A Conflict-Free Strategy for Team-Based Model Development

P.J. 't Hoen and M.H. ter Beek

Leiden Institute of Advanced Computer Science, Universiteit Leiden
P.O. Box 9512, 2300 RA Leiden, The Netherlands
E-mail: {hoen,mtbeek}@liacs.nl

ABSTRACT

Coordinating the efforts of multiple teams working in parallel on a model is a non-trivial task. A major part of this effort is to resolve conflicts, which are only detected when the work of the separate teams is integrated. In this paper we discuss how a model can be cut into distinct packages where in parallel each of these packages is locally modified by just one of the teams. Integration of the modified packages is straightforward as we only allow local changes to a package, i.e. changes that do not propagate beyond the package and that do not cause conflicts during integration. Additionally, we show how the package structure of a model and the teams working on the packages can be (temporarily) adapted to manage the need for non-local changes. We model the teams by team automata and discuss how their possible errant behaviour, which can lead to conflicts, is restricted by our strategy of model development.

Conflict

As models become larger and larger, it becomes inevitable to parallelise development in such a way that several teams must work in parallel on (parts of) the model. At some point in time the efforts of the teams are integrated and this, more often than not, leads to conflicts. Conflicts are changes to pieces of the modified parts of the model that do not match and that need to be resolved. Most of the time they are difficult and time consuming to resolve and they often require manual modeller intervention.

Conflict is a concept from the world of (Software) Configuration Management ((S)CM), the discipline for organizing and controlling evolving systems. CM is an old discipline, born out of systems manufacturing. It mandates procedures for identification of modules and their assemblies, for controlling releases and changes, for recording the product status, and for validating the completeness and consistency of a product [15, 16]. CM definitions [3] also cover areas like construction management, process management, and team work control.

In [11], a first approach was made to classify SCM functionality. This work examines the software process as enforced by existing SCM systems and distinguishes four CM models: a checkin/checkout model, a change oriented model, a composition model, and a long transaction model. Since this survey, many new SCM systems have emerged. All of these SCM models are however essentially based on one of these models [28].

These SCM models use a cooperation strategy [28] to ensure that changes are coordinated in such a way that one change does not, by accident, undo or conflict with the effects of another change. A *conservative* cooperation strategy prevents conflicting changes by using a simple locking scheme: developers working on a specific module version or configuration can lock it against further changes and while a version or configuration is locked, other developers are excluded from creating new versions. On the contrary, in an *optimistic* strategy each developer is active in his or her own workspace and various versions of the same module can be created. See [20] for an example.

In both conservative and optimistic cooperation strategies, parallel changes eventually need to be integrated or merged. This merging is either textual, syntactical or semantics-based [28]. All three approaches however lack early conflict detection. Conflicts only become apparent during actual merges. These conflicts then have to be resolved, a time staking business. A conservative cooperation strategy, as opposed to an optimistic strategy, reduces the potential number of conflicts. Each part of a model may only be changed by one team at a time and the situation where two or more teams are working at cross purposes is avoided. A change to one part can however affect all dependent parts and thus still lead to conflict during merge.

Avoiding Conflict

We note that problems during merge are avoided if we have a precise definition of when a change to a part is local, i.e. the change only affects the part and not the rest of the model. When using an optimistic strategy, each part is edited in its own workspace by one unique team. Then however, each team only makes *local changes* to its own part. Using this strategy, integration is straightforward and it can be done automatically as there are no conflicts. We call this strategy conflict free.

We illustrate our conflict-free strategy for the development of an object-oriented model and use as parts of the model packages of classes [21, 12, 19, 27] which are commonly used to structure a model. A notion of local change can, for example, be defined through invariancy of

the services offered through the interface of the package. The interface is then the contract of the package with the rest of the model [18].

In Figure 1 we give a part of a model where a package *Bank* models a real life bank[1]. Four of its departments are modelled as subpackages. The bank can be developed in parallel by four teams where each team separately develops one of the departments, as sketched in Figure 2. The changes made to each department are local and the integration to form the modified bank is straightforward. Note that these packages can be developed in entirely different geographic locations. Each team has its own workspace to make its changes and is only dependent on the other teams during a merge.

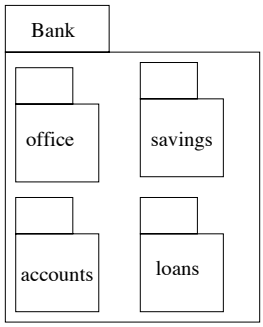

Fig. 1: The departments of a bank

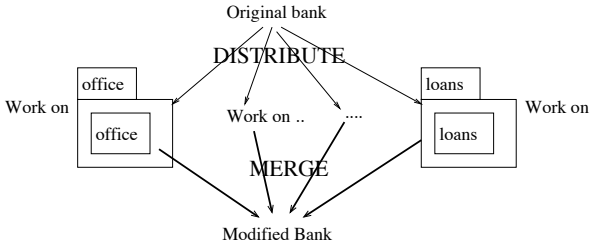

Fig. 2: Distributed development

We thus use an optimistic strategy where we constrain the changes in each workspace to prevent conflicts during integration. Figure 2 is an application of the Combined Reference Model and the View-Based Approach to System Specification [5]. A model, e.g. the bank, is split into several views for individual development and later integration. In this case we however block changes to the views which cause conflict during integration. Furthermore, this strategy is an extreme form of [13] and [14] which formalise and manage propagation of change to parts of a model. We additionally ensure that changes are only done to distinct parts/packages and we only allow these changes if they do not propagate beyond the packages. This work can also be seen as a variant of [22] where dependencies/constraints between modules are formalised using invariants in first order logic. Conflicts during merge are partially avoided by choosing the best possible fit, i.e. the parts which best meet the constraints. Our approach can be seen as a method of work where we actively block changes which violate these constraints; the existing interactions between the packages. In essence, the boundaries of our packages are change absorbers [10] which do not allow a change to propagate beyond a package.

The conflict-free strategy is useful, but in any realistic project the connections between the parts/packages of the model cannot stay the same during the life cycle of the model. There will be modifications required which require non-local changes of packages which invalidate the conflict-free strategy, i.e. a conflict-free merge cannot be guaranteed. These changes can however be localised by (temporarily) adding a new package, specifically to contain those original packages between which changes have to be made. These changes then are local with respect to the newly added package and thus allow the conflict-free strategy to be used for the model with the extra package.

In Figure 3, the packages P_1 to P_4 are edited using the conflict-free strategy. Non-local changes are however required between the packages P_2 and P_3. The work under development is merged and a temporary package *New* is added to group these two wayward parts. Note that because up to now the changes to the packages of the model have been local that the merge is without conflicts. The model is then redistributed with the new structure and work can continue with the conflict-free strategy as changes are once again local. The extra package can be removed once the new connections between the wayward packages are stable.

Note that in practice it may not be necessary to merge all the packages under development. It may be sufficient only to merge the packages for which the non-local changes are required to form a partial model. This is for example possible if each package is at most once the subject of such a temporary merge before a complete intermediary model is produced.

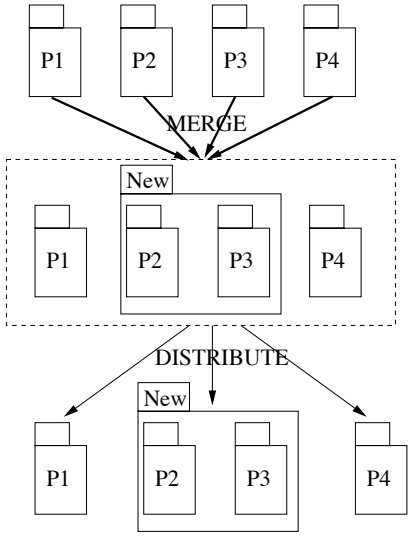

Fig. 3: A package is added

The architecture of a model is thus initially determined by top-down decomposition. This architecture can however be adapted to suit the need of our strategy. We

[1]The figure is drawn using the notation of [27].

call this part of the conflict-free strategy the renegotiation phase. Many such phases during the model life cycle are inconvenient. They are however an indication that the high-level architecture of the model is not yet stable, and possibly that the model is as yet too premature to be developed in a distributed fashion. Ideally, the initial breakdown of the model into packages should only be done by experienced modellers, thereby reducing the number of renegotiations as much as possible. The initial model should consequently be developed in one workspace until the confidence is high that a right choice has been made for a stable enough architecture to apply the conflict-free strategy to it. The same considerations apply for when one of the packages used in the conflict-free strategy is further split up into two or more subpackages for further parallel development.

Teams in the Conflict-Free Strategy

We also use the partitioning of a model into packages to dictate the team structure working on the model. Each team works on a distinct package of the model. Thus, for n packages, we will have at most n teams using the conflict-free strategy to work in parallel, each on one of these distinct packages.

Packages can be hierarchical, i.e. a package contains other packages as we have already shown in Figure 1. We use this hierarchical structuring of a package to likewise structure the teams working on the model using the conflict-free strategy. Teams, in our approach, can be hierarchical and the hierarchical decomposition of a package naturally leads to the decomposition of the team working on the package into subteams.

For example, in Figure 4 the hierarchical package P is given schematically. It contains the subpackages $P_{1,1}$ and $P_{1,2}$ and these two subpackages are likewise split into two smaller subpackages. In the same figure, the team T working on package P is given schematically. A dotted arrow from a package to a team indicates the team works (exclusively) on the package. Team T is split into two teams that work on the two subpackages of P and one of these teams is further split up, dictated by the package architecture. The conflict-free strategy is thus used to manage the efforts of T with the other teams working on other packages. The same strategy is also used within the hierarchical package P to internally structure the efforts of team T using subteams. Note that this is not required. We have not further split up team $T_{1,2}$ because in this example we have chosen to keep one large team to work on the entire package $P_{1,2}$.

The conflict-free strategy can thus be used to parallelise the development of the model into parts, up to the number of packages that exist in a model at the deepest level of nesting. The choice of packages then partially dictates the structure of the teams.

Note that during a renegotiation phase the team structure is affected to reflect the new distribution of packages. In Figure 3, we (temporarily) merge the teams working on packages P_2 and P_3 to reflect the fact that they are now working together to determine the new interactions between these packages. The initial team

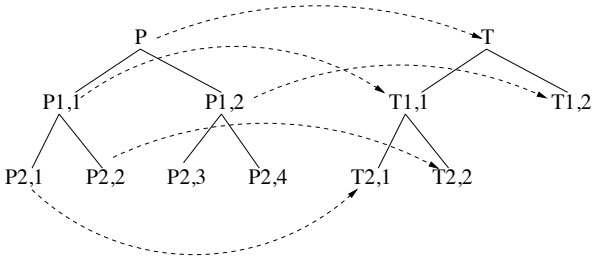

Fig. 4: Hierarchical teams

structure is thus determined by the architecture of the initial model and is adapted dynamically due to renegotiation.

In the example of Figure 3, the wayward packages P_2 and P_3, which are edited by the teams T_2 and T_3, are temporarily placed in a package *New* during renegotiation. These two teams together are then responsible for modifying this new package, as sketched in Figure 5.

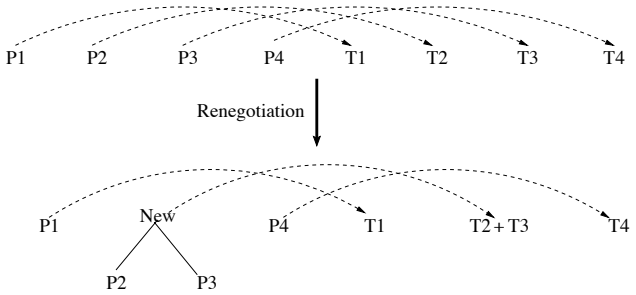

Fig. 5: Merging teams

The structure of the model and the structure of the teams are thus tightly coupled. The initial model determines how the teams can be distributed over the packages for parallel development. On the other hand, desired non-local changes of one of the teams can lead to a (temporary) change in architecture. The model itself is "actively" involved in the development process. This is opposed to many workflow [17] or software process models [4] where the model under development is not really relevant. They focus more on the to be produced documents and their timing. The contents of these documents do not really play an explicit role.

The activities of the teams in our approach can be divided into two categories; those which are internal to a team and those which involve other teams due to renegotiation. The management of the teams in the conflict-free strategy can be divided along these lines. On one hand, management can be localised and be only concerned with coordinating the changes to one package by one team. Here the focus is on coordinating a relatively small group in a well-defined context. On the other hand, the structure of the teams can be a separate management concern. The management of the hierarchical structure of the model and of the teams as given in Figure 4 can become an issue in its own right. This is a relatively more complex job than "just" managing one team. Seniority and experience can play a role in determining which role is played by an individual. Relatively unexperienced in-

dividuals should manage smaller teams, like $T_{2,1}$ while a more experienced manager could lead the more complex $T_{1,1}$. The most experienced manager can decide whether changes which lead to renegotiation fit within the direction the model should be heading.

Note that we do not discuss how one team should be led. We postulate a group of people who together perform a common editing of one package. We do not claim that they should coordinate their work in any specific way. We just define the extent of their possible changes through only allowing local changes. We also do not discuss how two separate teams when integrated should coordinate their efforts. This is a non-trivial task, especially if the two teams previously worked according to different philosophies. We just constrain the extent of their possible actions as a new, larger team. This is a topic of research with strong sociological impact, which is outside the scope of this paper. The conflict-free strategy however does provide a context within which knowledge about how people work can be embedded.

Team Automata

In the previous section we have sketched how a hierarchical team structure can be induced by the structure of the model under development. Here we discuss how we can model this structure in terms of team automata [6, 1]. For their formal details we refer the reader to [1]. Team automata were introduced in the field of Computer Supported Cooperative work (CSCW) as a modelling tool for capturing groupware notions such as coordination, collaboration, and cooperation in a mathematically precise way [7].

Team automata are composed from component automata. The basic concept underlying both team automata and component automata is a labelled transition system with initial states, which captures the idea of a system with states, together with actions the executions of which lead to (non-deterministic) state changes. Three types of actions of such automata are distinguished: *internal* actions, which cannot be observed by other automata, and externally observable *input* actions and *output* actions, which are used for communication between automata. We interpret actions as operations/changes of (a package of) the model. For example, we can characterise an action as the modification of a name of a model element. Since internal actions of a component cannot be observed by any other automaton, these actions are ideally suited for representing a local change to a package using the conflict-free strategy. The external actions, on the other hand, are ideal for the coordination between packages.

In Figure 6, we represent our example teams T_2 and T_3 by two quite trivial component automata. The states are numbered 1 to 5, the wavy arcs indicating the initial states, and the transitions are labelled with the actions a to d. Now let b and c be internal actions of T_2, whereas actions a and d are output actions in both T_2 and T_3. Hence a possible scenario could be as follows. First T_2 and T_3 execute action a in parallel. Then T_2 does a number of internal actions/local changes to its package without consulting the other teams. Eventually both teams can execute action d in parallel, after which this process can be repeated. Naturally we could imagine also T_3 having some internal actions/local changes to perform once in a while.

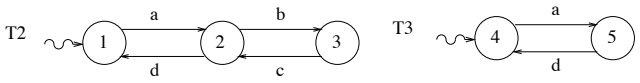

Fig. 6: Component automata T_2 and T_3

A team automaton is a set of component automata coordinated in such a way that components can either simultaneously participate in one team action (i.e. *synchronize* on this action) or remain idle during this team action. The state change of a team automaton is thus based on the local state changes of the component automata involved in the executed action. In Figure 7 the reachable part of a team automaton $T_{2,3}$ over T_2 and T_3 is given. Note that actions a and d are synchronizations between T_2 and T_3 requiring both automata to change state, whereas only T_2 is changing state when b or c is executed. The behaviour of both T_2 and T_3 is thus reflected in the behaviour of $T_{2,3}$. In our interpretation, such synchronizations can represent changes which affect two or more packages, i.e. non-local changes. The external actions of $T_{2,3}$ in Figure 5 thus represent the shared operations on the merged packages P_2 and P_3. These extra operations consequently can be hidden [1] so as to present a team with only internal actions/local operations on its packages for further usage.

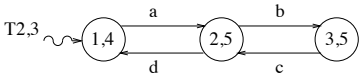

Fig. 7: A team automaton $T_{2,3}$ over T_2 and T_3

Several different types of such synchronizations on actions shared between components of a team are classified in [1]. The synchronization on output actions as seen in our example is called *peer-to-peer* because external actions of the same type synchronize. This is a useful notion for modelling and measuring team work. Another such useful notion is a *master-slave* synchronization, defined as a synchronization between input actions and output actions. The former can only be executed as a response to the latter, i.e. input is a slave of output. This concept can thus capture a boss-employee relation in which an employee has to follow orders from his or her boss.

The construction of a team automaton over a set of component automata sketched above is formally defined in such a way that a team automaton is itself a component automaton again. This allows us to use such a team as part of a larger team automaton again. In this way subteams and hierarchical team structures can be modelled. For example, in Figure 8, team T is defined as a composition of our team $T_{2,3}$ with automata T_1 and T_4. As such, team automata are well suited for modelling

(the actions of) the hierarchical teams in the conflict-free strategy.

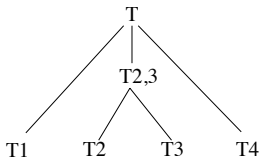

Fig. 8: A team T over T_1, T_4, and $T_{2,3}$

Results

In this paper we have discussed a strategy for the development of a model by several teams working in parallel on distinct packages of the model. We guide the changes made by each team so as to ensure no conflicts occur during the merge of the produced efforts. Hence we call our approach a conflict-free strategy. This approach is even scalable as each package can be developed in a similar fashion by further splitting up the package. We have moreover shown how packages under development can (temporarily) be merged if changes to a package that would invalidate the conflict-free strategy are required. We call this operation the renegotiation phase.

Additionally, we have discussed how the hierarchical structure of the model in packages can be used to structure the teams working on the model. The top-down decomposition of a model into packages guides the decomposition of the people working on the model into similarly structured teams. The renegotiation phase, when packages are temporarily merged, then gives heuristics on how the teams should further cooperate to implement changes without generating conflicts. We have sketched how we can formally model this by team automata.

The conflict-free strategy along with the explicit discussion on the team structure and its actions brings the worlds of CSCW, software engineering, CM, and process modelling very close together. We have discussed how a large model can be developed and how the work between the people doing the actual work can be coordinated. Special to the approach is that the subject of the work, the model under development, is used to structure the work and "plays" an active part in which changes are possible.

Future Work

Any object-oriented modelling language with packages and a notion of local change is suitable for use in the conflict-free strategy. We are currently working on a non-trivial notion of local change for SOCCA [2, 8, 9] packages [23]. SOCCA (Specification Of Coordinated Cooperative Activities) is a high-level specification language as well as a specification approach for object-oriented modelling. The behaviour of a class is defined from several perspectives with a high degree of parallelism and the separation of concerns in modelling functionality and coordination. The reader is referred to [26] for a more extensive introduction. Our notion of local change of a package will incorporate a notion of unchanged behaviour of the package with respect to the rest of the model. We will use this notion of local change to develop a conflict-free strategy for components in SOCCA as defined in [24, 25].

References

[1] M.H. ter Beek, C.A. Ellis, J. Kleijn, and G. Rozenberg. Synchronizations in team automata for groupware systems. Technical Report TR-99-12, Leiden Institute for Advanced Computer Science, Universiteit Leiden, 1999. For reprints, contact the first author.

[2] T. de Bunje, G. Engels, L. Groenewegen, A. Matsinger, and M. Rijnbeek. Industrial maintenance modelled in SOCCA: an experience report. In *Proceedings of the Fourth International Conference on the Software Process (ICSP4) — Improvement and Practice, Brighton, UK*, pages 13 – 26. IEEE, ISPA, IEEE Computer Society Press, Los Alamitos, California, 1996. Also available as Technical Report 96-37, Department of Computer Science, Leiden University (`ftp://ftp.wi.LeidenUniv.nl/pub/CS/TechnicalReports/1996/tr96-37.ps.gz`).

[3] S. Dart. Concepts in Configuration Management Systems. In P.H. Feiler, editor, *Proceedings of the Third International Workshop on Software Configuration Management, Trondheim, Norway*, pages 1 – 18. ACM Press, 1991.

[4] J.-C. Derniame, A.B. Kaba, and D. Wastell, editors. *Software Process: Principles, Methodology, Technology*, volume 1500 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1999.

[5] H. Ehrig, G. Engels, R. Heckel, and G. Taentzer. A Combined Reference Model- and View-Based Approach to System Specification. *International Journal on Software Engineering and Knowledge Engineering*, 7(4), 1997.

[6] C.A. Ellis. Team automata for groupware systems. In J. Clifford, B. Lindsday, and D. Maier, editors, *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work: The Integration Challenge (GROUP'97), Phoenix, Arizona*, pages 415 – 424, 1997.

[7] C.A. Ellis and J. Wainer. Groupware and computer supported cooperative work. In G. Weiss, editor, *Multiagent Systems. A Modern Approach to Distributed Artificial Intelligence*, pages 425 – 457. 1999.

[8] G. Engels. Modelling of Collaboration with UML and SOCCA. In H. Ehrig, G. Engels, F. Orejas, and M. Wirsig, editors, *Semi-Formal and Formal Specification Techniques for Software Systems*, number 218, page 13, 1998. See `http://www.dag.uni-sb.de/DATA/Seminars/98/`. Joint work with Luuk Groenewegen.

[9] G. Engels, L.P.J. Groenewegen, and G. Kappel. Coordinated Collaboration of Objects. In M. Papazoglou, St. Spaccapietra, and Z. Tari, editors, *Object-Oriented Data Modelling Themes*. MIT Press, 1999.

[10] Huw Evans. Architectural evolution: A multi-level perspective: Workshop w04 ECOOP 99 on object-

oriented architectural evolution, June 1999.

[11] P.H. Feiler. Configuration Management Models in Commercial Environments. Technical Report CMU/SEI-91-TR-7, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Philadephia, 1991.

[12] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice-Hall, Inc., Englewood Cliffs, 1991.

[13] J. Grundy and J. Hosking. Constructing Integrated Software Development Environments with MViews. *International Journal of Applied Software Technology*, 2(3-4), 1996.

[14] J. Grundy, J. Hosking, and W. Mugridge. Supporting Flexible Consistency Management via Discrete Change Description Propagation, September 1996.

[15] IEEE, New York. *IEEE Guide to Software Configuration Management*, 1988. ANSI/IEEE Standard 1042-1987.

[16] IEEE, New York. *IEEE Guide to Software Configuration Management Plans*, 1990. ANSI/IEEE Standard 828-1990.

[17] S. Khoshafian and M. Buckiewicz. *Introduction to Groupware, Workflow, and Workgroup Computing*. John Wiley & Sons, New York, 1995.

[18] B. Meyer. Applying design by contract. *IEEE COMPUTER*, 25(10):40–51, October 1992.

[19] B. Meyer. *Object-oriented Software Construction*. Prentice-Hall, Inc., New York, New York, 1997.

[20] T. Page, J. Guy, J. Heidemann, D. Ratner, P. Reiher, A. Goel, G. Kuenning, and G. Popek. Perspectives on optimistically replicated peer-to-peer filing. *Software – Practice and Experience*, 11(1), 1997.

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

[22] S. Sachweh and W. Schäfer. Version Management for tightly integrated Software Engineering Environments. In *Proceedings of the Seventh International Conference on Software Engineering Environments, Noordwijkerhout, Netherlands*. IEEE Computer Society Press, 1995.

[23] P.J. 't Hoen, J.H.M. Dassen, L.P.J. Groenewegen, I.G. Sprinkhuizen-Kuyper, P.W.M. Koopman, and G. Engels. SOCCA extended with UML like packages. Technical Report 99-06, Leiden Institute of Advanced Computer Science, September 1999.

[24] P.J. 't Hoen and L.P.J. Groenewegen. Abstract Plug-In Components with Behaviour, June 2000. accepted at WCOP, `http://www.ipd.hk-r.se/bosch/WCOP2000/`.

[25] P.J. 't Hoen, J.N. Kok, G. Busatto, and L.P.J. Groenewegen. From OO to Components: Components at the type and instance level (extended abstract). July 2000. to appear, accepted at SCI2000, `http://www.comp.lancs.ac.uk/computing/research/cseg/conferences/sci2000/` and downloadable at `http://www.liacs.nl:/~hoen/publications/sci2000.ps`.

[26] P.J. 't Hoen, V. Zweije, and L.P.J. Groenewegen. SOCCA Basics. Technical Report 99-14, Leiden Institute of Advanced Computer Science, 1999. downloadable at `http://www.liacs.nl:/~hoen/#Publications`.

[27] Unified Modeling Language 1.3. Technical report, Rational Software Corporation, 1999.

[28] Andreas Zeller. *Configuration Management with Version Sets*. PhD thesis, Technical University of Braunschweig, Germany, 1997.