Version Management for tightly integrated Software Engineering Environments

Sabine Sachweh and Wilhelm Schäfer
Universität GH Paderborn
Fachbereich 17 - Mathematik/Informatik
D-33095 Paderborn
Germany
sachweh@uni-paderborn.de
wilhelm@uni-paderborn.de

Abstract

The paper introduces a version management model which exploits knowledge about the contents of documents. This is in contrast to most existing models which basically consider versioned objects as flat (attributed) files. The benefits of the approach are illustrated by describing some sample operations which are not possible with a conventional model. The paper then discusses a feasible implementation of the model on top of an existing object-oriented data base management system. Finally, it discusses related work and indicates how a sophisticated configuration management system is being built on top of the version management system.

1 Introduction

The enormous increase in the size of software and the reliability required from modern software intensive systems has focussed attention on languages and corresponding tools which do not only support programming but also specification, design, and documentation of software. In fact, specification or design documents have become equally important as the final code in order to check and verify correctness, completeness and reliability of a delivered software product. In addition, the industrial-scale production of software today requires teams of developers who should be supported by tools which organise access to a large amount of shared and frequently changing information. This information consists of the above mentioned documents for the various phases of the software production process. Examples for such documents are data-flow diagrams, state-transition diagrams, petri-nets, entity-relationship diagrams, and modular design descriptions.

Tools support the syntactically and static semantically correct construction of documents which is denoted as *intra-document consistency.* A tightly integrated software engineering environment (SEE) is a collection of tools supporting various phases of the production process. Its main additional feature compared with a tool which supports the construction of a single document in a single language, is the possibility to maintain so-called *inter-document dependency* and consistency. An example for inter-document consistency is a name of a function which should be the same in the requirements specification, in the interface definition of a module in the design document and in the implementation.

The advantage of an SEE which maintains such fine-grained dependencies between documents is its support of an incremental, intertwined production and maintenance process (in contrast to a waterfall- or phase-oriented approach). Consider as an example the situation that a programmer may have detected an error in the code which is due to a wrong requirements specification of a particular function. If then the specification is changed, the environment could inform the user about all other places in all other documents which are affected by this change. It could even propagate a change trough all documents concerned automatically, if the user(s) require it. In contrast, doing such a small incremental change in a phase-oriented environment is rather tedious, because such an environment is usually based on a complete transformation of all documents concerned from one phase into the next one. For more details we refer to [ELN+92].

Furthermore, an SEE should support change control and multiple users. This requires a sophisticated version and configuration management system (VM/CM). Although, SEEs which maintain fine-grained dependencies as mentioned above, are now existing [Lew88], most available VM/CM models still assume a very coarse-grained object model. Basically, versioned objects correspond to files whose particular contents is not considered e.g. [Tic82, Fel88, BE87, ML88]. Such a coarse-grained version model makes it impossible to preserve inter-document dependencies and to control change propagation on the fine-grained level sketched above. Illustrating examples will follow in the next section.

The main new features presented in this paper are:

- the concept of a version model which is tuned to fine-grained inter-document dependencies and corresponding change control
- a feasible approach how such a concept is implemented on top of an existing commercially available fully object-oriented database system.

The reminder of this paper is organised as follows: The next section describes and formalises our underlying version model whereas section 3 gives an overview on the version operations which are available to a user. This section tries to clarify what are the benefits of our model. Section 4 describes how such a model is implemented on top of an object-oriented data base management system and thus provides evidence that the approach is feasible. Section 5 deals with related work and section 6 sketches ongoing work.

2 The VM Model

The VM Model is defined by a *system version graph*, which is a two level graph structure which consists of the *version level* and the *increment level*. The objects (nodes) on the version level correspond to document versions, whereas the nodes on the increment level correspond to the syntactic constructs of a document version, i.e. document versions are represented internally by abstract syntax graphs. The relations (edges) on the version level are refined on the increment level analogously to the nodes. Consequently, the increment level represents the refinement of the version level. Note, that the granularity of versioning is a document, consequently the term *version* is used as synonym for a *document version* in the following. Furthermore, the increment level is needed to preserve the consistency of document versions, whereas the version level is used as base for configuration management.

2.1 Version Level

The version level contains a *document version graph* for each document of the system, which represents the development history of this document. A document version graph consists of nodes which represent the different versions (and their refinements as explained in the following section) and a *successor relation* which defines the development history. A document version graph is single rooted, since the development of each document starts with an initial version from which all other versions of this document are derived, and it is acyclic, because it represents the development steps during the production of a document.

Inter-document dependencies, as sketched in the previous section, are expressed by further relations which connect versions in different document version graphs. We call the source version of a relation *depending version* and the target version *determining version* respectively. Basically, a version can be a *depending version* and a *determining version* at the same time. For example, a design version may depend on another design version and it determines an implementation version. Two types of inter-document dependencies are distinguished, which are the *based_on relation* and the *depend_on relation*, as is illustrated in figure 1.

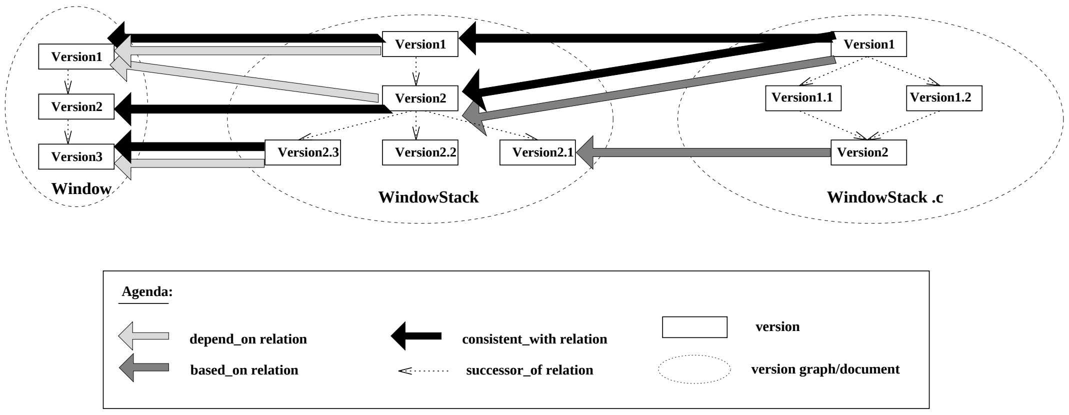
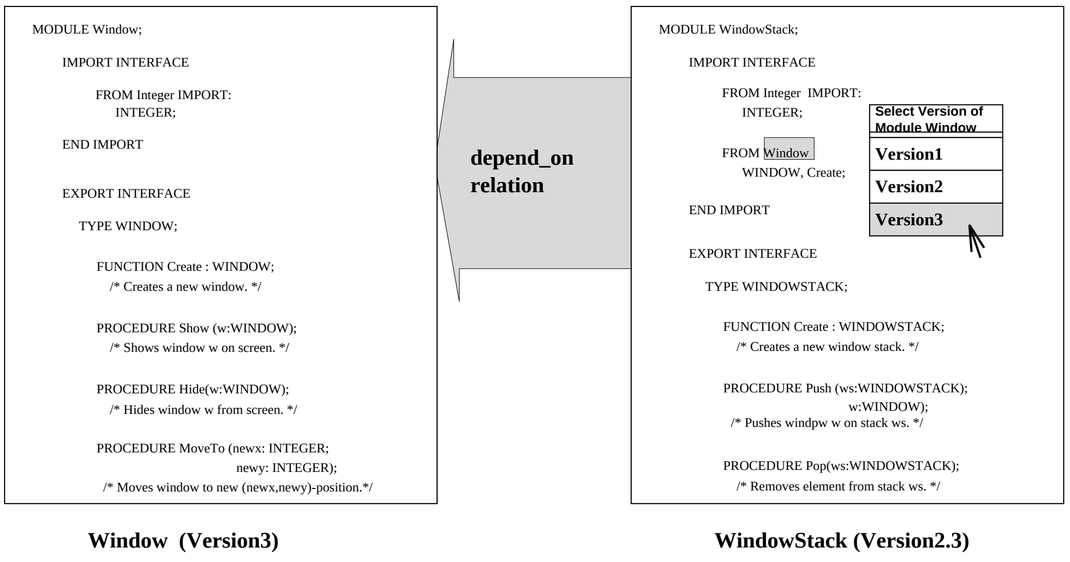


Figure 1: **An example for the version level of a system version graph**

A *depend_on relation* describes inter-document dependencies which are defined by referencing a version based on the contents of a version of another document. As an example assume that *Window* and *WindowStack* are excerpts from a system design which is given in terms of modules and use-relationships between modules which are defined by a MIL (Module Interconnection Language). In addition assume that the two modules have been designed rather independently possibly by two different designer (groups). *WindowStack* depends on *Window* because it imports resources defined in the export interface of window. Thus a system designer has to indicate explicitly which version of the Window should be used, as shown in figure 2. (Note that the export interface of different versions of *Window* could actually be different.) Note that figure 2 does not show an extract of a system version graph, but it

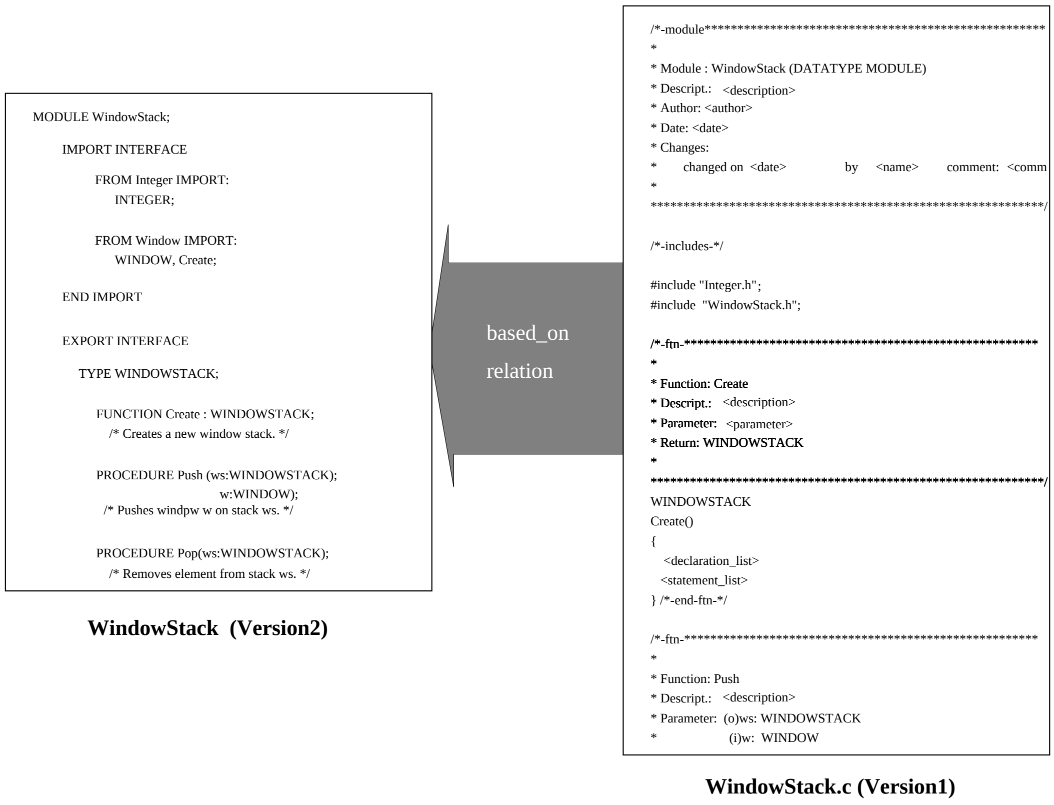

```
MODULE Window;
    IMPORT INTERFACE
         FROM Integer IMPORT:
            INTEGER;
    END IMPORT

    EXPORT INTERFACE
       TYPE WINDOW;

         FUNCTION Create : WINDOW;
           /* Creates a new window. */

         PROCEDURE Show (w:WINDOW);
           /* Shows window w on screen. */

         PROCEDURE Hide(w:WINDOW);
           /* Hides window w from screen. */

         PROCEDURE MoveTo (newx: INTEGER;
                           newy: INTEGER);
           /* Moves window to new (newx,newy)-position.*/
```

```
MODULE WindowStack;
    IMPORT INTERFACE
         FROM Integer  IMPORT:
            INTEGER;

         FROM Window
            WINDOW, Create;
    END IMPORT

    EXPORT INTERFACE
       TYPE WINDOWSTACK;

         FUNCTION Create : WINDOWSTACK;
           /* Creates a new window stack. */

         PROCEDURE Push (ws:WINDOWSTACK;
                         w:WINDOW);
           /* Pushes windpw w on stack ws. */

         PROCEDURE Pop(ws:WINDOWSTACK);
           /* Removes element from stack ws. */
```

Figure 2: **An example for explicit referencing under version control**

gives an example how *depend_on relations* of a system version graph can be established in a textual representation, i.e. how the user sees them.

A *based_on relation* describes that a text frame of the *depending version* was generated or updated based on the corresponding *determining version*, like for example the generation of code-frames based on a design document. Changing the depending version means to fill in the generated frame, i.e. the generated frame of the *depending version* cannot be changed. Consider for example figure 3 that shows a text frame for a c-program named *WindowStack.c (Version1)* which was generated from a design document named *WindowStack (Version2)*. For each import-clause in the design version an include statement was generated and for each exported operation, the interface of the corresponding c-function was generated. This means almost all increments of *WindowStack.c (Version1)* shown in this figure are generated except of those enclosed in brackets (<...>), i.e. only the latter increments can be expanded in order to fill in the frame. The consistency between these versions is ensured by propagating changes from the *determining version* to the *depending version*. Again, analogously to figure 2 figure 3 does not



```
MODULE WindowStack;

     IMPORT INTERFACE

          FROM Integer IMPORT:
             INTEGER;

          FROM Window IMPORT:
             WINDOW, Create;

     END IMPORT

     EXPORT INTERFACE

        TYPE WINDOWSTACK;

          FUNCTION Create : WINDOWSTACK;
            /* Creates a new window stack. */

          PROCEDURE Push (ws:WINDOWSTACK);
                                  w:WINDOW);
            /* Pushes windpw w on stack ws. */

          PROCEDURE Pop(ws:WINDOWSTACK);
            /* Removes element from stack ws. */
```

```
/*-module*****************************************************
*
* Module : WindowStack (DATATYPE MODULE)
* Descript.:  <description>
* Author: <author>
* Date: <date>
* Changes:
*      changed on  <date>            by    <name>      comment:  <comm
*
*************************************************************/

/*-includes-*/

#include "Integer.h";
#include  "WindowStack.h";

/*-ftn-*******************************************************
*
* Function: Create
* Descript.:  <description>
* Parameter:  <parameter>
* Return: WINDOWSTACK
*
*************************************************************/
WINDOWSTACK
Create()
{
   <declaration_list>
  <statement_list>
} /*-end-ftn-*/

/*-ftn-*******************************************************
*
* Function: Push
* Descript.:  <description>
* Parameter:  (o)ws: WINDOWSTACK
*                (i)w:  WINDOW
```

Figure 3: **An example for a generation dependency**

show an excerpt of a system version graph, but a user's view on two particular versions.

A final relation which is established on the version level is the *consistent_with relation*. It describes that two versions are consistent with each other. This relation will form the basis for a corresponding configuration management model, i.e. configurations are usually built based on consistent versions.

Two versions v_1 (determining version) and v_2 (depending version) are consistent concerning a *based_on* or *depend_on relation*, if for each increment i_2 of v_2 that is determined by an increment i_1 of v_1 the following equation holds: $i_2 = i_1$. (Instead of taking the identity function (as chosen in this paper) any other function f can be used, i.e. more general the equation to be held is: $i_2 = f(i_1)$.)

The *based_on relation* and the *depend_on relation* only describe dependencies between versions but do not define whether or not two versions are consistent. Remember the example shown figure 2. If the exported type *WINDOW* in version 3 of document *Window* is changed to *SCROLLBAR-WINDOW*, version 2.3 of document *WindowStack* is still depending on this version of *Window*, because of its import-clause. But both versions are not consistent any longer, since the type *WINDOW* has not been changed in the import part of version *WindowStack* so far.

2.2 Increment Level

We have distinguished different types of inter-document dependencies and corresponding relations on the version level based on how the source and target versions of those dependencies have been created. The edges on the version level however do not provide the information on the level of version contents, e.g. the *depend_on relation* and possibly the *consistent_with relation* between *WindowStack* and *Window* do not describe that this dependency is in particular, due to the import of type *WINDOW* and function *Create*. Therefore, it is not possible to identify only by the information which is provided by the version level, that a change of procedure *Show* or *Hide* respectively (e.g. extending the parameter list) would not violate the consistency between *Window* and *WindowStack*.

In order to provide this missing information, version nodes, *depend_on relations* and *based_on relations* are refined on the increment level. A version is represented by its abstract syntax graph (ASG). This is expressed by a *has_contents relation* between a node on the version level and the root node of the abstract syntax graph of its corresponding version. Similarly to nodes the *depend_on, based_on* and *the successor_of relation* are refined on the increment level. Those relations start at nodes of the ASG of the source version and end at corresponding nodes of the ASG of the target version. Corresponding nodes means the root nodes of the syntactic constructs (in the ASG-representations) which correspond to the constructs in the versions concerned which have been either identified explicitly by the user in the case of a *depend_on relation* (e.g. type *WINDOW* and procedure *Create* in Fig. 2) or which are defined by the mapping between the language of the determining and the language of the depending version in the case of a *based_on relation* (e.g. the *IMPORT INTERFACE* and the *include* commands in Fig. 3). Figure 4 shows the underlying graph structure of the example given in figure 2. Note, that the *successor_of relation* is refined analogously.

The only relation of the version level, that is not refined on the increment level is the *consistent_with relation*. The consistency concerning an existing inter-document dependency is checked automatically by the environment using the *fine-grained based_on* or *depend_on relations*, i.e. it is derived from the information available through the latter two relations and the definition of a consistent state between two versions.

2.3 Formalisation of the Graph Model

So far, we have described informally our version model which basically corresponds to a two-level graph structure. This graph structure is now formalised which enables to formally define the consistent, i.e. practically reasonable states of a version graph. Those definitions are the basis for the implementation of the VM-operations which are explained in the next section, i.e. they can be considered as the definitions of the operations' invariants (similarly to the definition of invariants in a language like EIFFEL).

A system version graph consists of two disjoint node subsets, namely VER and INC, which represent the nodes of the version level and increment level respectively. Analogously, the set of edges consists of three disjoint sets of edges, namely VER_REL, INC_REL and VER_INC_REL, whereas VER_REL denotes a multi-set since the *depend_on relation* as well as the *based_on relation* may be defined between the same nodes as the *consistent_with relation*.

VER_REL in turn consists of sets which represent the *depend_on*, *based_on*, *successor_of* and *consistent_with relation* of the version level and analogously INC_REL consists of sets which represent the *abstract_syntax*, *stat_semantic*, *fine grained depend_on*, *fine grained based_on* and *fine grained successor_of relation* of the increment level:

$$\begin{aligned} VER\_REL = {} & DEPEND\_ON \cup BASED\_ON \cup SUCCESSOR\_OF \cup \\ & CONSISTENT\_WITH. \\ & \subseteq VER \times VER \end{aligned}$$

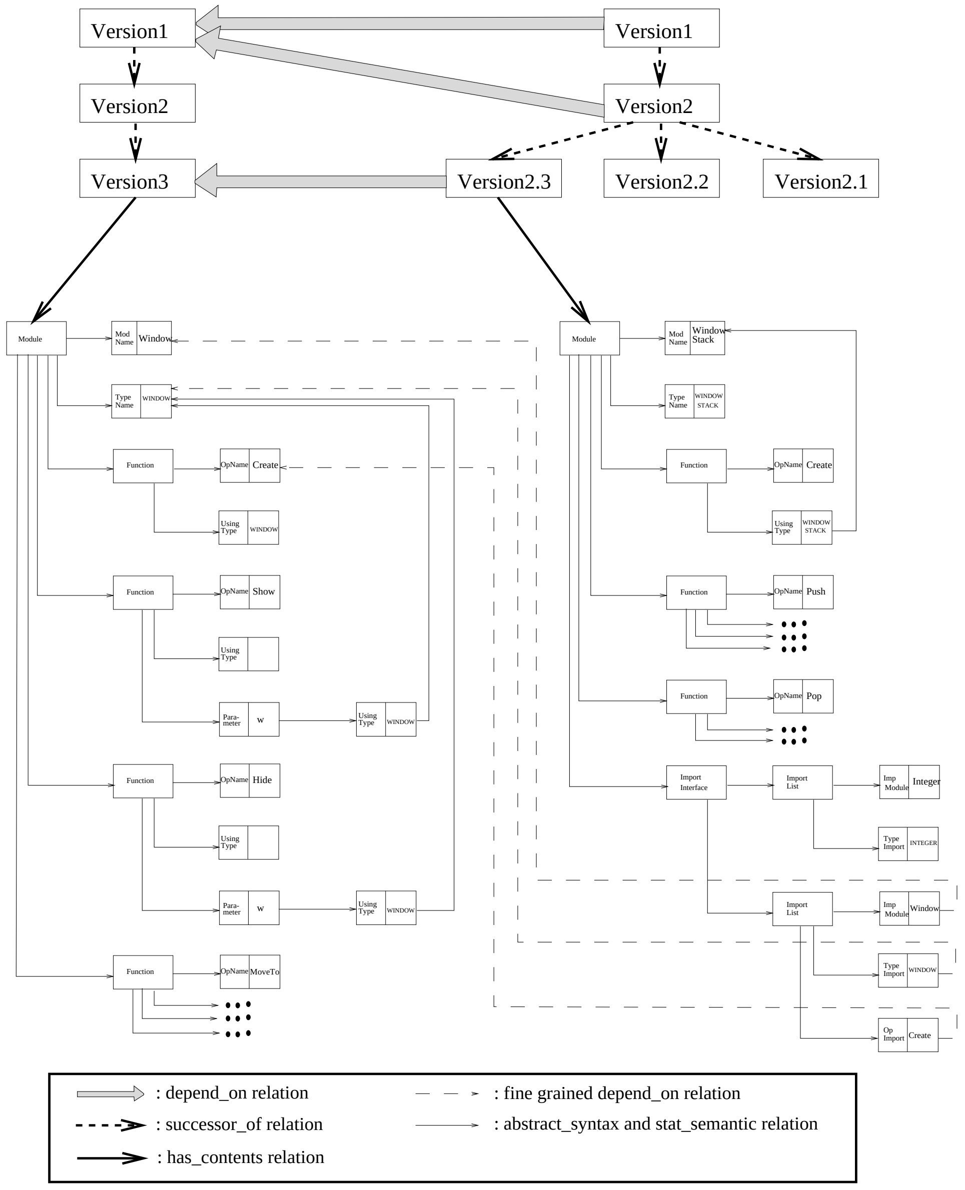


Figure 4: **An example for a system version graph**

$$\begin{aligned} INC\_REL = \; & ABSTRACT\_SYNTAX \cup STAT\_SEMANTIC \cup REF\_DEPEND\_ON \cup \\ & REF\_BASED\_ON \cup REF\_SUCCESSOR\_OF \\ . \quad & \subseteq INC \times INC. \end{aligned}$$

The VER_INC_REL set contains all edges of the type *has_contents*, which are edges connecting both levels:

$$VER\_INC\_REL = HAS\_CONTENTS \subseteq VER \times INC$$

First order logic is used to formally describe which restrictions hold for the combination of those edges, i.e. the above mentioned consistent states of a version graph are defined. We do not give all restrictions here but rather sketch a few examples to illustrate the basic idea.

In order to preserve the consistency of a system, the environment should provide consistency preservation on demand. Thus, temporary inconsistencies are allowed, but the user may call a function for change propagation to make a *depending version* consistent with a *determining version*. In order to provide change propagations along *depend_on* and *based_on relations*, a version must not have more than one outgoing *depend_on* or *based_on relation* to versions of the same document version graph. This means if O_{vs} is defined as the set of outgoing *depend_on* and *based_on edges* of a version v_s,

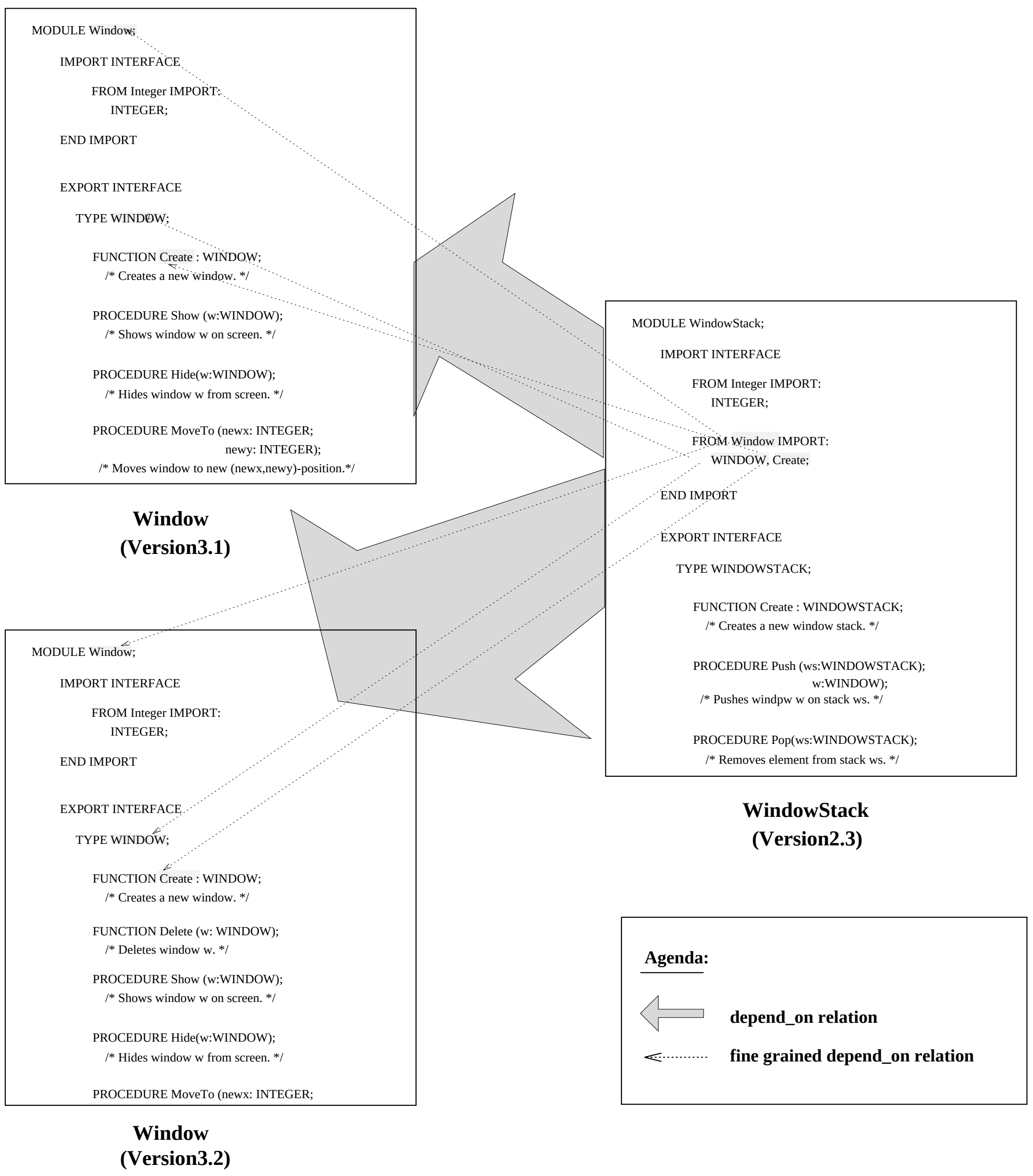
$$\forall v_s \in VER : O_{vs} = \{(v_s, v_t) | v_t \in VER \wedge ((v_s, v_t) \in DEPEND\_ON \vee (v_s, v_t) \in BASED\_ON)\}$$

the following consistency condition has to be fulfilled by each version graph:

$$\begin{aligned} \forall (v_s, v_t) \in O_{vs} : \neg \exists v_{t'} \in VER :& (v_s, v_{t'}) \in O_{vs} \wedge \\ & (((v_t, v_{t'}) \in (SUCCESSOR\_OF)^* \vee (v_{t'}, v_t) \in (SUCCESSOR\_OF)^*) \vee \\ & (\exists v_p \in VER : (v_p, v_t) \in (SUCCESSOR\_OF)^* \wedge \\ & \qquad (v_p, v_{t'}) \in (SUCCESSOR\_OF)^*))) \end{aligned}$$

Note, that $(SUCCESSOR\_OF)^*$ denotes the transitive closure of $SUCCESSOR\_OF$.

The above consistency condition will be illustrated by the following example. Assume that a version



```
MODULE Window;
    IMPORT INTERFACE
        FROM Integer IMPORT:
            INTEGER;
    END IMPORT

    EXPORT INTERFACE
      TYPE WINDOW;

        FUNCTION Create : WINDOW;
          /* Creates a new window. */

        PROCEDURE Show (w:WINDOW);
          /* Shows window w on screen. */

        PROCEDURE Hide(w:WINDOW);
          /* Hides window w from screen. */

        PROCEDURE MoveTo (newx: INTEGER;
                          newy: INTEGER);
         /* Moves window to new (newx,newy)-position.*/
```

```
MODULE WindowStack;
    IMPORT INTERFACE
        FROM Integer IMPORT:
            INTEGER;

        FROM Window IMPORT:
            WINDOW, Create;

    END IMPORT

    EXPORT INTERFACE
      TYPE WINDOWSTACK;

        FUNCTION Create : WINDOWSTACK;
          /* Creates a new window stack. */

        PROCEDURE Push (ws:WINDOWSTACK;
                        w:WINDOW);
         /* Pushes windpw w on stack ws. */

        PROCEDURE Pop(ws:WINDOWSTACK);
          /* Removes element from stack ws. */
```

```
MODULE Window;
    IMPORT INTERFACE
        FROM Integer IMPORT:
            INTEGER;
    END IMPORT

    EXPORT INTERFACE
      TYPE WINDOW;

        FUNCTION Create : WINDOW;
          /* Creates a new window. */

        FUNCTION Delete (w: WINDOW);
          /* Deletes window w. */

        PROCEDURE Show (w:WINDOW);
          /* Shows window w on screen. */

        PROCEDURE Hide(w:WINDOW);
          /* Hides window w from screen. */

        PROCEDURE MoveTo (newx: INTEGER;
```

Figure 5: **Fine-grained inter-document dependencies**

of document *WindowStack* and two versions of document *Window* are given, whereby the versions (3.1 and 3.2) of the latter document are variants.(Variants are versions which are members of different paths from the root node of a document version graph.) The difference between these variants is, that the second variant exports an additional procedure *Delete*. Due to its import interface, version 2.3 of document *WindowStack* depends on a version of document *Window*. If this version of *WindowStack* would be linked to version 3.1 and 3.2 of document *Window* as shown in figure 5 by a *depend_on relation*, this would mean, that version 2.3 of document *WindowStack* is intended to be consistent with both variants of *Window*. If type *WINDOW* changed to *SCROLLBAR-WINDOW* in version 3.1 of *Module Window*, the environment would not know from which variant of *Module Window* changes should be propagated to *WindowStack*, if the user has called the change propagation function. If there is only one outgoing *depend_on* or *based_on relation*, it identifies unambiguously which version of *Window* has to be consistent with *WindowStack*.

If a *consistent_with relation* is defined between two versions, the source version must have an outgoing *depend_on* or *based_on relation* to the document version graph, which contains the target version.

$$\begin{aligned}
&\forall(v_s, v_t) \in CONSISTENT\_with :\\
&(\exists_1 v'_t \in VER : (\ (((v_s, v'_t) \in \mathrm{DEPEND\_ON}) \vee ((v_s, v'_t) \in \mathrm{BASED\_ON}))\wedge\\
&\qquad(((v_t, v'_t) \in (SUCCESSOR\_OF)^*)\vee\\
&\qquad((v'_t, v_t) \in (SUCCESSOR\_OF)^*)\vee\\
&\qquad(\exists v \in VER : (((v, v_t) \in (SUCCESSOR\_OF)^*)\wedge\\
&\qquad\qquad((v_t, v'_t) \in (SUCCESSOR\_OF)^*))))))
\end{aligned}$$

In order to avoid cyclic dependencies between versions each sub-graph of our model, which is spread out by *depend_on* and *based_on relations* has to be acyclic:

$$\forall v \in VER : ((\ v, v) \notin (DEPEND\_ON \cup BASED\_ON)^*)$$

3 VM-Operations

Based on the conceptual model as defined in the previous subsections, the version management system provides the following operations which are the "traditional" operations on versions like e.g. create, derive, freeze and the "new" operations edit, update and merge which exploit the knowledge available by the increment level of the system version graph. We will illustrate the benefits of the exploitation of this knowledge and thus the benefit of our approach by some examples. In order to present an overall view on our approach we will briefly discuss the traditional operations first. Note that we will describe the semantics of these operations based on the graph model as introduced in the previous subsections. A user of our VM system does not necessarily see this graph but rather operates on the level of document versions and its corresponding syntactic structure using the appropriate tool support. This tool support is however outside the scope of this paper which only lays the the foundation for a sophisticated VM system.

Operation create is used to create the initial version of a document version graph and its corresponding root node for the ASG representation on the increment level which are connected by the *has_contents relation*.

Operation create is not used, if a new version is created which is partly generated. In this case the operation generate is used which creates an initial version of a new document version graph and connects it to an already existing version by a *based_on relation*. The abstract syntax graph for the depending version, i.e. that part of the graph which corresponds to the frame, is generated and the *fine grained based_on relations* are established.

The operations create and generate are only used to create root versions of document version graphs. All other versions are created using the derive operation. It creates a new version on the version level and links the new version to the existing one by a *successor_of relation*. On the increment level the ASG representation is copied to the successor version. It is linked to the newly created version node by a *has_contents relation* and *fine grained successor relations* are established between the increments of the new version and the corresponding increments of the predecessor version. Finally, outgoing *depend_on* and *based_on* relations of the predecessor version are copied to the newly created version, which are also copied on the increment level, i.e. the *fine-grained depend_on* and *based_on relations*.

Operation freeze is applied to define a version as stable. Operation derive can only be applied to a version if the version is stable. Note, that the operation freeze cannot be undone.

Operation delete deletes a version on both levels. It can only be applied to versions which are not connected by an ingoing *depend_on* or *based_on relation* to any other version.

As mentioned, the benefits of our approach become clear, if one takes a closer look at the provided edit, update and merge operations.

Merge supports to derive a common successor of two frozen versions within the same document version graph, provided they are members of different paths from the root version. As this operation knows about the contents of both versions based on the increment level representation, it automatically transfers all syntactic constructs which are the same in both versions or which have only been changed in one version, into the new version. If a syntactic construct has been changed in both versions, the operation transfers the construct which fits best in order to preserve the internal consistency of the merged version. Only in those cases, in which a syntactic construct has been changed in both versions and the criterion of consistency preservation does not help to decide from which version the syntactic construct should be transferred, the user is automatically asked for advice. A more detailed description of this algorithm can be found in [Wes91b].

Edit operations, i.e. changes of a version are done based on the ASG representation. This enables to keep track of any inconsistency existing between two versions with respect to the *based_on* or *depend_on* relations. Those inconsistencies can be removed later on (on demand by the user) such that a *consistent_with relation* holds again between the versions concerned.

Basically, update supports to adapt a new version of a depending version to a new version of its determining version. Assume, that the programmer working on version 2 of *WindowStack.c* has almost finished his work, when a new successor of the determining design version *WindowStack* is created. The new version of *WindowStack* has been created, because a new Function *Delete* has been added. In this case, the programmer wants to adapt his version to the new version without loosing the work already done, i.e., he wants the frame of his document to be adapted to the new version, so that the contents of those parts of the frame, which have not changed, is kept. As the update operation knows the determining and depending version, the *fine grained depend_on* or *based_on* relations between those two and the *fine grained successor_of relation* between the determining version and its successor, it can easily perform changes of the depending version, e.g. the generated frames, without a need to touch other parts of the depending document version.

It is worthwhile to note that our approach does, of course, include one serious restriction. Document development tools can only apply the approach, if they are based on an abstract syntax graph representation of documents or versions respectively. However, as we briefly sketch in the next section, modern object oriented database management systems make it easier today to develop such tools. Such that they should become widely available. In any case their sophisticated functionality compared with traditional file-based tools make them highly desirable.

4 Implementation using a Database Management System

As discussed earlier [ESW93], fully object-oriented database management systems (ooDBMS) provide sufficiently powerful data modelling and performance capabilities to store and manipulate documents in terms of ASG-representations. What is additionally needed to implement our VM-system is the possibility to easily create and manipulate versions of those ASGs.
In more detail, we need an ooDBMS which provides versioning of composite objects, i.e. which provide methods to derive, freeze, delete and merge versions of composite objects and a method to create the root version of a composite object. Obviously, an ooDBMS which provides a method derive and the concept of composite objects manages the version level successor relation by itself, i.e. the ooDBMS takes care of efficiently managing marginal different versions of usually large ASGs.

Furthermore, all versions of an object should have the same object-id. Then the *fine grained grained successor_of relation* is derivable from the *successor_of relation* and the fact, that all versions of an object have the same object-identifier. In order to be able to easily introduce fine-grained depend_on and based_on relations, each object which is a member of a composite object must carry an explicit reference.

Operation derives offered by an ooDBMS corresponds directly to the operation derive as explained in section 3, provided that one composite object contains one complete document version graph.

All other operations of our approach have an application specific semantics like e.g. generate which links a preexisting object with a new object or delete which has to check whether ingoing *based_on* or *depend_on relations* exist. They are implemented as a special layer on top of an existing ooDBMS.

To the best of our knowledge, only three existing ooDBMSs currently provide versioning of composite objects. Those systems are O_2, Ontos, and Itasca/Orion. Ontos is not suited for our approach since it does not support merging of variants.

We chose O_2 as the basis for our implementation because its further development is significantly influenced by our own work. Among others, we are in fact partners of O_2 Technologie (which is the vendor of O_2) in an ESPRIT III project called GOODSTEP (General Object-oriented Data Bases for Software Engineering Processes). The goal of GOODSTEP is to develop an SEE platform through extensions of the currently commercially available version of O_2. The project is well underway now. It started in September 92 and will continue until September 95.

O_2 provides the concept of a version unit. This version unit contains the document version graph as is illustrated in figure 6. The only problem with O_2 is that it does not yet support to explicitly reference object versions contained in a composite object. We therefore implemented our approach by introducing two attributes for each relation representing inter-document dependencies. One attribute contains a reference to version of the composite object, which contains a particular object and a second attribute contains a reference to the particular object in that version. Access to a particular object then means the evaluation of two attributes as opposed to the ideal situation where only one attribute has to be evaluated.

5 Related Work

To the best of our knowledge, the VM model in the IPSEN project [Wes91a, Wes89] is the only approach which also exploits the documents' internal structures to improve the power of version operations. In fact, our approach was inspired significantly by the IPSEN project.

One main difference between the IPSEN approach and our approach is the introduction of the *depend_on relation* which IPSEN does not have. Thus, IPSEN does not support to version e.g. architecture definitions in a MIL (cf. figure 2). IPSEN always assumes a master document which cannot be split into parts which are again versioned. This makes it impossible to support multiple users for developing this master document, although such a master document is in fact usually that voluminous that its development requires very much adequate multi-user support.

A second difference results from the fact that we distinguish between *consistent_with relations* and *depend_on relations*. This distinction enables to propagate changes unambigously from one version to other versions without necessarily creating a new version, because each version has at most one outgoing *depend_on relation* (and possibly multiple *consistent_with relations*). If only (multiple) *consistent_with relations* exist, as in IPSEN, a change in one version, which requires a change in another version to preserve consistency between the two of them, always enforces to create a new version. Thus, in IPSEN, even a very small incremental change like the change of an identifier, enforces to create possibly quite a number of new versions in different document version graphs (, if consistency should be preserved).

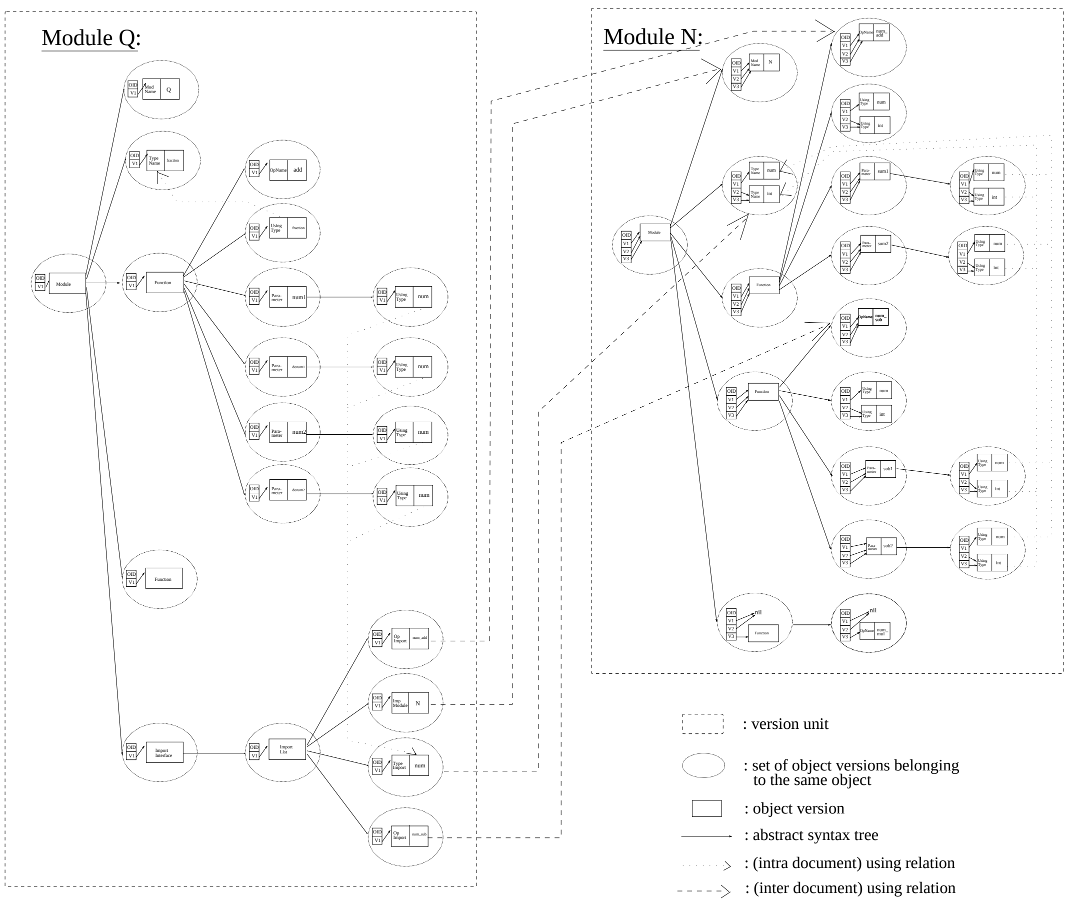

Module Q:
Module N:
: version unit
: set of object versions belonging to the same object
: object version
: abstract syntax tree
: (intra document) using relation
: (inter document) using relation

Figure 6: **References between versioned objects in O_2**

6 Current Work

The implementation as sketched in section 4 is currently underway as part of the GOODSTEP project. This work is related to the development of a tool specification language which includes features to define specific constraints on versions. This language, in general, enables to specify syntax-directed tools and inter-document dependencies which, after automatic translation into executable code, run on top of O_2 [BBD+93].

The CM-model corresponding to the presented VM-model is under development. The main advantage of this model is that it easies very much to build (consistent) configurations. In contrast to CM-models like [Whi91, ML88], a user does not need to define a complete system model upfront, but rather selects only one particular version. Based on the defined *based_on*, *depend_on* and *consistent_with* relations, the environment automatically proposes (all) configurations which are possible based on the selected version.

Finally, we embed the VM/CM model into a process-centered SEE which is geered towards multi-user support. This environment which is called Merlin [PS92, JPSW94], supports users in automatically deriving and displaying so-called user-specific working contexts which show all document versions and all relations between those versions which a user needs to see to perform his/her particular job. A main interesting feature of Merlin is that it automatically updates all working contexts if one user makes a change which concerns others.

References

[BBD+93] W. Beckmann, J. Brunsmann, D. Dong, W. Emmerich, P. Kroha, W. Reimer, S. Sachweh, and W. Schäfer. The Goodstep Tool Specification Language Reference Manual. Deliverable ESPRIT Project GOODSTEP 6115-6P, Commission of the European Communities, November 1993.

[BE87] N. Belkhatir and J. Estublier. Experience with a data base of programs. *SIGPLAN Notices*, 22(1):84–91, December 1987.

[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.

[ESW93] W. Emmerich, W. Schäfer, and J. Welsh. Databases for Software Engineering Environments — The Goal has not yet been attained. In I. Sommerville and M. Paul, editors, *Software Engineering ESEC '93 — Proc. of the 4th European Software Engineering Conference, Garmisch-Partenkirchen, Germany*, volume 717 of *Lecture Notes in Computer Science*, pages 145–162. Springer, 1993.

[Fel88] S.I. Feldman. Evolution of Make. In *Proceedings of the 1st International Workshop on Software Version and Configuration Control, Grassau, FRG*, pages 413–416. Teubner, 1988.

[JPSW94] G. Junkermann, B. Peuschel, W. Schäfer, and S. Wolf. Merlin: Supporting cooperation in software development through a knowledge-based environment. In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors, *Software Process Modelling and Technology*, chapter 5, pages 103–129. John Wiley & Sons Inc., 1994.

[KSW94] N. Kiesel, A. Schürr, and B. Westfechtel. Gras, a graph-oriented database system: Data model, functionality and applications. In R. King, editor, *Workshop on the Intersection Between Databases and Software Engineering, Sorrento, Italy.* IEEE Computer Society Press, 1994. To appear.

[Lew88] C. Lewerentz. Extended Programming in the Large in a Software Development Environment. *ACM SIGSOFT Software Engineering Notes*, 13(5):173–182, 1988. Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Boston, Mass.

[ML88] Axel Mahler and Andreas Lampen. Shape — A Software Configuration Management Tool. In *Proceedings of the 1st International Workshop on Software Version and Configuration Control, Grassau, FRG*, pages 228–243. Teubner, 1988.

[PS92] B. Peuschel and W. Schäfer. Concepts and Implementation of a Rule-based Process Engine. In *Proc. of the 14th Int. Conf. on Software Engineering, Melbourne, Australia*, pages 262–279. IEEE Computer Society Press, 1992.

[Tic82] Walter F. Tichy. Design, Implementation, and Evaluation of a Revision Control System. In *Proceedings of the 6th International Software Process Workshop*, pages 58–67. ACM Press, 1982.

[Wes89] B. Westfechtel. Revision Control in an Integrated Software Development Environment. *ACM SIGSOFT Software Engineering Notes*, 17(7):96–105, 1989.

[Wes91a] Bernhard Westfechtel. *Revisions- und Konsistenzkontrolle in einer integrierten Softwareentwicklungsumgebung.* Springer, 1991. Informatik Fachberichte, 280.

[Wes91b] Bernhard Westfechtel. Structure-oriented Merging of Revisions of Software Documents. *Proceedings of the 3rd International Workshop on Software Configuration Management, Trondheim, Norway*, June 1991.

[Whi91] David Whitgift. *Methods and Tools for Software Configuration Management.* John Wiley & Sons, 1991. (Chapter 7: DSEE).