The constraint diagram: an approach to visualizing the version space

Bjørn Gulla*

Department of Computer Systems and Telematics
Norwegian Institute of Technology
N-7034 Trondheim, Norway

Abstract

The version space consists of a point for each possible version (or configuration) of some data structure. Visualizations of this space can be used to show which versions exist and their interrelationships. Version graphs have traditionally been used to visualize the structure of this space. However this approach has serious limitations, as it is only able to visualize previously stored versions, and not the set of possible versions. Although this property is most annoying for versions groups of composite objects (configurations), the problem also applies for single components (files).

In this paper we propose a novel graphical formalism to communicate interesting features of the version space to ease access to and manage versioned structures. The technique emphasizes potential versions, using intensional characterizations of the version space.

1 Introduction

During the last decades, the importance of configuration management (CM) techniques and procedures has been widely recognized. There is an increasing awareness and use of configuration management tools in many areas, both in academia and industry. Since configuration management tools are gaining popularity and are being introduced in new “territories”, organizations, and to unskilled users, their underlying principles and pedagogical presentation is important.

In this paper we will focus on CM for support of software engineering activities, although the proposed techniques are likely to be applicable in several related areas such as VLSI design, CAD and document processing as well. We will mainly talk about models and possible visualizations of versioning.

As a scenario envisage a software system that due to evolution and varying operational requirements exists in a number of versions and variants, both for the system as a whole and for the individual components comprising the software system.

The problem is then how to conceptualize and visualize the space of potential versions of this software system. In a software system consisting of 1000 modules each in 10 versions, there are $10^{1000}$ different possible configurations. Which of this vast number of configurations are meaningful, and what are their properties?

The problem of making a conceptual “picture” of the versioning structure is generally acknowledged. Lack in current techniques and tools constrain the effective use of versioned products. Developers are not able to exploit the information really available. As a consequence, most CM systems are treated as a passive box just recording the evolution history.

2 Existing approaches

A number of different diagrammatic techniques has been proposed and used to illustrate the versioning structure of components and systems. The visualization techniques have primarily been used to explain the versioning principles used in the models, but lately some has even been implemented in computerized tools. Here we summarize some of the more popular techniques to represent and visualize versions.

- unstructured collection or list

In some of the more primitive data models supporting versioning, the version space has no structure. The versions are treated as unordered members of a set, or enumerated in a list. Visualization techniques such as the one above can be envisaged.

As an example we could mention the Cedar environment [SZH85] which supports immutable ver-
sions of files. There is no structure among the versions, versions are identified just by their timestamps. The Orion object-oriented database system for CAD applications [K+89] supports versioned objects. Unique version numbers are used to identify a version within a version group, but no information about which previous version a version is derived from is represented.

The main problem in these models is of course that there are relations among the versions, relations that prove important and useful in practice. One version might supersede another, two are variants of each other etc. Also, if merges between versions is desired (e.g. to support multiple development paths), comparison with their common ancestor is essential to be able to do intelligent (semi)automatic merge [LCS88].

- version graph

![Version Graph](image)

The version graph is by far the most popular visualization technique to illustrate versions and their interconnections. It tracks the evolution of a component, and alternatives are identified. Different versioning models might restrict the form of the version graph, for example in DAMOKLES [DGL86] it might be restricted to either a linear, tree-like or acyclic graph. The basic relation in all version graph formalisms is the predecessor or revision-of relation, linking each version to the version(s) from which it was derived. In addition to the revision-of relation, explicit variant-of relationships [Tie88] might be illustrated.

There are three problems:

1. The version graph is of limited value when visualizing the versioning structure of composite objects. There is no way to relate version selection in the composite object with version selection done in the components. Symptomatically, a commercial tool [LCS88] only allows version graphs to be displayed for base components, not for systems. Also, version selection in composite systems is usually constrained in some ways. Such knowledge is impossible to state or deduce from version graphs.

2. Another problem is the lack of “semantics” in this kind of graph. There is no indication of what kind of changes are done between versions, or how large portions of the component is affected. This could however be added by annotating the predecessor links, either by labels or by different graphical attributes. The underlying requirement is really a desire to get some indications of the properties of the individual versions.

3. A version graph does not show all possible versions, only previously stored versions are depicted [Cro92]. In order to conserve storage space, the changes between versions (deltas, differences) are usually isolated and stored instead of the full versions. As a side-effect this allows for later flexible composition of changes, enabling construction of a large set of possible versions. Deltas might be deleted, copied, or moved around with respect to their original position in the version graph. Of course, not all of these combinations are meaningful.

- version threads

![Version Threads](image)

To remedy the first of the problems mentioned for the version graph notation, version threads has been used to illustrate version selection for composite objects. In its base form, relations between versions of the top-level object are not shown (see the version numbering scheme in the figure). A version threads diagram visualizes the decomposition structure for versioned composite objects, i.e. how a version is constructed of specific versions of the components. We will name this relation the version decomposition relation.

There are a number of assumptions more or less implicit in this formalism. First, the components are linearly versioned; no variants, temporary fixes, parallel development etc. are assumed. Secondly, all combinations are regarded to be legal or possible, i.e. there are no constraints on combining subcomponent versions. The lack of overview of constraints during the selection process is a serious defect, since these anyhow have to be examined during establishing of the version decomposition relation. The same problems with predicting properties of the resulting configurations applies here as for version graphs.

- multi-level version graph

The last notation discussed here is an amalgam of the version threads and version graph techniques. Both the intra-object version relations (revision-of,
variant-of) and the version decomposition relation are illustrated. As for the version threads technique, it is easy to show how new versions of the composite object can be formed by selecting a novel combination of versions of the sub-objects.

The main problem in connection with multi-level version graphs is information overload. This technique does not scale up at all! In the figure above a two-level graph consisting of a composite with two subcomponents is shown, and yet it is quite complex. In addition, the second and third problem listed under version graphs are also apparent for this formalism.

A sound visualization technique for the version space must provide a global overview of existing and potential version, their interconnections, and their properties.

As demonstrated earlier in this section, none of the conventional visualization formalisms fulfill these requirements. Either they are not able to show all possible versions, or the indicated space of possible versions is too large since constraints are not represented. None of the techniques give any hints about what properties can be expected in a version. We claim that these problems are really caused by weaknesses in underlying versioning model.

Three other observations should also be noted. Common to all these techniques is that they all are extensional with respect to versions: all previous versions are explicitly enumerated in the diagram representations. A second observation is that in a multi-level decomposition structure, versions are named separately at each level and in each part. Thirdly, the decomposition structure (modularization [Par76]) profoundly affects what possible configurations can be constructed by combination.

This allows for reasoning about the properties of configurations, and improved conceptualizations of the version space.

### 3.1 Versioning model

[Tic88] claims that conventional configuration management is modification request (MR) driven. Modification requests are sometimes called change-requests (CR), but we use the term change in a somewhat more limited meaning, a MR is typically realized through a set of changes on the software. Imagine a scenario with a MR-based development or maintenance of a software system. A MR is usually realized through a series of logical edit jobs. The edit jobs are possibly carried through in parallel, potentially by different persons etc. The sum of the edit jobs implements the total required modification. Each edit job usually affects a specific property in the final system.

During an edit job usually several files are updated. For each file or component, a delta or difference produced during the edit job can be computed. The sum of all deltas in all components will be called a functional change, or just a change for short. The deltas in all the files modified will be annotated with the same global name, the name of the functional change.

Versions, of the individual components or of the whole system, can now be formed by naming which changes to include and which to exclude. Configurations are not constructed by selecting versions for each sub-object recursively down the decomposition structure, but by combining changes. This is the basic idea underlying the change-oriented versioning (COV) model [Hol88, LDC+89]. The set of global changes characterizes the total variability of the software system. The basic idea for conceptualizing the version space is rather to consider how these changes are related than considering the associations between the “resulting” versions.

The COV model is particularly advantageous compared to conventional versioning in two areas: first since changes are typically closely related to user observable properties, change-based version selection allows for reasoning about properties of “end” configurations. Secondly, global changes enables uniform version naming for the whole product and configuration construction independent of system modularization.

### 3.2 Constraints

Not all combinations of changes will form valid versions or configurations. This is analogous to the restrictions that might exist between version selections in different sub-objects of composite objects using conventional versioning.

We use the term constraint to denote any kind of restrictions of use of changes when constructing configurations.
Constraints typically limits how different changes can be combined.

So how can we hope that restrictions on change combination can replace both the version graphs and version decomposition information? The version decomposition relation is not longer needed as all object decomposition information is factored out since changes are named globally. Version decomposition is merely a mechanism for mapping version names down the system composition structure. Secondly, versions and their interrelations are now not explicitly represented. Both previous and future possible versions are valuations of the functional changes. Relations among the changes represents the same kind of information as the version graph, but in addition gives a much more accurate specification of valid configurations and their properties.

How is this information about relations between changes gained? During an edit job the developer might discover for example that the modifications assumes the presence of other changes. While implementing functional changes, the programmer build up a knowledge about the software system, and how the current change relates to previous changes. During testing or validation procedures, illegal combinations are discovered. Some constraints might also be stated à priori, as they basically reflect limitations in the “real world”.

Note that the number of constraints do not monotonously increase as the system evolves. As new changes are being carried through, new constraints are also being added to delimit how these new changes can be selected. However during later development, code might be added so that previously illegal combinations are now possible. For example, in a software system running both on SunView and X11 windowing systems, it is decided to add pie-chart graphics support. First this feature is only implemented on the SunView platform, and therefore a constraint is added to disallow selecting simultaneously pie-charts and X11. Later when the pie-chart option is also realized for the X11 windowing system, this constraint can be removed. In this example, separation of logical and physical changes (found in COV but not in the changset model [SMD89]) or change abstraction mechanisms (Section 4.4) are assumed.

4 The form of constraint expressions

We will here briefly present the kind of constraints we have found useful to express between changes in order to support version selection. See also [GKY91].

Basically, if there are no constraints, all combinations of changes are allowed. A constraint restricts this freedom, removing some of the possible combination patterns. In practice constraint expressions tend to be rather short; they are “wide” in application (removing many potential change combinations).

4.1 Implication

The most common form of constraints is that some change assumes the inclusion or exclusion of some other named change(s), or combinations of other changes. We use the implication symbol for denoting such restrictions. In its most simple form, a change might assume the presence of single other change. This kind of constraint typically arises when making a fix to a previous change, or when a change must assume the existence of resources or functionality realized in some previous change. If the change a requires the presence of the change b, we write:

\[ a \Rightarrow b \]

The usual semantics as in propositional logic is assumed. In any legal configuration, either a must not be selected or b must be selected.

As indicated above, a change might just as well preclude the presence of another change in valid configurations. This might reflect known incompatibilities between the changes, name clashes, conflicting use of other entities, or resource competition. The negation symbol is used:

\[ a \Rightarrow \neg b \]

Only exceptionally the selectivity of a change depends exclusively on a single other change. Quite usually the use of a change assumes several inclusions or exclusions of other changes which all must hold, and this is expressed by a conjunction:

\[ a \Rightarrow b \land c \land \neg d \]

In other cases, there are several alternative assumptions, of which at least one must hold, resulting in a disjunction. A change might for example use a resource that must be offered somewhere, and each of the other changes could cause that resource being available.

\[ a \Rightarrow b \lor c \lor d \]

In general, the set of all constraints will form a large conjunction of sub-formulas, where each sub-formula is an implication to a disjunction. Any set of constraints can be transformed into this representation, since this is an equivalent to the conjunctive normal form. The bulk of constraints are typically expressed as quite simple implications.

4.2 Mutually exclusive

Although the previous constructs are sufficient for expressing any set of constraints and are quite convenient
for most use, we have found it advantageous to include an additional connective. Frequently, a set of changes is mutually exclusive, i.e. at most one of the changes in the set can be selected at any time. For example it is not meaningful to select adaptations to several different operating systems simultaneously. For our use in COV we found it beneficial to define the semantics such that none of the changes selected is also allowed (i.e. the base version). In writing we use the \( \odot \) operator:

\[
a \odot b \odot c \odot d
\]

By this type of constraint we are conceptually defining an enumerated type as a dimension of variability. Example: the dimension for user interface selection for a software system could be the range consisting of the values text, curses, x-athena, x-motif, x-openlook and windows. If expressing mutually exclusive sets by means of the more basic connectives, diagram readability and compressibility would be severely reduced.

### 4.3 Incompatible combinations

During testing, reviewing or validation procedures some combinations of changes might be discovered to result in invalid configurations, that is not passing the acceptance test. We use an incompatible combination type of constraint to record such restrictions. Since specific configurations are tested one at a time, such expressions will typically be quite specific, i.e. contain a large number of terms.

\[
\neg (a \land \neg b \land c \land d)
\]

These kinds of constraints can easily be transformed to an equivalent imply expressions such as \( a \land c \land d \Rightarrow b \) (or \( a \land \neg b \land c \land d \Rightarrow \bot \) or \( a \Rightarrow b \lor c \lor \neg d \) etc.).

### 4.4 Abstraction mechanisms

Over the lifetime of a software product, the number of changes increases monotonously. After some time, the number of changes (even logical ones) will be quite large. To be able to conceptualize and navigate in this large space of changes, viewing and abstraction mechanisms might be devised. Viewing, which mainly is a visualization concern, is discussed in Section 5.6.

When working with systems containing a large number of changes, apparent complexity can be reduced by defining abstractions or aggregates. A “high-level” abstract change is defined as a collections of functional changes, possibly via multiple levels of abstractions. This is basically a macro-like expansion mechanism.

\[
\text{current-dec} = \text{port-dec} \land \text{dev2.0} \land \text{bugfix217} \\
\land \neg \text{save-memory}
\]

Note that this name mapping evolves, the user adds or removes terms in this definition at will. Also, there are no restrictions on defining overlapping abstractions, i.e. a functional change might be shared among several abstractions.

Defining abstractions in COV corresponds to deleting versions in conventional systems. Some versions are removed by concatenating changes (not physical delete, but conceptually). Abstractions allow for modularization of the set of changes.

### 4.5 Levels of constraints

In addition to grouping changes by abstraction mechanisms, we have also observed the need for grouping constraint definitions.

Constraints are inherently related to the concept of valid configurations. In different contexts it might be sensible to talk about different sets of valid configurations for the same software system. For example when making a customer release, changes cannot be so freely combined as when a developer makes his or her private test configuration. When establishing the customer release, additional constraints are typically included to ensure that the selected change combination has reached a certain level of approval.

There are two main uses:

- **confidence level**

  As mentioned above, many constraints are entered in connection with testing and validation procedures. Different combinations of changes have reached different levels of acceptance: passed compilation, passed module testing, passed integration testing, passed pilot application, passed customer approval etc. If requiring a configuration at a specific confidence level, change combination not approved for that level must not be allowed.

- **project or user-specific sets**

  A project or a specific user might define a library of constraints to remove unwanted variability.

The constraints might be grouped into some kind of hierarchy. Basic constraints that must never be violated are located at the bottom. Additional constraints further delimiting valid combinations are located higher up in the hierarchy. When entering a constraint, the developer also assigns it to the correct level.

Also note the evolution of constraint expressions as the software system matures. Constraints might be classified along a static-dynamic scale according to how viable they are to change as the software system evolves. Constraints in the base levels tend to be more static, expressing properties of the “real world”, company policies
etc. In the higher levels there are more fluctuations, constraints appear and vanish as the software evolves and testing/approvals progress.

5 Visual formalism

To facilitate building conceptual models of and reasoning about the version space when using COV, we have developed a visualization technique for showing the constraint structure between functional changes.

Graphical symbols and connection illustrate how the constraints restrict change combination to specify configurations. This graphical formalism enables the user to form intuitive mental models of the domain of changes and get a global overview of the space of potential versions. Also, a developer can make hypothesis and reason about the feasibility change combinations and properties of the resulting configuration.

Note the principal difference between e.g. the version graph and the constraint diagrams we propose here: version graphs represent the extension of the version space for a component, while the constraint diagram formalism presented here represent the intension [Sow84]. All potential version and their interrelationships are not visualized, but rather a scheme for constructing and characterizing versions.

5.1 Symbol for functional change

The base symbol in a constraint diagram is the symbol for a functional change. This is displayed as a rectangular box with a label containing the change name. There is at most one occurrence of each functional change in a diagram.

If no edges are attached to a node symbol, the corresponding change can be freely combined. If there are no constraints at all, the diagram will consist of n nodes with no edges between (n is the number of functional changes), and all 2^n possible combinations are legal.

Changes might be categorized into different groups, cf. types of maintenance activities [LS80]: adaptive, corrective, perfective. We have considered using a scheme based on the categories: porting (platform changes), corrections, enhancements, perfections, and restructuring. Such schemes could be used to augment the base formalism, by using different graphical attributes (e.g. different icons, shapes, colors, sizes, line types, fill patterns) when displaying nodes corresponding to changes of different categories. However, there is a danger of overloading the diagrams.

5.2 Implication symbol

As noted in Section 4.1, a simple implication is just a special case of imply disjunction construct. However it occurs so often that we have decided to have a simple way of representation. We use an edge connecting the two participating changes, with the arrow end at the "implied" change. See Figure 1.

![Figure 1: Simple implication: bugfix30 ⇒ bugfix27](image)

When conjunctions occur as the right-hand side of an implication, there is no need for a special symbol since a ⇒ b ∧ c ⇔ a ⇒ b ∧ a ⇒ c is a tautology. Such conjunctions can always be decomposed and visualized using the base arrow notation, see Figure 2.

![Figure 2: Implication merge2.0 ⇒ dev2.0 ∧ temp-fix](image)

5.3 Negation symbol

Negations in formulas are indicated by a circle ◯ at the endpoint of an arrow, see Figure 3. A negation symbol might appear at any end of an imply symbol.

![Figure 3: Implication with negation: myfix ⇒ ¬ wrongfix](image)

Note that the arrow representing the constraint in Figure 3 could be replaced by an arrow in the opposite direction (wrongfix ⇒ ¬ myfix). Negation indications at each end of an arrow must be inverted when reversing an arrow since a ⇒ b is equivalent to ¬ b ⇒ ¬ a. See Section 6 on how this freedom might be exploited when generating diagrams. In any one diagram, only one of these equivalent arrows is drawn to avoid unnecessary complex diagrams.
5.4 Disjunction symbol

For visualizing constraints of the type \( a \Rightarrow b \lor c \), a special node with the form of a diamond \( \Diamond \) is used. An edge from the left-hand symbol to the diamond and edges from the diamond to each of the right-hand symbols are drawn, see Figure 4.

![Diagram](image)

Figure 4: Implication with disjunction: \( X11 \Rightarrow \text{Dec} \lor \text{Sun3} \)

Reversing arrows is also possible here. Figure 5 shows some possible permutations of the constraint in Figure 4 (\( X11 \Rightarrow \text{Dec} \lor \text{Sun3} \iff T \Rightarrow \text{Dec} \lor \text{Sun3} \lor \neg X11 \iff X11 \land \neg \text{Sun3} \Rightarrow \text{Dec} \iff \neg \text{Dec} \land \neg \text{Sun3} \land X11 \Rightarrow \bot \)). Note however the interpretation for valid valuations: when there are several arrows to a diamond node, there is an AND between them (and implicitly true if none). If there are no arrows from a diamond node, false (a contradiction) is implied.

5.5 Mutually exclusive

For this connective we have investigated several possible graphical notations. We first tried to allude to similarity with a multi-valued change, without finding any good metaphor which also had the correct intuitive meaning when combined with other constraints.

Relatively comprehensible diagrams are achieved when all symbols for mutually exclusive changes are drawn at the same rank (above each other) and connected by special kind of edges. Figure 6 shows two possibilities for visualizing a relationship between mutually exclusive changes. We have decided to recommend the (a) variant, since diagrams using the (b) variant tend to be less readable.

![Diagram](image)

Figure 6: Two possible graphical notations for \( \text{unix} \otimes \text{vms} \otimes \text{os2} \otimes \text{domain} \)
in a separate window or expand in place. [NPT90] discusses layout of what she calls graph abstractions in a graph editor framework, with different degrees of expansion in place. To facilitate easy zoom into and out of abstract changes, abstraction is indicate with a clickable box \( \square \) in the upper right corner, see Figure 7.

5.7 Discussion

The constraint diagram is really one possible solution on the problem on how to best visualize a large arbitrary sentence expressed in propositional logic, given the probabilities of the different types of constraints. We hope to have found an intuitive graphical formalism that facilitates comprehension and ease of orientation.

For small subsets of the total set of functional changes, the subgraph of the constraint diagram might look like (a part of) a version graph if there exist strong dependencies between the selection of changes. The constraint diagram emphasizes freedom of combination. If individual changes or parts of the diagram are disconnected, the changes can be arbitrarily combined.

Constraint diagrams typically grow in the right-hand direction as changes are added (dependent changes), and downwards as portability or orthogonality of changes is improved (independent changes).

5.8 An example

In Figure 8 a view of a constraint diagram is displayed. The constraints between abstract porting changes are depicted. Two levels of constraints are shown in the same diagram. The base constraints are shown using

![Diagram](image)
solid lines (and thick dotted lines for mutually exclusive sets), while the additional constraints indicating the “module-tested” level are drawn using dotted lines.

6 Layout of constraint diagrams

The constraint diagram notation is well suited for automatic layout computations. Manual layout and coordinate recording is not opportune when supporting abstractions, constraints grouped in levels and viewing. We will here just briefly indicate how the layout is computed.

The base graph (ignoring mutually exclusive relations and abstractions for the time being) is a hierarchical graph [ET89]. By customizing the first of the three phases in an ordinary hierarchical graph layout algorithm [STT81], we are able to handle the special layout requirements for mutually exclusive sets and remove cycles. Abstraction can be handled as described in [NPT90].

A convenient property of the constraint diagrams is the large degree of layout freedom allowed by manipulation of the implication constraints. Using known equivalences from propositional logic, the form of the graph can be altered and readability significantly improved without modifying its meaning. As a coarse-grained heuristic, we have found it advantageous to generally place “older” changes to the left of “newer” ones, and preferably use single left-hand sides in implications. However, such restructuring must be seen in connection with the overall diagram layout and should to some extent be controlled by personal preferences.

7 Use of constraint diagrams

There are three main areas of application of constraint diagrams:

1. Aid version selection

Selecting a unique version is determining a valuation for all change symbols, that is to decide whether to include or exclude each of the functional changes. The main purpose of the constraint diagram is to support finding consistent selections, i.e.
a valuation of the changes that satisfies all defined constraints (at a certain level).

2. Aid comprehension
By interactive navigation and manipulation of the constraint diagram some mental picture of the the changes and their interconnections is gained. Thereby also an understanding of what possible versions can be constructed, their properties and their differences is built up.

3. Aid definition of new constraints
When a set of constraints has already been defined, adding new constraints can be an error-prone and dangerous operation. With no machine support, addition of constraints contradicting previous constraints is easily done.

The main operation when investigating a constraint diagram is determining valuations of change symbols. Using the mouse, the user can point-and-click on the change symbols and thereby toggle between the states of the change. A change basically has three possible values: included, excluded, or unset (not yet decided). Some possible graphical annotations to attach change symbols to indicate the current change state is shown in Figure 9. When updating the state of one change, this is propagated throughout the constraint diagram so that no contradictions exist. When there are several possibilities, the heuristics try to keep the most recent user selections. In addition it is possible to choose which levels of constraint to consider, set views and navigate in the hierarchy of abstractions.
8 Related work

The change-oriented versioning model resembles and generalizes conditional compilation schemes [WS88]. Our bottom-level universal constraints corresponds to restrictions between use of conditional compilation variables. In addition to such static constraints, we are also able in the same framework to add evolution-dependent constraints, i.e. reflecting the current state of the implementation.

The internal operation of SCCS [Roc75] reveals some interesting similarities with COV. When producing a version, the delta table (a linearization of the version graph) is read to find which deltas (physical changes) to include or exclude. The user might also explicitly specify other deltas to include, exclude or ignore. See [Gla78] for a discussion of how users intuitively tried to use this mechanism, and the fate of optional deltas. One of the main problems in SCCS is the lack of a global delta naming scheme.

The ADC changset model of versioning [SMD90] is similar to our basic model, but ADC just as SCCS lacks the separation between logical and physical changes. However, changesets (resembling our functional changes) are globally named and applicable. The version selection model supported in ADC is basically only what is called abstraction in the above presentation.

Our mutually exclusive connective might be related to dimension expression supported in the P-EDIT multi-version text editor [Kru84]. Dimension expressions using comparison operators and generalized integer literals control inclusion or exclusion of text fragments. As for the underlying versioning model, this is very similar to COV.

In the Adele system [BE86] for configuration management of software systems, implementations might have a set of predicates constraining version selections for the modules they depend on. The version selection process consists of an intertwined product and version space traversal.

We have not been able to locate any references to similar work on visualizing large sets of constraints. Surely there must exist some work on graphical notations for propositional sentences. There exists a graphical notation after C. S. Peirce called existential graphs [Rob73] which is a kind of area graph with the full power of first-order logic (can also represent higher-order logic). It was not found suited for our use.

9 Conclusion

Deltas were originally conceived as a method to conserve space. However, as hopefully this article has conveyed, delta storage allows for a whole new flexibility by allowing novel versions to be formed as combinations of changes. This is somewhat similar to the effect of modularization when combined with conventional versioning. A partitioning which was originally introduced to circumvent different machine restrictions or aid comprehension, allows as a side-effect building novel system configuration by creative sub-object version selection.

COV really poses a shift in scope and motivation for use of versioning. While many conventional systems might be considered an on-line (instead of tape), economic backup tool with undo operation, the COV view is emphasizes versioning as a mechanism to support flexible, creative, property based system composition.

The proposed solutions for managing changes and visualizing the constraint structure is an attempt at unleashing the power of versioning to the software developers and maintainers. As identified in the Introduction, the lack of proper conceptual models and visualization techniques is a serious drawback that limits the use and usefulness of current tools. This is a first proposal that will probably need refinements and validation in an industrial environment.

10 Acknowledgments

We wish to thank Reidar Conradi for valuable comments and discussions.

References


