Next-Generation Viewpoint-based Environments

Wolfgang Emmerich, George Spanoudakis and Anthony Finkelstein
Dept. of Computer Science, The City University,
Northampton Square, London EC1V 0HB, UK
{emmerich|gespan|acwf}@cs.city.ac.uk

Abstract

This paper discusses the notion of, and outlines requirements for Viewpoint-based Environments. These are next-generation CASE environments, which support the specification of requirements from multiple perspectives or so-called viewpoints. Requirements for such environments are mainly concerned with the detection and management of interference between viewpoints. Viewpoint-based Environments should also support the cooperation of multiple developers and maintain development histories in terms of multiple viewpoint versions. We briefly sketch an architecture for such environments and outline a research agenda for developing them.

1 Viewpoint-based Environments

It has been widely recognised that a software development method that is appropriate for a particular software process might prove disastrous if it is applied to another process. A classical information system might best be defined using an entity relationship approach. A database schema can be generated from the entity relationship model and a fourth generation language might be used to implement forms and reports. A safety-critical real-time system, in contrast, must not be developed in this way. It will require a formal specification to be defined, for instance in Z [7], and its implementation should partially be proven against this formal specification. As we will not be able to suggest a method mix that can be used in any development method, the need arises to support the design of development methods and their integration into a method mix that is tailored towards particular processes. The Viewpoint framework as discussed in [4, 5] have been dedicated to this problem. Viewpoints are defined as a loosely-coupled, locally managed object encapsulating representation knowledge, development process knowledge and partial specification knowledge about a system and its domain. Viewpoints are structured into five slots:

- The *style* slot describes the representation scheme used by the viewpoint.
- The *work plan* slot describes development actions together with a strategy for their application to construct the viewpoint.
- The *domain* slot describes the area of concern of the viewpoint and sets its context in the overall system under construction.
- The *specification* slot describes the viewpoint domain in the representation scheme determined by the style slot.
- The *work record* slot records the history and current state of the viewpoint development.

A *viewpoint template* is a viewpoint type in which the style and work plan slots have been filled. It may be *instantiated* and then yields a viewpoint where the remaining slots can be filled. A *method* definition can then be seen as a collection of related viewpoint templates. The instantiation of various viewpoint templates need to be supported by appropriate tools. These tools will provide the necessary editing commands and/or assembling actions for implementing the workplans for these templates and will keep track of work records in a way transparent to the viewpoint owners. Such tools may need to be configured into integrated environments if their associated viewpoint templates together constitute a development method. We will refer to such configurations as *viewpoint-based environments*. The requirements for such environments are discussed below and built on a series of experiments, prototypes and systems developed by the authors and contributors of the viewpoints framework.

2 Advanced Requirements

A single tool will enable viewpoint owners or other tools to construct viewpoints by instantiating viewpoint templates. Viewpoint construction will be supported by *editing commands* for all the assembly actions identified in the work plan of the template. It will also be supported by browsing commands allowing the visualisation of the viewpoints' contents and traversing of relationships between viewpoints. To the extent that these relationships associate viewpoints of different templates it is necessary to support an effective communication between tools.

The autonomous instantiation of viewpoint templates doesn't necessarily require tools to be substantially different from generic text and graphics editors or hypertext viewers. The distinguishing feature of a viewpoint-based environment is the fact that viewpoints may interfere with each other to the extent they refer to or assert properties of common aspects of the system under development and its domain. This interference may take the form of either a mere ontological overlap (i.e. incorporation of components in different viewpoints referring to common aspects of the "real world") or an inconsistency between ontologically overlapping viewpoints. In both cases it needs to be managed.

The interference management which is required (or indeed possible) varies. In certain cases it might need to be lazy, leaving it up to the viewpoint owner to decide about both the kind of the checks and the time to perform them. This avoids the disruption of viewpoint development by notifications of overlaps and inconsistencies, which could - anyway - be dealt with later on. In other cases, interference management might need to be eager. Certain overlaps and inconsistencies may lead to a waste of effort if they become apparent only very late in the development process, if at all. To avoid them, an eager approach where viewpoint owners get immediate feedback from automatically invoked checks, is necessary. These two approaches regarding the time of interference management can be supported by tools, which switch between eager and lazy interference checks.

Similarly, the way to deal with detected interferences may vary. In some cases, interference should be prevented. This might be appropriate in cases where many further actions would be affected by a pending interference. Prevention goes hand in hand with eager checking. Interferences that should be prevented can be dealt with in three different ways. The first is the immediate rejection of the command, whose completion would cause an interference. The second is to let the tool undertake a set of default actions, appropriate for dealing with the type of interference detected. The third is to guide the viewpoint owner(s), according to some resolution strategy, to rectify immediately the problem or otherwise to abort the execution of the problematic command. However, there are types of interference that may well be tolerable. These might be interferences among evolving viewpoints or viewpoints which reflect different perspectives over which a decision cannot be reached for the time being or which should be maintained along the system life-cycle. Interferences of those types should be maintained as pending and be dealt with whenever the viewpoint owner(s) or the overall progress of system development requires. In dealing with them, viewpoint owners may decide to rely on default or guidance-based resolution strategies.

Different resolution strategies may or may not be required to deal with tolerated inconsistencies. Some inconsistencies may just be due to different point of views that do not cause any harm and may well coexist. Developers should not be forced to resolve these inconsistencies. Others, however, may have a negative impact on the development of related viewpoints and should be resolved at some stage, defined in the workplan slot of the viewpoint. The resolution strategy that is to be deployed then depends on whether only a single or multiple developers are involved. If a single developer is the owner of all the viewpoints involved in the inconsistency, he/she will have complete freedom to decide when and how to remove the inconsistency. All in-viewpoint consistency constraint violations fall into this category. If different developers are owners of viewpoints involved in a conflict, negotiations among them will be required to agree who will perform actions to resolve the inconsistency. We note that significant cooperation is involved in this and viewpoint-based tools will have to support this cooperation.

If multiple developers cooperate on the resolution of interference, it will be inevitably necessary to review the impact of each other's changes as they occur. In the above example and with an eager approach towards constraint checking, an inconsistency of the Booch viewpoint should be removed as soon as the other developer has changed the

C++ class name. With a lazy approach, the designer of the Booch diagram should see the resolution after he has applied the respective check command the next time. We note that viewpoint-based tools have to be integrated in a way that they can access and update each other's viewpoints in a concurrent way.

It is often not appropriate for a developer to be disturbed by other developers' changes. Developers may want to work in isolation for some period of time, especially when they perform major changes to a viewpoint. Moreover,their changes should only become visible to other developers after they have reached a certain degree of (in-viewpoint) consistency. A way of achieving this is to arrange for viewpoint-based tools that are able to maintain different *versions* of a viewpoint. Version management has so far only gained widespread attention for source code viewpoints that are produced during implementation tasks, but we strongly believe that any viewpoint produced during any task of a software process deserves the same attention. The concept of viewpoint versions is not only required for isolating developers from each other, but also to keep track of the viewpoint history while a system is under maintenance. When a system is ported to a new platform, for instance, the versions of viewpoints for the previous platform must be retained. Developers will then need to *freeze* versions of a viewpoint so as to prevent it from being further modified. This will be necessary whenever a viewpoint has reached a state to which it might have to be restored in the future. Developers will then need a mechanism to *derive* a version from another frozen version and *select* a particular version. Then successive changes must only be done in that selected version. If no version is selected, a *default version* of a viewpoint will be used. Further version management support is required for labelling versions, traversing through the version history graph and for merging different alternatives to a common successor version.

Given that viewpoints exist in different versions and inter-viewpoint consistency has to be checked, a need arises to maintain *viewpoint configurations*. A viewpoint configuration is determined by a viewpoint that serves as a component model and a number of version selection rules. The component model viewpoint identifies all viewpoints that are components of a configuration. Version selection rules then identify for each component viewpoint the particular viewpoint version that belongs to the configuration. Version selection rules can be *explicit* or *associative*. Explicit version selection is based on the identification of a version for each component viewpoint. Associative version selection rules determine versions by a particular property they have to fulfill. An *implicit selection rule* determines the *default configuration*. It consists of all default versions of all component viewpoints and can be considered as the baseline of a development. Different versions of the component model viewpoint are then used to store different configurations.

We note that it is often not required to define a viewpoint whose only purpose is to serve as a component model. It is rather very likely that such a viewpoint can be identified among the viewpoints that are produced anyway. In the above example, the Booch class diagram viewpoint can serve as a component model. Any class in the Booch diagram represents two viewpoints. One for the C++ class interface definition and another one for its implementation.

3 Architectural Considerations

An autonomous implementation of viewpoint tools has always been one of our major concerns. Autonomy to this respect means that viewpoint tools are developed and executed in a distributed way and that they can be developed in a heterogeneous manner. We suggest employing an object request broker architecture, like OMG/CORBA to achieve this. About a dozen implementations have become available and they can be used to provide access to viewpoints in a heterogeneous and distributed manner.

To achieve persistent storage of viewpoints, concurrent viewpoint access as well as version and configuration management, we advocate the use of object databases [1, 2]. Therefore, the structure definition as defined in viewpoint templates has to be implemented in an object database schema. Instances of this schema then implement different viewpoint specification slots. The transaction management of object databases can be used to control concurrent access of viewpoint based tools, if transactions are used to implement individual tool commands rather than complete editing sessions. Some object database implementations provide version management facilities for composite objects so that the composite objects that represent viewpoint specification slots can be versioned on behalf of viewpoint based tools.

4 Summary and Further Work

We have discussed requirements for a next-generation of Viewpoint-based Software Development Environments. These requirements were mainly concerned with interference detection, management and their resolution. Furthermore, we have identified the need for cooperation that should be computer-supported. Finally different versions and configurations of viewpoints must be maintained. We then briefly sketched how to achieve a heterogeneous and distributed environment architecture on the basis of OMG/CORBA and how to address persistent storage of viewpoints, concurrency control and version management by employing object databases.

We have to investigate how these environments can be constructed systematically. A first approach was taken in [2], where the high-level GOODSTEP tool specification language (GTSL) [3] was developed together with a compiler for this language. We believe that many of the techniques suggested there can also be applied for the specification of viewpoint-based environments. The feel, however, that it is possible to raise the level of abstraction so as to make tool specification even more simpler.

A method for the reconciliation of specifications on the basis of detecting ontological overlaps has been defined. This method is based on a computational model for the detection of similarities [6]. This method needs to be implemented in the above architectural setting of distributed computing based on OMG/CORBA and persistent storage in object database systems.

References

[1] R. Cattell, editor. *The Object Database Standard: ODMG-93.* Morgan Kaufman, 1993.

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

[3] W. Emmerich. Tool Specification with GTSL. In A. L. Wolf and J. Kramer, editors, *Proc. of the 8^{th} Int. Workshop on Software Specification and Design, Velen, Germany.* IEEE Computer Society Press, 1996. To appear.

[4] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: a framework for integrating multiple perspectives in system development. *Int. Journal of Software Engineering and Knowledge Engineering*, 2(1):21–58, 1992.

[5] B. Nuseibeh, J. Kramer, and A. Finkelstein. A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification. *IEEE Transactions on Software Engineering*, 20(10):760–773, 1994.

[6] G. Spanoudakis and P. Constatopoulos. Elaborating Analogies from Conceptual Models. *International Journal of Intelligent Systems*, 1996. To appear.

[7] J. M. Spivey. *The Z Notation - A Reference Manual.* Prentice Hall, 1989.