language-specific subclasses.

```
ABSTRACT INCREMENT INTERFACE DefiningName;
 INHERIT Increment;
 ...
 SEMANTIC RELATIONSHIPS
  Block:ScopingBlock;
  UsedBy:IMPLICIT SET OF
           UsingName.DefinedIn
 END SEMANTIC RELATIONSHIPS;

 METHODS
  METHOD myScope():ScopingBlock;
  DEFERRED METHOD ErrorId():ERROR;
 END METHODS;
 ...
END INCREMENT INTERFACE DefiningName;
```

```
INCREMENT SPECIFICATION DefiningName;
 INITIALIZATION
  Block := SELF.myScope();
 END INITIALIZATION;

 SEMANTIC RULES
  ON CHANGED(Block.DefinedNames)
  ACTION
   IF(SELF.Block.DefinedNames.is_dupl(SELF)) THEN
    SELF.Errors.append_error(SELF.ErrorId());
   ELSE
    SELF.Errors.clear_error(SELF.ErrorId());
   ENDIF
  END ACTION;
 END SEMANTIC RULES;
END INCREMENT SPECIFICATION DefiningName.
```

Figure 4: Abstract Increment Class `DefiningName`

In many languages that occur in software engineering practice, applied occurrences of names must match a declaration. This property is defined by a semantic rule in class `UsingName` as depicted in Figure 5. The pre-condition of the rule depends the symbol table of the scoping block determined by link `Block`. The link is established in the same way as above, but this is for reasons of brevity omitted. The rule also depends on the lexical value of the changed name. Whenever any of these are changed, a symbol table lookup is performed in order to store a matching declaration in the local variable `inc`. If `inc` is not `NIL`, a match has been found. We then have to obtain, whether the declaration is of the right type. In our interface editor, for instance the use of operation names as parameter or result types is inhibited. This is again language-specific and determined by the deferred method `DeclClass` that must be redefined in all subclasses. If the increment is of the right class, the semantic relationship `DefinedIn/UsedBy` is established and a language-specific error descriptor, determined by the deferred method `ErrorId`, is deleted from the set of errors. In the other case the semantic relationship is deleted and the error descriptor is inserted into the set.

```
ABSTRACT INCREMENT INTERFACE UsingName;
 INHERIT Increment;
 ...

 SEMANTIC RELATIONSHIPS
  DefinedIn:DefiningName;
  Block:ScopingBlock
 END SEMANTIC RELATIONSHIPS;

 METHODS
  DEFERRED METHOD DeclClass():STRING;
  DEFERRED METHOD ErrorId():ERROR;
  ...
 END METHODS;
END INCREMENT INTERFACE UsingName;
```

```
INCREMENT SPECIFICATION UsingName;
  ...
  SEMANTIC RULES
   ON CHANGED(Block.DefinedNames)
      OR CHANGED(value)
   VAR inc: Increment;
   ACTION
    inc:=Block.DefinedNames.increment_at(value);
    IF inc != NIL THEN
     IF( inc.IS_KIND_OF(SELF.DeclClass())) THEN
      DefinedIn := <DefiningName>inc;
      Errors.clear_error(SELF.ErrorId());
     ELSE
      DefinedIn:=NIL;
      Errors.append_error(SELF.ErrorId());
     ENDIF;
    ELSE
     DefinedIn:=NIL;
     Errors.append_error(SELF.ErrorId());
    ENDIF;
  END ACTION;
 END SEMANTIC RULES;
END INCREMENT SPECIFICATION UsingName.
```

Figure 5: Abstract Increment Class `UsingName`

These classes have been reused in various tools to specify name analysis. In the module interface editor, `Module` is for instance a subclass of `ScopingBlock` and thus implements a new scope. `TypeName` and `TypeImport` are classes that inherit from `DefiningName`, while `UsingType` inherits the semantic rule from `UsingName`. In these subclasses only the deferred methods are redefined in order to determine the language-specific properties.

Interactions: The steps of a software development method are implemented by the commands that the tool offers. The command definition determines the names of commands, pre-conditions for their applicability and the particular dialogues between tool and user, if any. In GTSL commands are defined as *interactions*. 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` 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.

The interaction displayed in Figure 6 is considered to be offered if the selected increment is a type name. It is actually offered if the type has already been expanded. If this is the case and the user has requested a menu, the string `Change Type` will become a menu item. If the user chooses this item, the action is executed and the user will be prompted to edit the type identifier in a line edit window. The default character string in this line edit window is the value of the old type identifier. If the dialogue is completed, the `LINE_EDIT` method returns `TRUE` and the method `scan` is executed. The method implementation is generated from a regular expression that is provided for terminal increment classes. It returns `TRUE` if the identifier is lexically correct, otherwise it returns `FALSE`. If the identifier is correct the two semantic relationships of a type name are exploited to propagate the change to dependent increments such as parameter types or type imports in order to retain consistency. Then the new lexical value is stored in attribute `value`. If the identifier is wrong an appropriate error message is displayed.

Multiple users cannot concurrently execute commands in a totally unrestricted way. This is due to the *lost update* and *inconsistent analysis* problems, known from concurrency control in database systems [Dat86]. As an example of the inconsistent analysis problem, consider the following scenario. A designer uses the above interaction to change the name of an exported type. A concurrently working designer creates an import statement referring to the old type. During that, the included type name is searched in the symbol table `DefinedNames` of the module where the type is being changed. An inconsistent analysis problem occurs if this search is performed after the other tool has done the change propagation and before the association was changed in the table. Then the import statement will not be displayed as inconsistent although the imported type does not exist anymore. The construction of an example for lost updates is straight-forward.

Now we have encountered the dilemma that we cannot lock document versions exclusively while they are being edited without hampering cooperation. On the other hand, we must restrict concurrency to avoid the lost update and inconsistent analysis problems. The dilemma is solved by decreasing granularity with respect to both the subject that performs locking and the objects that are being locked. This means that tool sessions are considered as sequences of command executions, each of which is executed in isolation from concurrent commands. Isolation is achieved by locking objects in a traditional way [Gra78]. Locking is inferred from the use of objects and relationships and need not be specified explicitly. An object is locked in shared mode when the object is read and in exclusive mode when it is updated. While shared locks are compatible to each other, any other combination reveals a concurrency control conflict. To decrease the probability of concurrency control conflicts, commands do not lock the complete representation of a document version, but only those nodes that are being accessed or updated during the execution of the command. In the examples that encounter lost updates or inconsistent analysis problems, we would then obtain a concurrency control conflict. Tools react to these conflicts by delaying the execution of one command to await completion of the conflicting command, that is until conflicting locks have been released. This is appropriate because command execution requires only a few hundred milliseconds, which users will hardly recognise as delays.

Apart from the *isolation* property sketched above, interactions have further transaction properties. They are *atomic*, i.e. they are either performed completely or not at all. Once completed, the effect of an interaction is durable, i.e. all changes that were made during the interaction persist even if the tool is stopped accidentally by a hardware or software failure. Due to atomicity, tools then recover to the state of the last completed command execution.

```
INCREMENT SPECIFICATION TypeName;
  ...
  INTERACTIONS
    INTERACTION ChangeType
    NAME "Change Type"
    SELECTED IS SELF
    ON (SELF.expanded)
    VAR t:TEXT;
        err:TEXT_SET;
    BEGIN                              // start a new transaction
      t:=NEW TEXT(value);                       // read-lock SELF
      IF (t.LINE_EDIT("Enter New Type!")) THEN
       IF SELF.scan(t.CONTENTS()) THEN
         FOREACH i:TypeImport IN ExpTo DO
          i.react_to_change(t.CONTENTS())
         ENDDO;
         FOREACH i:UsingType IN UsedBy DO
          i.react_to_change(t.CONTENTS())
         ENDDO;
         value:=t.CONTENTS()                   // write-lock SELF
       ELSE
         err:=NEW TEXT_SET(SELF.get_errors());  // read-lock SELF
         err.DISPLAY;
       ENDIF
      ENDIF
    END ChangeType; // release all locks, make changes persistent
```

Figure 6: Command Definition to Change a Type

4.3 External Document Representation

The external document representation is determined in terms of unparsing schemes. Unparsing schemes are defined for non-terminal increment classes only. They cannot be defined for abstract increment classes. In that case abstract syntax children that might be added in subclasses would not be reflected. Neither are unparsing schemes required for terminal increment classes. For terminal increments the layout computation only needs to output the terminal increment's lexical value. As an example for a textual document representation consider the unparsing schemes of classes `ADTModule` and `OperationList` from the module interface editor below. The external representation of graphical documents must also define bitmaps or shapes of lines for increment representation purpose.

```
NONTERMINAL INCREMENT INTERFACE ADTModule;
 ...
 UNPARSING SCHEME
  "DATATYPE",WS,"MODULE",WS,name,";",(NL),(NL),
  ("   "),com,(NL),(NL),
  ("   "),"EXPORT",WS,"INTERFACE",(NL),(NL),
  ("      "),"TYPE",WS,typ,";",(NL),(NL),
  ("      "),op_list,(NL),
  ("   "),imp,(NL),(NL),
  "END",WS,"MODULE",WS,name,".",(NL)
 END UNPARSING SCHEME;
 ...
```

```
NONTERMINAL INCREMENT INTERFACE OperationList
 ...
 UNPARSING SCHEME
  ol DELIMITED BY (NL),(NL) END
 END UNPARSING SCHEME;
```

5 Evaluation

A compiler for GTSL has been implemented within the GOODSTEP project. It generates an object database schema for the O_2 database system [BDK92], which is used for persistent storage of abstract syntax graphs following the strategy discussed in [EKS93]. A detailed discussion of the implementation is beyond the scope of this paper and we refer to [Emm95]. GTSL has been evaluated within GOODSTEP on the basis of requirements provided by British Airways, an industrial GOODSTEP partner. British Airways does most of its software development in-house, with an increasing number of projects using object-oriented techniques. A department of the IT infrastructure division, with seven developers at present, is supposed to design, implement and document class libraries for the purpose of corporate reuse. The evaluation scenario, which we refer to as *BA SEE* in the following, is an environment that supports this class library development and maintenance process.

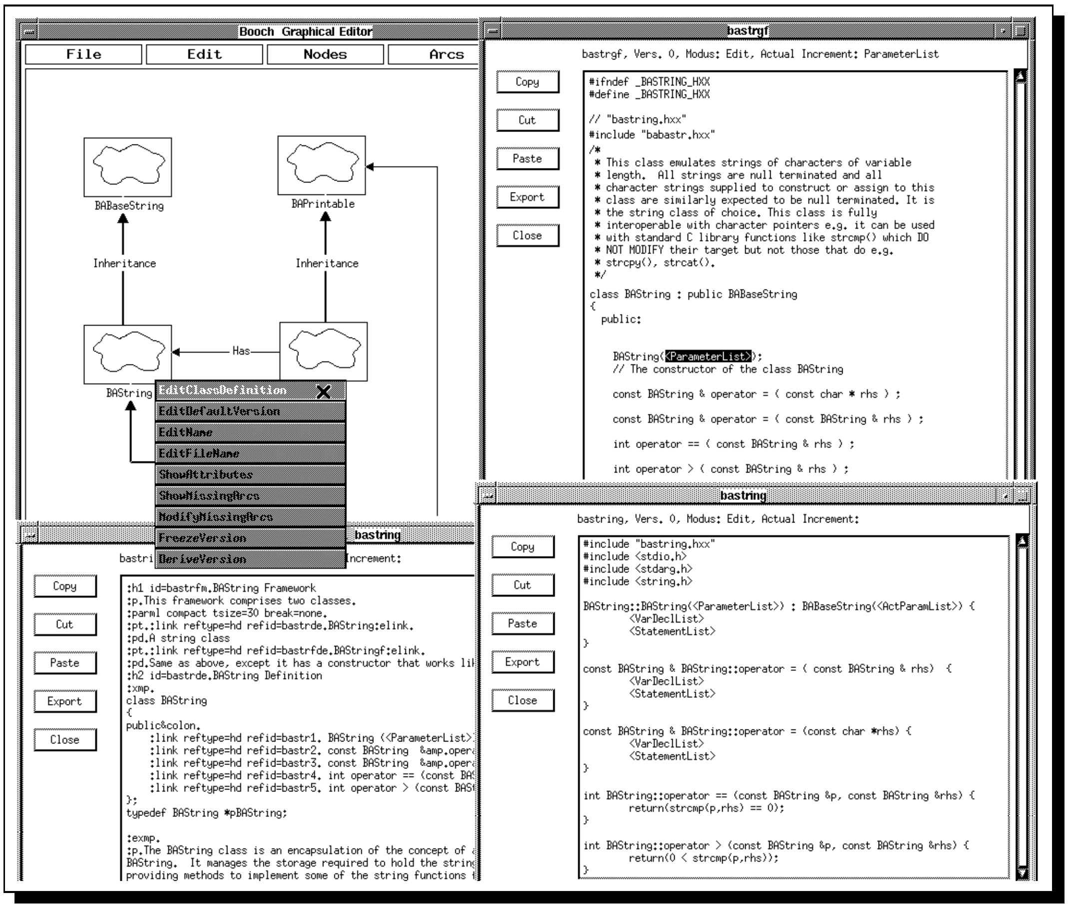

Figure 7: Tools Contained in the BA SEE

Design documents of class libraries are defined in the Booch notation [Boo91]. A Booch class diagram is structured into *categories*. A category may contain a number of classes and nested categories. The most recent definition of the Booch notation is strongly tight towards C++, which is the programming language used for any object-oriented development at British Airways. Booch diagrams identify different kinds of relationships between classes, namely inheritance, has- and use-relationships. They have different C++ specific *adornments*. Inheritance may, for instance, be public, protected or private.

The language used for class definitions at British Airways is, in fact, only a subset of the C++ programming language. The subset is determined by British Airways corporate programming guidelines. The guidelines exclude C++ statements, such as ellipses, inlines and friends, whose use would be contrary to accepted software engineering principles. The dependencies of a class definition document to other class definitions are defined in terms of `#include` statements and forward declarations.

An implementation has to be provided for each defined class. Therefore, a further document type includes C++ method implementations. Each method that has been defined in a class interface must be implemented in the respective class implementation document.

Since class libraries are developed for corporate reuse, they must be accompanied by documentation that allows customers to reuse classes from the library. Library customers require a technical documentation of the functionality provided by the public and protected methods of a class. This technical documentation is defined in the *information processing facility (IPF) language*, a mark-up language for online-hypertexts defined by IBM. For each class of the library an IPF document must be provided. Besides the signatures of public and protected methods of the class, it also includes a description of each method and examples illustrating how to use the methods. A hypertext, which can be compiled from an IPF document, is then delivered to customers as an on-line help facility.

The method integration defines numerous inter-document consistency constraints between the above documents. These are implemented by the tools contained in the environment. Each class in the Booch diagram must be refined in terms of a class definition, an implementation and an IPF documentation. Upon creation of a class, the Booch editor creates these documents as well. If a class name is changed, the Booch editor consistently changes the class names in the corresponding documents. Moreover, the relationships that are defined in the Booch diagram must be reflected in the class definition and implementation documents. The Booch editor creates, for instance, the corresponding C++ declarations for an inheritance relationship between two classes in the Booch diagram. Similarly, the integration of the three textual tools causes the insertion of the respective counterparts into the corresponding implementation and IPF documents, upon creation of a method. Furthermore, the matching of method signatures of the class definition and their corresponding definitions in the implementation and IPF documents is ensured.

The number of classes required for a tool specification is dominated by the number of productions in the grammar of the respective language. The C++ class definition grammar consists of 87 productions. 13 abstract classes were added to the non-terminal and terminal classes derived from the grammar to specify static semantics and inter-document consistency. The average size of a class specification depends largely on the number of tool-specific commands that have to be defined. In the C++ class definition tool a number of interactions had to be defined in order to meet the specific requirements of British Airways. The average size of a GTSL class was 2,730 Bytes. The complete C++ class definition tool was defined in a GTSL specification with 9,800 lines or 273 KBytes.

Compared to tools that can be bought of the shelf the environment has a number of advantages. Multiple users can access the same documents concurrently and immediately see each other's updates. Versions and configurations are not only managed for the source code, but also for Booch diagrams and IPF documentation. The tools enforce the corporate programming guidelines. They check and even preserve the numerous inter-document consistency constraints by change propagations.

6 Summary and Further Work

We have discussed the need for method definition and integration. Method definitions have to identify document types. Method integration must define inter-document consistency constraints. The application of methods should be supported by tools whose integration implements the method integration. We then have discussed why documents should be considered represented as project-wide abstract syntax graphs. Then we have outlined GTSL as a specification language capable to define these project-wide abstract syntax graphs as well as commands that are offered by tools to modify these graphs. GTSL has been used for the construction of an integrated development environment to support class library development and maintenance and we have sketched the evaluation results.

A result of the evaluation was that there are often document types, like C++ class interfaces and the corresponding IPF documents whose structure is so similar that the documents should not be stored redundantly. To improve efficiency and reduce the number of required change propagations these documents should be considered as different views of the same conceptual syntax graph. An extension of GTSL with language concepts to define different views has been done and it is now being implemented on the basis of a view mechanism for object-oriented databases [SAD94].

Different document versions can be managed on the basis of the version manager of the $O2$ database system [DM93]. The problem of configuration management has not yet been sufficiently addressed. Semantic relationships with other document versions are established during editing as determined by the semantic rules. They are, however, only created with those other versions of documents that have either been selected explicitly or are the default version. In that way a user accesses exactly one configuration at a time. What is not yet supported is the explicit construction of a configuration. To facilitate this, tools would have to compute the set of document versions that are consistent with each other. This obviously interferes with evaluation of semantic rules and it is not clear to us when the required evaluations can best be done.

Acknowledgements

We received considerable feedback on the appropriateness of GTSL during the construction of the BA SEE. I, therefore, thank Jim Arlow and Mark Phoenix from British Airways for exposing us to tool requirements from an industrial perspective. I thank a number of students, namely Werner Beckmann, Joerg Brunsman, Boris Gesell, Jens Jahnke, Matthias Kurth, Ralph Mertingk and Wiebke Reimer who contributed to the implementation of the GTSL compiler. I thank Prof. Uwe Kastens, Prof. Carlo Ghezzi and Prof. Wilhelm Schäfer for the discussions we had on GTSL. The presentation of this paper was improved by valuable comments that I got from Dr. Willi Hasselbring on an earlier draft.

References

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullmann. *Compilers – Principles, Techniques and Tools.* Addison Wesley, 1986.

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis. *Building an Object-Oriented Database System: the Story of O_2*. Morgan Kaufmann, 1992.

[Boe88] B. W. Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, pages 61–72, May 1988.

[Boo91] G. Booch. *Object Oriented Design with Applications.* Benjamin/Cummings, 1991.

[Dat86] C. J. Date. *Introduction to Database Systems, Vol. 1.* Addison Wesley, 1986.

[DGKLM84] V. Donzeau-Gouge, G. Kahn, B. Lang, and M. Mélèse. Document structure and modularity in Mentor. *ACM SIGSOFT Software Engineering Notes*, 9(3):141–148, 1984. Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Penn.

[dM78] T. de Marco. *Structured Analysis and System Specification.* Yourdan, 1978.

[DM93] C. Delobel and J. Madec. Version Management in O_2. Technical report, O_2-Technology, 1993.

[EKS93] W. Emmerich, P. Kroha, and W. Schäfer. Object-oriented Database Management Systems for Construction of CASE Environments. In V. Mařik, J. Lažanksý, and R. R. Wagner, editors, *Database and Expert Systems Applications — Proc. of the 4^{th} Int. Conf. DEXA '93, Prague, Czech Republic*, volume 720 of *Lecture Notes in Computer Science*, pages 631–642. Springer, 1993.

[ELN+92] 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.

[ELS87] G. Engels, C. Lewerentz, and W. Schäfer. Graph-grammar engineering: A Software Specification Method. In *Proc. of the 3^{rd} Int. Workshop on Graph Grammars and their Application to Computer Science*, volume 291 of *Lecture Notes in Computer Science*, pages 186–201. Springer, Berlin, 1987.

[Emm95] W. Emmerich. *Tool Construction for Process-Centered Software Development Environments based on Object Database Systems.* PhD thesis, University of Paderborn, Germany, 1995. Forthcoming.

[ES94] W. Emmerich and W. Schäfer. Groupie — An Environment supporting Group-Oriented Architecture Development. Technical Report 71, University of Dortmund, Dept. of Computer Science, Chair for Software Technology, 1994. Submitted for Publication.

[GMT87] F. Gallo, R. Minot, and I. Thomas. The Object Management System of PCTE as a Software Engineering Database Management System. *ACM SIGPLAN NOTICES*, 22(1):12–15, 1987.

[Gra78] 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.

[HN86] A. N. Habermann and D. Notkin. Gandalf: Software Development Environments. *IEEE Transactions on Software Engineering*, 12(12):1117–1127, 1986.

[Hoo87] R. Hoover. *Incremental graph evaluation.* PhD thesis, Cornell University, Dept. of Computer Science, Ithaca, NY, 1987. Technical Report No. 87-836.

[JF82] G. F. Johnson and C. N. Fisher. Non-syntactic attribute flow in language based editors. In *Proc. of the* 9^{th} *Annual ACM Symposium on Principles of Programming Languages*, pages 185–195. ACM Press, 1982.

[Knu68] D. E. Knuth. Semantics of Context-Free Languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[KW91] U. Kastens and W. M. Waite. An abstract data type for name analysis. *Acta Informatica*, 28:539–558, 1991.

[Nag85] M. Nagl. An Incremental and Integrated Software Development Environment. *Computer Physics Communications*, 38:245–276, 1985.

[Par72] D. C. Parnas. A Technique for the Software Module Specification with Examples. *Communications of the ACM*, 15(5):330–336, 1972.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[Roy70] W. W. Royce. Managing the Development of Large Software Systems. In *Proc. WESCON*, 1970.

[RT81] T. W. Reps and T. Teitelbaum. The Cornell Program Synthesizer: A syntax-directed programming environment. *Communications of the ACM*, 24(9):449–477, 1981.

[RT84] T. W. Reps and T. Teitelbaum. The Synthesizer Generator. *ACM SIGSOFT Software Engineering Notes*, 9(3):42–48, 1984. Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Penn.

[SAD94] C. Santos, S. Abiteboul, and C. Delobel. Virtual Schemas and Bases. In J. Bubenko M. Jarke and K. Jefferey, editors, *Proc. of the* 4^{th} *Int. Conf. on Extending Database Technology, Cambridge, UK*, volume 779 of *Lecture Notes in Computer Science*, pages 81–94. Springer, 1994.

[Sch91] A. Schürr. *Operationales Spezifizieren mit programmierten Graphersetzungssystemen.* PhD thesis, RWTH Aachen, 1991.

[SS95] S. Sachweh and W. Schäfer. Version Management for tightly integrated Software Engineering Environments. In M. S. Verrall, editor, *Proc. of the* 7^th *International Conference on Software Engineering Environments, Nordwijkerhout, The Netherlands*, pages 21–31. IEEE Computer Society Press, 1995.

[Tho89] I. Thomas. Tool Integration in the PACT Environment. In *Proc. of the* 11^{th} *Int. Conf. on Software Engineering, Pittsburg, Penn.*, pages 13–22. IEEE Computer Society Press, 1989.