# *COMAND* – A Distributed Configuration Management Framework

Victor Volle[*], Ingrid Fischer
University of Erlangen-Nürnberg
Martensstr. 3
91058 Erlangen, Germany
vrvolle, idfische @informatik.uni-erlangen.de

Detlef Kips
Basys GmbH
Am Weichselgarten 4
91058 Erlangen, Germany
kips@basys-gmbh.de

Manuel Koch
Technical University of Berlin
Franklinstr. 28/29
10587 Berlin, Germany
carr@cs.TU-Berlin.de

## Abstract

*Software development is becoming a more and more distributed process, which urgently needs supporting tools in the field of configuration management, software process/workflow management, communication and problem tracking. In this paper we present a new distributed software configuration management framework* COMAND*. It offers high availability through replication and a mechanism to easily change and adapt the project structure to new business needs. To better understand and formally prove some properties of* COMAND*, we have modeled it in a formal technique based on distributed graph transformations. This formalism provides an intuitive rule-based description technique mainly for the dynamic behavior of the system on an abstract level. We use it here to model the replication subsystem.*

## 1. Introduction

Developing software is not an easily planable process.

Not only that software projects are much bigger our days than they were before but stricter quality requirements make it even more difficult to plan and to coordinate a project.

The result of a development process often has not much in common with the original concept. Additionally two third of the complete amount of time and money are spent for maintenance and further developments after the first project was completed.

Furthermore the knowledge to complete a big project is hardly available in a single company. Hence, external project partners must contribute. More often it is also economically sensible to pass work to external subcontractors. One possible scenario is to have just a core of developers located in the main company. Additional staff is situated around the world wherever the knowledge needed is available for a reasonable price. This is known under the catchword *virtual company*: a company that just exists for a single project.

With an ever growing demand on Distributed Software Development one of the results are communication, coordination and quality management, as all project sites must have access to a consistent up to date set of project documents. When one project site changes a concept, it must become known to all other project sites, too. This is especially a problem when there is no central archive available for all project partners.

In this paper we are describing the architecture of the distributed software configuration management (SCM) framework *COMAND*. To ensure the consistency of our SCM framework, in particular the replication mechanism, we have specified the core of *COMAND* by distributed graph transformations [14], which gave us important insights into the intricacies of such a complex system.

### 1.1. Design goals

It is – or better: should be – common knowledge that the main problem of software configuration management is not of technical nature, but a question of human interaction or "People Ware" [9, 4]. Introducing SCM into an organization has to overcome some major obstacles. Every developer has to be convinced to constantly and correctly use the SCM system. You may call it sluggishness or experience: The average developer considers SCM as yet another bureaucratic plot of management.

Therefore the SCM system must be as easy to use as possible and the underlying model must be intuitive. The alternative may be to use a big SCM system like ClearCase [10] which hides every detail of configuration management from the developers. In this case, however, there must be someone who does nothing but administering and maintaining the SCM system. This is not a feasible option for most

*Victor Volle's work is partly supported by a research grant from the German Ministry for Culture and Science (BMBW) under contract number 0004402D7A

projects which are under constant pressure to finish their work under a tight budget.

Since distributed projects are taking place in rather heterogeneous environments, it is important to have the SCM system running on as many platforms as possible. To achieve this we choose to implement the complete framework[1] in Java™. This decision allows us to offer SCM services even over the World Wide Web.

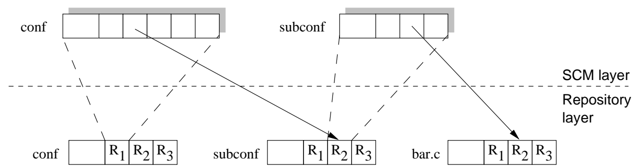


**Figure 1. Repository and SCM layer**

## 2. Architecture

In the following we describe a give an overview of the architecture of *COMAND*. A more complete description can be found in [20, 15].

### 2.1. Configurations are Revisions, too

Selecting a revision of a document from a repository has been rather straightforward since the days of SCCS or RCS: You could ask for a revision EITHER with a given revision number or with a label. Each has to be unique with regard to all other revisions of this document. But when it comes to selecting a consistent set of revisions you are almost left on your own. Revisions which belong together have to be explicitly labeled with the same label. If the responsible developer forgets to do so, the revisions are left floating in the repository with no context information at all. Heidenreich [6] proposed an automatic labeling based on the dependencies of a revision. A revision was labeled with a vector containing the revision numbers of all revisions it depended on. Therefore you could request a revision from the archive and getting all revisions belonging to it with a single request. No explicit labeling or other overhead was necessary.

But since the dependencies were not always available and this labeling proved to be overkill in many situations, we made one step further. In Heidenreich's approach a *configuration* was a vector containing the element-wise maximum of all vectors of all revisions belonging together at a given time. In [19] it was proposed to write this *configuration vector* into a file and to archive it in the repository. Thereby configurations have become revisions, too. You could now request a configuration from the repository and would get all revisions which belong to this configuration.

Configurations themselves may again be identified by revision numbers or labels. Since configurations are revisions, the notion of *subconfigurations* can be introduced rather easily into our model.

In figure 1 the layered architecture of *COMAND* is shown. In the repository layer, configurations are nothing but revisions. But if a revision containing a configuration is retrieved from the repository (checked out), the SCM layer knows how to get all revisions contained in this configuration.

[1] A minimal problem tracking tool has been completed and the repository will be completed soon.

### 2.2. Supporting change

When a configuration is checked out, a user *workspace* is established, containing all revisions of the configuration as *work revisions*. Normally the directory structure of the project is recreated, too. A configuration is generally identified with a directory, and a subconfiguration is (normally) identified with a directory below the configuration directory. Because this structure is archived, too, the structure of the complete project under SCM control may be changed completely. The SCM system knows how to find older revisions even if a configuration has been split in two, subconfigurations have been moved into another (sub-) configuration etc. To achieve this, we needed to assign a unique ID to every document and an ID to each revision which is unique with regard to the document to which it belongs. Even renaming a document is easily supported by this approach.

Imagine a user has moved a file (working revision) from one directory in the workspace into another. When he or she now tries to check in the configuration, the SCM system has to decide whether this file has to become the first revision of a new document or (as is the case in our scenario) a new revision of an existing document. The repository server first tries to find a document with this name in the archive, if it can't be found it offers a dialogue for the user where he or she can decide on how to continue. To further ease the use, you can move, rename etc. the files in your workspace from within the GUI. While doing so, the association of working revision with the document and revision ID is never lost.

Our model – in the terminology of Conradi and Westfechtel [3] – belongs to the category of "version first" instead of "product first" selection of configurations. That is you have to select a specific version of your project, and the structure of the project is determined by the selected version.

## 3. Replication in *COMAND*

The replication of revisions and configurations takes place in the repository layer. The SCM system only uses the services provided by the repository and is – generally – not aware of the replication. Since the repository handles configurations like any other document the complexity of the system is therefore greatly reduced.

To support the replication mechanism the unique document and revision IDs have to be enhanced. Every repository is assigned a site number. The document and revision IDs now consist of a site number, where the document (resp. revision) has been created and a serial number: The serial number for documents is maintained separately for each repository (s. fig. 5). The serial number for revisions is maintained by the revision container which is part of the repository.

Additionally, every change in the repository is "timestamped" with a serial number [1, 6]. Each change is written into a log file, which is also used to support recovery after a system failure.

Since we felt the need to thoroughly analyze the replication mechanism, we chose a formal modeling technique based on graph transformations [12], which has been enhanced by [14] to support the modeling of distributed systems.

### 3.1. Distributed Graph Transformations (DGT)

DGT can be seen as hierarchical non distributed graph transformation (GT) on two levels. Therefore, we first review the basic concepts of GT, that are *labeled graphs* and *transformation rules*. A labeled graph consists of a set of edges $G_E$, a set of nodes $G_V$, and two mappings $s_G, t_G : G_E \to G_V$ specifying source and target node for each edge. Nodes as well as edges may be labeled by elements of different label sets, for instance real numbers or strings. In figure 2 some sample labeled graphs are shown.

A graph can be modified by transformation rules as shown in the upper half of figure 2. It consists of two *graph morphisms* $l : K \to L, r : K \to R$, where graph $L$ is called left hand side, $K$ gluing graph, and $R$ right hand side of the rule. A graph morphism $f = (f_E, f_V)$ between two graphs $G$ and $H$ consists of two mappings $f_E$ between the edges and $f_V$ between nodes. A rule can be additionally furnished with a set of morphisms $A_G = \{a_i : L \to A_i | 1 \leq i \leq n\}$, called *graphical conditions* and a set $A_L$ of boolean expressions, called *label conditions*. In figure 2 the rule possess one graphical ($a : L \to A$) and one label condition ($x < y$).

In the sample rule in figure 2 nodes are labeled with numbers (accurately: elements from the term algebra for integers), where $x$ and $y$ are variables. When applying a rule to a graph $G$ first an embedding $m$ of the left hand side $L$

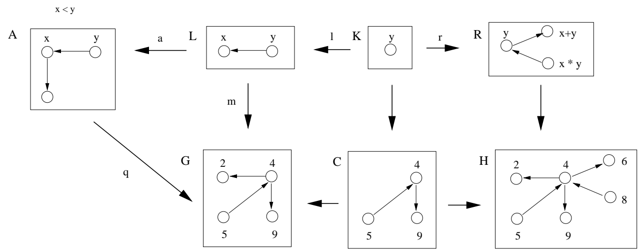


**Figure 2. A non distributed transformation.**

into $G$ must be sought. By the embedding the values of the variables $x, y$ are replaced by the values $2, 4$. Next the label conditions have to be checked. The embedding satisfies a label condition, if its evaluation is true under the variable assignment of $m$. In figure 2 $x < y$ denotes that the value substituted for $x$ must be smaller than the value substituted for $y$, which is true in our example.

Then the graphical conditions $a_i$ have to be checked. The embedding $m$ satisfies $a_i$ if there is *no* embedding $q : A_i \to G$ such that $q \cdot a_i = m$, i.e. $a_i$ specifies forbidden graphical structures. In figure 2 there is a graphical condition forbidding an edge outgoing from the node labeled $x$. In our example the embedding satisfies the graphical condition. In the following these negative graphical conditions will be written with $\neg\exists A_i$.

If all label as well as all graphical conditions are satisfied by $m$, the rule can be applied. Otherwise a new embedding must be sought, if possible. We apply the rule by deleting the left hand side $L$ from $G$ except $K$ which contains nodes and edges that are necessary to insert $R$. The result is the context graph $C$. Theoretically $C$ is constructed as $G - m(L - l(K))$. Then $R$ is inserted, the result $H$ is calculated as $C + (R - r(K))$. The new labels of $H$ are obtained by evaluating the expressions given in $R$ under the variable assignment given by $m$. In figure 2 the nodes in $H$ are now labeled with the result of addition and multiplication.

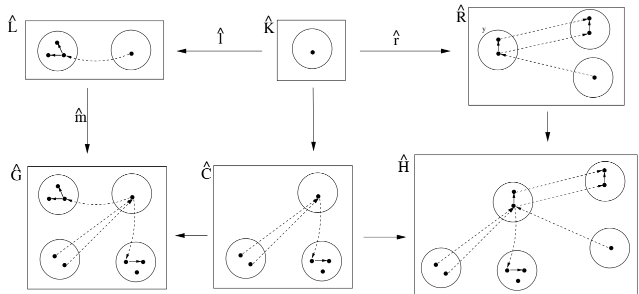


**Figure 3. A distributed transformation**

We show next how GT can be used for describing distributed systems. We are especially interested in the modeling of the system's topology , the local states and their relations to other local systems, and dynamic changes of both topology and local states including their relations. As the topology is often depicted in a graph-like fashion, it is natural to describe the topology by a graph. Just as for the topology graphs are been used for describing local states whereas relations between local states are expressed by graph morphisms. Presently both network and local states are modeled by graphs but they are still separated. In order to combine both the network graph and all local graphs a two-level hierarchic graph, called *distributed graph*, is introduced. The two levels are: (1) A *network graph* describing the topology of the system and (2) To each node in the network graph is assigned a graph representing its local state and to each edge in the network graph is assigned a graph morphism representing a relation between local graphs. We denote a distributed graph by $\hat{G}$, its corresponding network graph by $Net(\hat{G})$, and the local graph that is assigned to network node $i \in Net(\hat{G})_V$ by $\hat{G}_i$. Sample distributed graphs are shown in figure 3. Here local graphs are drawn inside the network nodes and local graph morphisms are drawn by dashed arrows. Network edges are implicitly given by local graph morphisms.

A distributed graph morphism $\hat{f} = \langle f, (f_i)_{i \in G_V} \rangle$ between two distributed graphs $\hat{G}$ and $\hat{H}$ consists of a graph morphism $f : Net(\hat{G}) \rightarrow Net(\hat{H})$ between the network graphs and a set of graph morphisms $f_i : \hat{G}_i \rightarrow \hat{H}_{f(i)}$ for all nodes $i$ in $Net(\hat{G})_V$.

Actions on the distributed system are described by *distributed transformation rules* consisting of two *distributed graph morphisms* $\hat{l} : \hat{K} \rightarrow \hat{L}, \hat{r} : \hat{K} \rightarrow \hat{R}$ where a sample is shown in the upper half of figure 3. In fact, a distributed rule consists of a non distributed rule for the network level and a set of non distributed rules for each node $i$ in the network graph $Net(\hat{K})$. Because of that an application of a distributed rule consists of the application of all these non distributed rules, i.e. network rule and local rules. We have to mention, however, that some additional conditions for the embedding $\hat{m}$ has to be satisfied in order to guarantee this component-wise application. These conditions are explained in more detail in [14, 15]. Moreover, distributed rules can be furnished with application conditions as in the non distributed case.

## 3.2. Modeling replication with distributed graph transformations

In the following we explain the formal specification of the replication mechanism. The complete specification of *COMAND* by DGT can be found in [15]. A more informal description is in [20]

## 3.3. The network graph

Repositories at different sites are modeled with nodes in the network graph. Each such node has an interface node, which is necessary for the underlying formal method, but is also used to decouple communication in the implementation. Each site receives information from other distinguished sites and passes its information to various other sites. So the edges of the network graph indicate the *replication direction* of the information. In figure 4 such a network graph is given consisting of three sites and its interfaces (depicted with a dashed line).

Each node of this network graph contains another graph representing the project structure local to this site – as described in chapter 2.2.

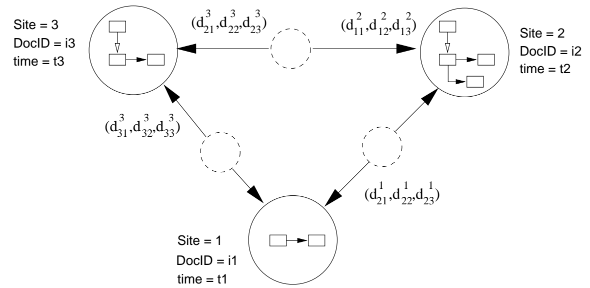


**Figure 4. The network graph**

The sites are labeled with its name (a number), its local time and a document counter. When a new document is created this counter is used to build the unique document ID.

The edges denote the replication direction. They are labeled with tuples containing the information about the assumed knowledge of the other site. E. g., the edge between Site 1 and Site 2 is labeled with the tuple $(d^1_{21}, d^1_{22}, d^1_{23})$. The tuple contains information about what site 1 (indicated by the superscript) thinks, which changes are known to the other site. Remember that each change in a repository is time stamped with a serial number. The tuple contains the highest time stamp which is known to the other site – as far as site 1 knows.

$d^1_{21}$ therefore contains the last change (time stamp) of site 1 that has been replicated to site 2 – as far as site 1 knows. The real actual state of site 2 is not known to site 1. Site 1 may have replicated some changes to site 2 which has replicated them to site 3. Site 1 has no way of knowing about this replication. Only when a replication from site 2 reaches site 1, This site also gets to know the actual state of site 2. More general $d^x_{yz}$ is what site $x$ thinks about the knowledge of site $y$ concerning the change with the highest timestamp from site $y$ that has been replicated to site $z$.

### 3.4. Rules for Replication

Whenever it seems to be necessary the content of one site can be replicated to another. This can be done at a fixed time, e.g. midnight every day, every 15 minutes, etc. or it can be explicitly started. To replicate a revision in the source site to a target site via a replication edge, the site and time label of the revision to replicate is compared with the corresponding entry of the replication vector the edge is labeled with. During that replication the vector will be updated.

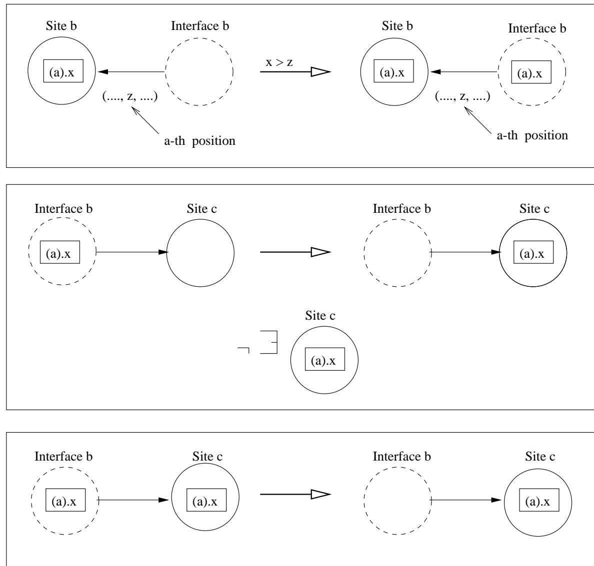


**Figure 5. Replication between two sites**

The replication is divided into four parts. First the revision to be replicated has to be sent to the target site. In the second step the target site has to accept or refuse the replicated revision (Fig. 5). The third and fourth step do the replication of all edges between the replicated revisions (Fig 6). Assume we want to replicate from site b to site c. We first have to check whether a revision with *(site).time = (a).x* has to be replicated. As described above we must take a look at the corresponding replication vector at position $d^b_{ca}$. If $x > d^b_{ca}$ we know that the revision was created after the last replication and must therefore be replicated. This replication is done by writing the revision into the interface of the replicating site, which is shown in the first rule. The graphs to the left and right of the arrow with the hollow head correspond to the graphs marked $\hat{L}$ and $\hat{R}$ in fig 3. A graph below the arrow depicts an *application condition*. A rule with an application condition can only be executed if this condition is matched.

The second and third rule in figure 5 carry out the second step: importing the new revisions. The second rule imports those revisions which have not been imported yet. The application condition to the right ensures the correct application of the rule. The third rule rejects those revisions while cleaning up the interface.

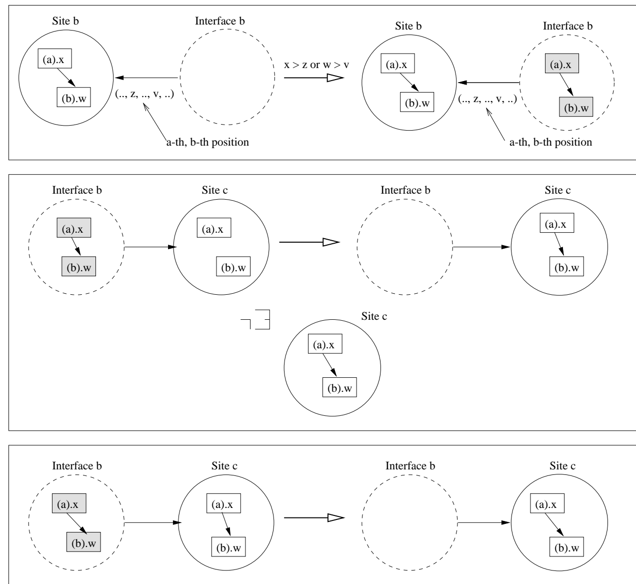


**Figure 6. Replicating the edges**

After all revisions have been replicated the edges get their turn (s. fig. 6). Since we can not replicate bare edges, the nodes connecting these edges are replicated as *shallow copies*, i. e., only the document ID and the revision ID of these revisions are replicated. The importing site uses this information to correctly insert the edges. Again all edges which have been replicated before are rejected and the interface is cleared.

The only problem remaining is that it is not ensured for a configuration to be complete. How can we ensure that all revisions contained in one configuration are really replicated? Therefore a lock is set on the source site of the replication. While this lock is set nothing else than a replication is running. When everything has been replicated, which can be tested with the help of the site and time stamps of the revisions and the replication vector of the corresponding edge, the site is unlocked again and the replication vector is updated. A similar procedure has to be carried out on the target site: It must also be ensured that every update is included in the repository so that configurations are complete before a check out starts. Therefore a message is sent from the source site (a special node that is written into the interface), that a replication starts. When the target site receives this message it locks itself and starts to handle the replication. So the rules given in figure 6 can only be applied if the replication lock is set. When the source site of the replication

finishes locking, it also sends a message to the target indicating that the replication is finished. The target can unlock when this message arrived and all other messages have been consumed.

## 4. Related Work

Nowadays Distributed Software Configuration Management (DSCM) is becoming a widely discussed topic. The first commercial tool offering DSCM was ClearCase [1]. Based on its own NFS-compatible file system, ClearCase provides a similar replication mechanism. But ClearCase doesn't fit into smaller projects working at a fast (or frantic) pace to deliver the next version. ClearCase is better suited for business critical, medical or military projects, where the administrative overhead doesn't matter that much.

Hoek et. al. [18] also propose a DSCM system based on their own file system. Since they do not replicate all revisions, they do not offer the high availability [7] we have in mind. Whenever the remote site is not available, a user of Hoek's system can not get the requested revision. *COMAND* can fall back to FTP or even transporting floppy disks. This proved to be necessary, because of security considerations and network problems.

The work of Hunt et. al. enhancing Tichy's RCS [16] for the World Wide Web [8] has the same direction as ours, offering SCM services in an intuitive, platform independent way. However RCE is still file oriented, whereas *COMAND* is based on the concept of whole configurations.

## 5. Conclusion

The field of distributed software configuration management has not yet been consolidated. New research directions have to be taken. Thoroughly specifying such a system with distributed graph transformations may be a fruitful direction. Even if we think the formalism is not (yet) mature enough for every day use.

On the other hand there is an urgent need for flexible, easy to use and powerful distributed configuration management systems for small to middle-sized teams. Therefore we are trying to keep *COMAND* as simple as possible – but not simpler.

Besides the insights given by DGT, we hope to prove the correctness of the model. But some further theoretical investigations are still necessary. Anyhow, the modeling process helped in identifying problems even before the design phase, which should lead to a more reliable implementation.

## References


[1] L. Allen, G. Fernandez, K. Kane, D. Leblang, D. Minard, and J. Posner. Clearcase multisite: Supporting geographically-distributed software development. In *[5]*, pages 194–214, 1995.

[2] R. Conradi, editor. *Proceedings of the 7th Int. Workshop on Software Configuration Management*, volume 1235 of *LNCS*, Berlin u. a., 1997. Springer.

[3] R. Conradi and B. Westfechtel. Version models for software configuration management. Technical Report AIB 96-10, RWTH Aachen, Lehrstuhl für Informatik III, RWTH Aachen, D-52056 Aachen, 1996.

[4] S. Dart. Best Practice for a CM Solution. In *[13]*, pages 239–255, 1996.

[5] J. Estublier, editor. *Proceedings of the 4th and 5th Int. Workshop on Software Configuration Management*, volume 1005 of *LNCS*, Berlin u. a., 1995. Springer.

[6] G. Heidenreich, M. Minas, and D. Kips. *A new approach to consistency control in software engineering*. In *[11]*, pages 289–297, 1996.

[7] A. A. Helal, A. A. Heddaya, and B. B. Bhargava. *Replication Techniques in distributed systems*. Kluwer academic publishers, Boston, 1996.

[8] J. J. Hunt, F. Lamers, J. Reuter, and W. F. Tichy. Distributed configuration management via java and the world wide web. In *[2]*, pages 161–174, 1997.

[9] S. Kolvik. Introducing configuration management in an organisation. In *[13]*, pages 220–230, 1996.

[10] D. B. Leblang. The cm challenge: Configuration management that works. In *[17]*, pages 1–38. 1994.

[11] D. Rombach, editor. *Proc. 18th International Conference on Software Engineering*, Los Alamitos, CA, 1996. IEEE.

[12] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations. Vol. I: Foundations*. World Scientific, 1997.

[13] I. Sommerville, editor. *Proceedings of the 6th Int. Workshop on Software Configuration Management*, volume 1167 of *LNCS*, Berlin u. a., 1996. Springer.

[14] G. Taentzer. *Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems*. PhD thesis, TU Berlin, 1996. Shaker Verlag.

[15] G. Taentzer, I. Fischer, M. Koch, and K. V. Volle. Distributed Graph Transformation with Application to Visual Design of Distributed Systems. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformations. Vol. III: Concurrency and Distribution*. World Scientific, 1998. to appear.

[16] W. F. Tichy. *RCS – a system for version control. Software – Practice and Experience*, 15(7):637–654, July 1985.

[17] W. F. Tichy, editor. *Configuration Management*. Trends in Software. Wiley, Chichester, 1994.

[18] A. van der Hoek, D. Heimbigner, and A. L. Wolf. A generic, peer-to-peer repository for distributed configuration management. In *[11]*, pages 308–317, 1996.

[19] K. V. Volle. Eine prozeßbezogene Konfigurationsverwaltung. Master's thesis, IMMD-II, Universität Erlangen-Nürnberg, 1996.

[20] K. V. Volle. Verteilte Konfigurationsverwaltung: *COMAND*. Technical report, Basys GmbH, Am Weichselgarten 4, 91058 Erlangen, Germany, 1997. A postscript version can be requested from the author at volle@basys-gmbh.de.