# Tool Specification with GTSL*

Wolfgang Emmerich
Dept. of Computer Science, City University London,
Northampton Square, London EC1V 0HB, UK

## Abstract

*The definition of software development methods encompasses the definition of syntax and static semantics of formal 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. Method integration must determine inter-document consistency constraints between documents produced in the various tasks. Tools must, therefore, be integrated to implement the required method integration and check or even preserve inter-document consistency. The focus of this paper is on the specification of such integrated tools and outlines the main concepts of the object-oriented tool specification language GTSL.*

## 1 Introduction

A software process that develops and maintains a software system performs a number of different *tasks*. Examples are *requirements analysis* tasks where the 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 [21] that these tasks be performed in mutual exclusion has been proved infeasible [3]. Instead the tasks are often carried out in an incremental and intertwined manner [23, 12].

Tasks are performed using *development methods*, like *structured analysis* [6] for requirements analysis or object-oriented methods (for instance the Booch methodology [4]) for architectural design. Method definitions have to determine formal graphical or textual *languages*. Examples are data flow diagrams or class hierarchies. 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 the syntax and static semantics of the underlying formal 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. 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.

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 a banking application, where response time constraints need not be expressed. This means that it is impossible to find **the** mix of methods that could be used in arbitrary software processes. Therefore, method integration must become part of process modelling and deserves appropriate attention.

Development 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. The language is executable and the GTSL compiler GENESIS generates tools from GTSL specifications.

The rest of this paper will be structured as follows.

*This work has been partly funded by the CEC as part of ESPRIT-III project 6115 (GOODSTEP) and was done while the author was at University of Dortmund, Germany.

[1]GOODSTEP's aim is to enhance a general object-oriented database for software engineering processes.

In the next section, we discuss the need for method and tool customisation 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. We conclude the paper in Section 5 with work that remains to be done.

## 2 Method and Tool Customisation

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 definition that identifies exported types and operations as well as an import interface, and a module 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 a module interface definition. Operations and types that have been identified in the export interface must be implemented in the C document. Therefore, parameter lists and result types in the interface design and in the C document should match. Moreover, import interfaces are refined by preprocessor `#include` statements. Conversely, 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.

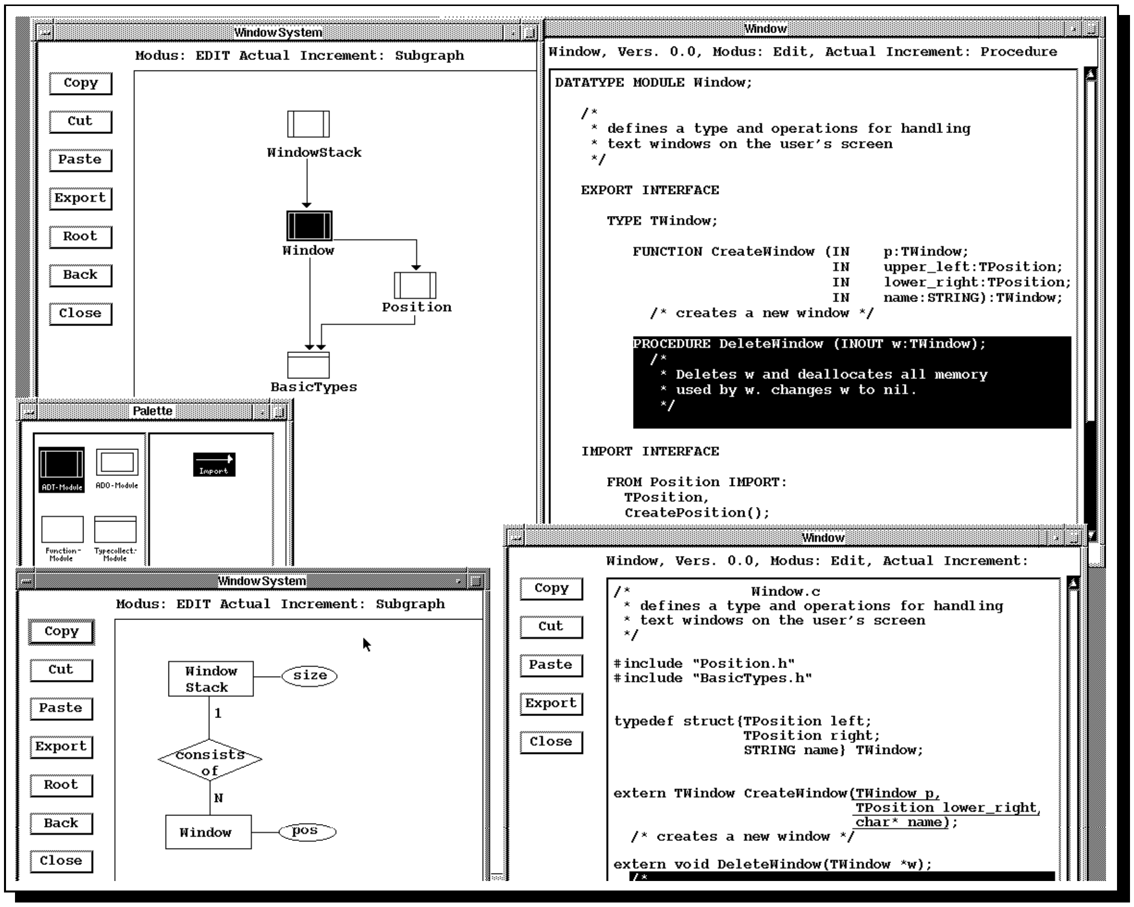
The need arises to assist software developers in the production of documents that meet inter-document consistency constraints such as those outlined above, and thus to implement the methods and their integration. Users[3] require a *tool* for each document type. Such a tool should support the methods and offer commands to edit documents of that type. It should be supportive in achieving syntactic and static semantic correctness of documents, browsing of semantically related documents and most importantly, it must check for inter-document consistency. This requires that tools be aware of the syntactic structure of documents. We denote different syntactic units of documents as *increments* because users can modify these units incrementally. The granularity of increments can range from complete documents to single identifiers. The example of Figure 1 displays the user interfaces of tools contained in the *Groupie* environment [11]. 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 tolerate a violation 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 increment 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. Alternatively, a tool might follow an intolerant 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*.

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. 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. In the above example a requirements engineer might use the entity relationship tool to define an information model of a software system, while a designer

[2]The detailed notion is of no concern here and we refer to [11].

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



Window, Vers. 0.0, Modus: Edit, Actual Increment: Procedure

```
DATATYPE MODULE Window;

   /*
    * defines a type and operations for handling
    * text windows on the user's screen
    */

   EXPORT INTERFACE

      TYPE TWindow;

         FUNCTION CreateWindow (IN    p:TWindow;
                                IN    upper_left:TPosition;
                                IN    lower_right:TPosition;
                                IN    name:STRING):TWindow;
           /* creates a new window */

         PROCEDURE DeleteWindow (INOUT w:TWindow);
           /*
            * Deletes w and deallocates all memory
            * used by w. changes w to nil.
            */

   IMPORT INTERFACE

      FROM Position IMPORT:
        TPosition,
        CreatePosition();
```

Window, Vers. 0.0, Modus: Edit, Actual Increment:

```
/*            Window.c
 * defines a type and operations for handling
 * text windows on the user's screen
 */

#include "Position.h"
#include "BasicTypes.h"


typedef struct{TPosition left;
               TPosition right;
               STRING name} TWindow;


extern TWindow CreateWindow(TWindow p,
                            TPosition lower_right,
                            char* name);
  /* creates a new window */

extern void DeleteWindow(TWindow *w);
  /*
```

Figure 1: User Interface of Groupie Environment

is defining its architecture. 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 corresponding versions of the entity relationship and the architecture diagram concurrently.

To reach a state of consistency, users want to see the impact of concurrent document updates as soon as possible. Tight cooperation 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 version of 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.

## 3 Document Representation

Before we can identify concepts of a higher-level specification language for the definition of methods and their integration, we 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 [20, 8, 13]. Nodes in the abstract syntax tree often have additional attributes whose values represent semantic information such as references to a string ta-

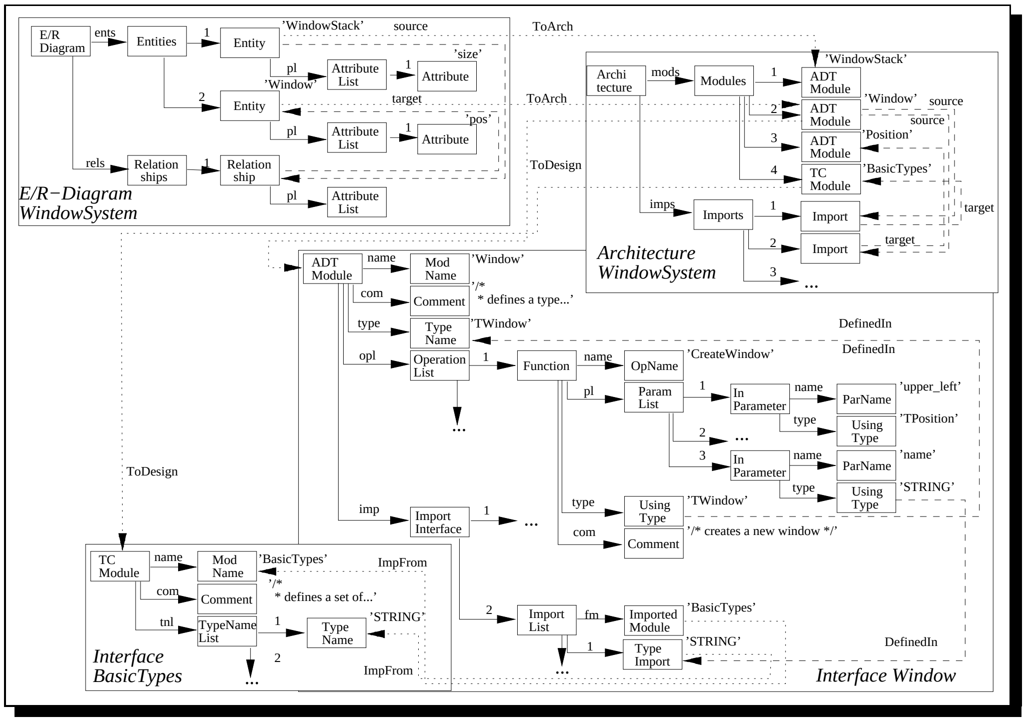


Figure 2: Excerpt of a Project-wide Abstract Syntax Graph

ble, symbol tables or type information. Operations, like insertion of a new parameter list place holder can be implemented as subtree replacements. After free textual input, the abstract syntax tree can be established with parsing techniques well-known from compiler construction [1].

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 [17, 20]. The evaluation paths are computed at tool construction time based on attribute dependencies. If inter-document consistency checks are implemented by attribute evaluations, 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 tools' performance. Therefore, techniques based on the introduction of additional, non-syntactic paths for more direct attribute propagation have been developed [15, 18, 14]. 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*.

In such a graph, we distinguish between *aggregation edges*, which implement syntactic relationships, from *reference edges*, which arise from semantic relationships. A document is then represented by the subgraph whose node-set is the closure of nodes reachable by aggregation edges from a document root node. 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 are 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 (drawn as dotted arrows) 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 `ToDesign`.

Intra-document reference edges (drawn with dashed arrows) 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` (given in quotes), 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 [12] 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 can hardly be managed appropriately.

## 4 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 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 language grammars. 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*. Import/Export relationships between tool configurations make the dependencies between different tool specifications explicit and simplify customisation and reuse. 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 standard scoping rules.

Due to the heterogeneity of the different static and behavioural concerns, it is impossible to find a unique formalism that is appropriate for their specification. Instead, we separate the different concerns and offer the most appropriate formalism for each of them. We integrate these different formalisms into a domain-specific *multi-paradigm language* that uses rule-based, object-oriented and imperative concepts.

Following the principle of information hiding [19], the definition of a class is 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.

The node types in an abstract syntax graph definition play different roles. *Terminal classes*, which define leaf 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 starting from nodes of a type are defined within an *abstract syntax section* in the increment class interfaces that specifies the type. The abstract syntax section is available for abstract and non-terminal increment classes. 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, which are induced by the inheritance hierarchy. Multi-valued aggregation edges are defined by the `LIST` type constructor. Below there are several examples of abstract syntax sections for node types displayed in Figure 2. Note, how `ADTModule` reuses definitions inherited from `Module`.

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

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

**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. The *explicit link* denotes the direction from the source to the target increment class. The *implicit link* denotes the reverse direction. Both kinds of links can be single- and multi-valued so as to allow for 1:1, 1:n and m:n relationships.

Relationships are created and deleted during static semantics and inter-document consistency analysis. Creation of a relationship is specified by assigning an expression that denotes an increment to an explicit link. 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. As an example, consider the relationships `DefinedIn/UsedBy` between `UsingType` and `TypeDecl`.

```
TERMINAL INCREMENT INTERFACE UsingType;
 INHERIT UsingName;
 SEMANTIC RELATIONSHIPS
  DefinedIn: TypeDecl
 END SEMANTIC RELATIONSHIPS; ...
```

```
ABSTRACT INCREMENT INTERFACE TypeDecl;
 INHERIT DefiningName;
 SEMANTIC RELATIONSHIPS
  IMPLICIT UsedBy:SET OF UsingType.DefinedIn;
 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. If `TypeName` and `TypeImport` are declared as subclasses of `TypeDecl`, polymorphism can be exploited to have DefinedIn edges to the two classes as they occur in Figure 2. UsedBy then refers to the set of increments that use the type declarations.

## 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.

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 online, 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* [16]. 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 definitions 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 below.

```
INCREMENT SPECIFICATION ScopingBlock;
 INITIALIZATION
  DefinedNames := NEW DuplicateSymbolTable;
 END INITIALIZATION;
 SEMANTIC RULES
  ON EXISTS(name:DefiningName IN IncludedNames):
           CHANGED(name.value);
  ACTION
   DefinedNames.associate(name,name.value);
  END ACTION;
  ON EXISTS(name:DefiningName IN IncludedNames):
           DELETED(name);
  ACTION
   DefinedNames.deassociate(name,name.value);
  END ACTION;
 END SEMANTIC RULES;
END INCREMENT SPECIFICATION ScopingBlock.
```

In these two rules, IncludedNames denotes the implicit link of a semantic relationship between class DefiningName and ScopingBlock. We can assume that it is established during construction of defining name increments. The first rule then fires whenever a new declaring increment is created or its value is being changed. Then the symbol table is updated to include the association between the new value and the increment declaring the value. If a name in the scope is deleted, the respective association is removed from the table by the second rule. Then the symbol table can be accessed from semantic rules in class DefiningName to check for uniqueness of names and from class `UsingName` to check for existence of applied occurrences of names.

**Interactions:** The steps of a software development method are implemented by the commands that the tool offers. The command definition must determine the names of commands, preconditions 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 below is considered to be offered if the selected increment is a type name.

```
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   // read-lock SELF
     err:=NEW TEXT_SET(SELF.get_errors());
     err.DISPLAY;
    ENDIF
   ENDIF
  // release all locks, changes persist
  END ChangeType;
```

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 semantic relationships of a type name will be 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 will be 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 [5]. 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. 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.

### 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 `OperationList` and `ADTModule` below.

```
NONTERMINAL INCREMENT INTERFACE OperationList;
 UNPARSING SCHEME
  ol DELIMITED BY (NL),(NL) END
 END UNPARSING SCHEME; ...
NONTERMINAL INCREMENT INTERFACE ADTModule; ...
 UNPARSING SCHEME
  "DATATYPE",WS,"MODULE",WS,name,";",(NL),(NL),
  ("   "),com,(NL),(NL),
  ("   "),"EXPORT",WS,"INTERFACE",(NL),(NL),
  ("      "),"TYPE",WS,type,";",(NL),(NL),
  ("         "),opl,(NL),
  ("   "),imp,(NL),(NL),
  "END",WS,"MODULE",WS,name,".",(NL)
 END UNPARSING SCHEME; ...
```

## 5 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 method integration. We then have discussed why documents should be 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. The implementation of the GTSL compiler GENESIS generates ASG schemas for the $O_2$ database system [2] as discussed in [9]. Standard database transactions are exploited to implement interactions.

GTSL has been evaluated within the GOODSTEP project for the construction of an SEE for British Airways. An account on this evaluation is given in a companion paper [10]. One of the results of this evaluation was that often document types, such as module interface definitions and the corresponding implementations have a similar structure and documents should, therefore, 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 [22].

Different document versions can be managed on the basis of the version manager of the $O_2$ database system [7]. 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 yet clear to us when the required evaluations can best be done.

## Acknowledgements

I thank a number of my students, namely Werner Beckmann, Jörg Brunsman, Boris Gesell, Jens Jahnke, Matthias Kurth, Ralph Mertingk, Wiebke Reimer and Mike Wagener who contributed to the implementation of the GTSL compiler. I thank Uwe Kastens, Carlo Ghezzi and Wilhelm Schäfer for the discussions we had about GTSL. The presentation was improved by the valuable comments I got from Jun Han, Willi Hasselbring, Jim Welsh and the anonymous referees on earlier drafts of this paper.

## References

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

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

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

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

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

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

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

[8] 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.

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

[10] W. Emmerich, J. Arlow, J. Madec, and M. Phoenix. Construction of the British Airways SEE with the $O_2$ ODBMS. Technical report, City University London, Dept. of Computer Science, 1996. To appear.

[11] 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.

[12] 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.

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

[14] R. Hoover. *Incremental graph evaluation.* PhD thesis, Cornell University, Dept. of Computer Science, Ithaca, NY, 1987.

[15] 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.

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

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

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

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

[20] T. W. Reps and T. Teitelbaum. The Synthesizer Generator. *ACM SIGSOFT Software Engineering Notes*, 9(3):42–48, 1984.

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

[22] C. Santos, S. Abiteboul, and C. Delobel. Virtual Schemas and Bases. In M. Jarke, J. Bubenko, 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.

[23] 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.