




*Why struggle over reams of abstract code when
you could walk through the architecture instead?*

3D VISUALIZATION OF SOFTWARE ARCHITECTURES



SOFTWARE ARCHITECTURE REFERS TO THE ART AND SCIENCE OF STRUCTURING VERY large programs and concerns the organization of a system in terms of its components, global control structures, physical distribution, evolution, and more. The discipline of software architecture recently received a boost from new technological options for visualizing and analyzing software systems. These options are the result of the virtual reality modeling language (VRML), which is a de facto worldwide standard. For example, we've found it easy to gen-

erate virtual worlds from the "uses relations" of large software systems, allowing software engineers to see the architecture of the systems they are working on as 3D structures. They can create interesting viewpoints with respect to an architecture, looking at it, literally, from different perspectives.

The cornerstones of the software analysis methodology we describe here are extraction, visualization, and calculation. Extraction concerns obtaining structural information from source code, specification, and documentation files and casting it in a form suitable for further analysis. Visualization is needed to make the extracted information accessible to the analyst. But there is no such thing as the one right view; there are many different views of a given system, and an analyst needs the right tools to create

such views fast. For example, certain design objects can be derived from a particular view, and several types of relations can be combined. This process of using views is where calculating with relations comes in; we've had good experience with tools for manipulating binary relations. For example, suppressing certain design objects from a view amounts to accepting the domain restriction of a relation. Combining relations can also be done through such operations as union, intersection, and difference. In other words, we calculate with relations. A relation is flexible and powerful, but it doesn't help much to look at all of a system's individual design objects. From an architectural point of view, it is more important to find out how design objects are related.

The approach we've used as a basis for our 3D

LOE FEIJS AND ROEL DE JONG

work is documented only in internal Philips publications [4, 5, 12]. So in summarizing the approach here, we say tools are available to support the three cornerstones of software analysis—extraction, visualization, and calculation. For extraction, many techniques and tools help extract uses (such as import or call-tree) and part-of relations (such as in-layer) from source code and from other files (such as Unix awk scripts and QAC) [10]. In this respect, our approach is similar to that in [2, 3, 6, 8], which also use tools to extract and check uses and part-of relations from code.

For visualization, we use the TEDDY browser, built by R. C. Van Ommering, a Dutch researcher at Philips, enabling users to automatically read relation files as well as the associated layout information [1]. The browser also supports several interactive editing functions, such as choosing object shapes and modifying and storing layouts. The idea of storing layouts is advantageous because only a few uses-pairs are added during minor design modifications. In such cases, the results of the layout work (done manually and hence expensive) can be reused, whereas the new relation is automatically extracted from the code. We could have automated the layout work too but found that artificial layouts are much more useful because they can be made to fit designers' intuition about their designs.

For calculating with relations, we use tools inspired by the mathematical theory of relations [7, 11]. For example, one can calculate the transitive closure of a relation in many kinds of reachability problems. An interesting type of operation is the Hasse operator, or the (pseudo)-inverse of the transitive closure operation; some authors of mathematical literature call it "transitive reduction." If we have a given cycle-free relation R , then $\text{Hasse}(R)$ is a relation containing all the pairs of R , except for the shortcuts. For example, the pair (x, z) is a shortcut if there are already pairs (x, y) and (y, z) in R . This operation can be used to simplify a uses-relation, removing many redundant pairs.

Another useful operation is lifting [4, 8], which is

MAKING DESIGN OBJECTS CONSISTENT WITH AN OBJECT'S INTENDED ROLE, CHOOSING NEW VIEWPOINTS, AND IDENTIFYING NEW DESIGN METAPHORS ARE SERIOUS REASONS FOR INVESTIGATING 3D.

important for abstracting a given uses-relation according to some grouping of the related design objects. The low-level uses is converted into a uses-relation on the level of the groups, according to the following definition: There is a pair $(G1, G2)$ in the resulting lifted relation if there is an object x in group $G1$ and an object y in $G2$ such that (x, y) is a pair in the original uses-relation.

If a structure is too complex to be viewed as a whole, the calculation tools can help calculate views (such as lifting, Hasse, transitive closure, and subset-

ting) and perform further analysis.

Why 3D?

There are several reasons why it is worthwhile to investigate and apply the emerging VRML technology to visualizing software designs. The main one is that VRML technology is available today and can help upgrade 2D approaches to show design information. Moreover, 3D makes it possible to make a layout of the design objects more consistent with the object's intended role than can be done in a single plane. An example that occurs frequently is when trying to explain a layered software architecture in which components are grouped into layers and the uses-relation is supposed to be restricted to vertical use by a layer using only the next lower layer. However, a lot of confusion can result from trying to draw a group of components in 2D representing resources for use by all layers (such as data type definitions and operating system functions). Some programmers propose vertical layers; others object because these vertical layers are not really layers or try to restrict the way the operating system is used. This dilemma could be remedied by viewing the resource layer as a back-plane in the third dimension.

3D also provides new options for choosing viewpoints to provide new views of a given design. We can zoom-in (by walking) and choose another angle (by rotating the design). The classical ways of developing views (such as algebraic manipulation of relations and placing of nodes) are still available, but new mechanisms can now be explored.

3D visualization helps identify new design



metaphors, fostering new ideas with respect to design principles. There are more possibilities to shape objects and lines, beyond the usual flat options (such as boxes, ovals, icons, text labels, and arrows). Well-chosen shapes help programmers and architects remember complex structures. And 3D designs can be attractive and made to be fun to walk through. Such visual appeal can be used for presentation purposes for, say, explaining a product's design to potential customers.

Although helping remember complex structures and making appealing images are alone not strong reasons for pursuing 3D, making design objects consistent with an object's intended role, choosing new viewpoints, and identifying new design metaphors are serious motivation and should be investigated.

VRML is a way of describing multiparticipant interactive simulations—virtual worlds networked via the Internet and hyperlinked through the World-Wide Web. The VRML 1.0 used in our work has allowed us to create virtual worlds with limited interactive behavior. VRML has a very readable formatting language with an ASCII syntax, like TEX and HTML, so VRML is easy to generate, and libraries of objects described in VRML pose no special problems. Objects are built from such elementary shapes as cylinders, cubes, and cones. For VRML information, we refer to vrml.wired.com/vrml.tech/vrml10-3.html. Several browsers are available, including VRWEB (with which we began our earliest experiments) and plug-ins for Netscape, such as Live3D, available through the Web.

As there are various ways to explore a software architecture using an extra graphical dimension, a few choices must still be made. For example, we used the third dimension for grouping nodes (such as modules) of a design into a number of horizontal layers, according to some freely selectable partition of the node set. Each next-higher layer is positioned equidistantly above the preceding layer. We also used arbitrarily shaped and colored 3D body types for showing the nodes in several LEGO-brick styles (see Figure 1) according to selectable attributes (other than the ones used for layering). We show relations between nodes with colored arrowed lines, or small radius tubes with a cone in the middle, between the LEGO bricks. Several relations are shown simultaneously using a separate color for each relation. We used the current TEDDY editor and file

formats for preparing layouts within each plane, together with the extraction and calculation tools described earlier.

We built a simple VRML 1.0 generator called ArchView as a prototype in the CLEAN programming language [9]. The editing functions of ArchView include interactive editing of node-set order, determining plane stacking, and relation coloring. The tool contains a hardwired library of fixed-shape VRML objects, including LEGO bricks, arrowed pipes, and hat boxes.

When asked to generate a virtual world, ArchView derives for each node a 3D coordinate from its 2D coordinate and its layering position, examining the node type from a layout file, and picking a corresponding brick from the LEGO library. ArchView also instantiates the colored arrows corresponding to all

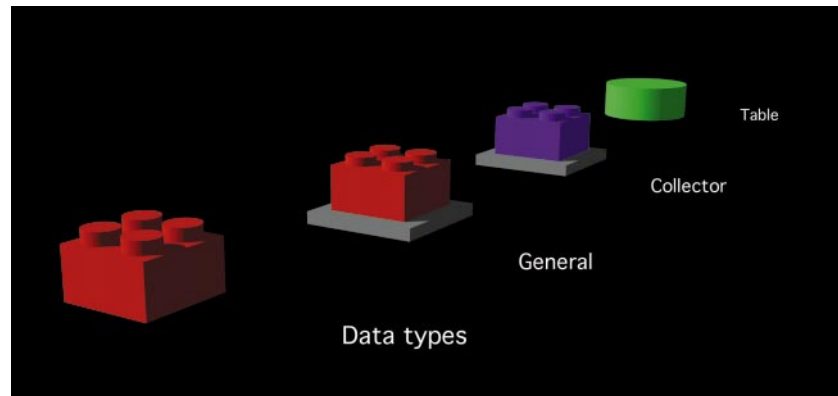


Figure 1. Module type values mapped onto LEGO-like bricks

related relations. The generated objects are in VRML world format, ready to be browsed by any Web browser with VRML plug-in.

An example application of our 3D visualization experiment is a small (but nontrivial) software system—the authoring environment of Philips's Generation of User Control Software (GUCS) project. GUCS allows a product manager to select a platform (within the product family) and a set of user tasks, then customize them to generate a concrete product. After this authoring process, the control software for the product can be generated instantly. GUCS contains 34 modules, which have an import relation in Modula-2 style. Thus, if a module M1 imports M2, all exported symbols from M2 are within the scope of M1. (The design includes 105 module imports.) Visualization in GUCS involves assigning a type and a kind to each of the design's modules. A module can have four values. For example a "Table" module contains data in tabular form, that is, it includes no

functions or procedures. A “Collector” module bundles a collection of modules, importing the modules it bundles, exporting them all, and exporting management information, such as module name. The other module types are called “General” and “Data types.”

The various module type values are mapped onto LEGO-like bricks, as in Figure 1. Note that these types are based on a very general classification, quite independent from the GUCS system and the programming language used to create it.

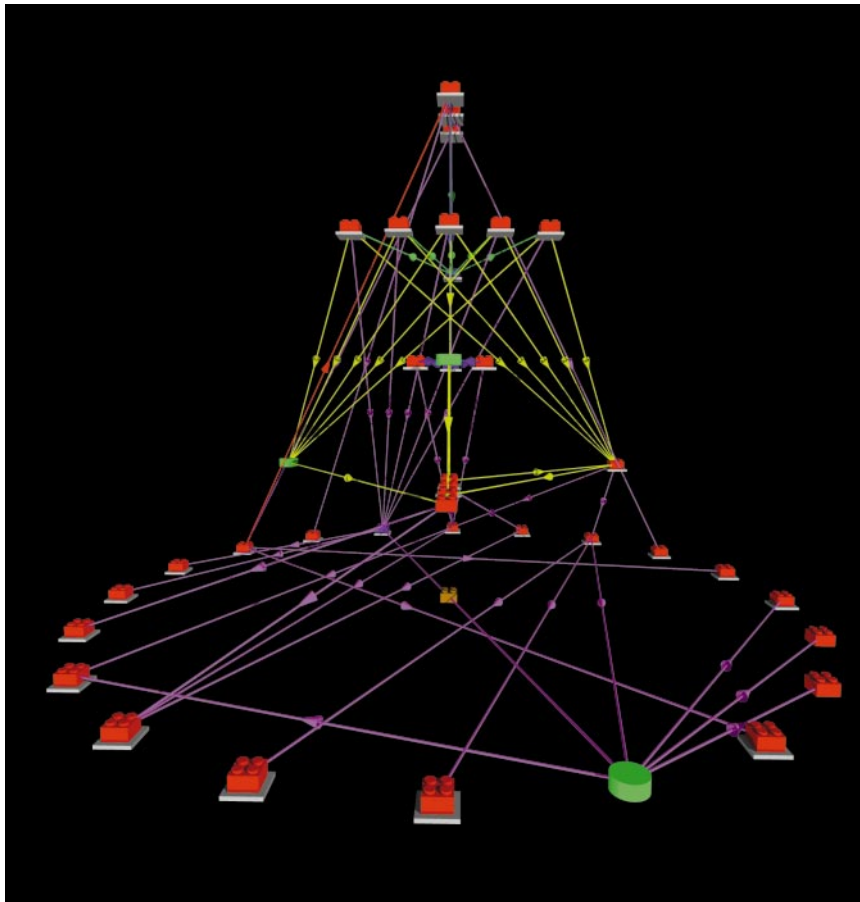


Figure 2. Import relations, all modules

The kind of a module is an orthogonal classification with respect to type and can have five different values: authoring (modules concerned with authoring dialogues); user tasks (modules encapsulating generic user tasks); family and devices; method; and environment (library utilities and graphics I/O). The last classification is specific to this application domain.

Generating Worlds

The kind of a module determines the layer in which it is placed by the ArchView VRML generator. The imports of the GUCS modules are shown as colored arrows in the LEGO world. If a module M1 imports

M2, an arrow connects the M1 node to the M2 node. As a relation color key, we partitioned the original import relation according to the kind of the imported module of each relation element. Before color partitioning, the Hasse operation reduces the size of the relation by 50%. Therefore, if the method subrelation is blue, any blue arrow from a module means that this module is method-dependent; according to Hasse operation semantics, it may then also be implicitly dependent on any of the environ-

ment modules. To experience the 3D effect, we don't read a paper report but look in the *.wrl file in a Web browser and do the navigation ourselves.

Figure 2 shows the result of ArchView generation after editing each of the five planes with TEDDY into regular constellations wherever possible. We defined a circle for the environment base plane, a Y shape for the method, a kite for a family, a pie segment for user tasks, and a linear arrangement for the authoring modules on top. This picture provides a quick impression of the relative dimension of the planes, and, assuming roughly equal-size module internals, a first impression of the method's complexity, family design, and user-task design. Furthermore, without being interested in the internal components or even the identity of modules, we can see that the method Y has two modules that are used heavily by user tasks;

the environment likewise has only a few modules that are used heavily from the higher planes, so most use is internal to the environment plane. Patterns, like method Y and the device/family kite, may become more common design notions for which it might be desirable to satisfy certain interfacing and use rules.

Walking Around

When browsing, a curious user typically zooms in on graphically distinct modules, seeking their names, other graphically conveyed properties, or to reveal and explore their inner workings. For example, zooming in on the upper four layers of the GUCS system (by suppressing all environment modules and their importing arrows) yields (after rotation and

translation) the back view and the realization that the method Y in the base plane is upside down (see Figure 3). The linear authoring top plane includes only one module not using anything; this top plane is the main module, recognized by its position in the structure. The other authoring module uses a collector in the user tasks plane (tip of the pie segment) and a collector module in the family plane (top of the kite). Using a collector can be viewed as the need to make dialogue (authoring) components aware of user tasks and device class plug-ins. All other modules in the user task plane are ordinary user-task definitions, each relying on two modules in the method plane (the base plane in Figure 3). These two modules, a general one (left) and a table one (right), seem to contain all the operations the user tasks might find useful.

Isn't such reliance on only two modules strange? Aren't user tasks supposed to rely on the devices in a family? The answer to both questions is yes. Therefore, we may have stumbled on a GUCS architectural design flaw. Inspecting GUCS source code reveals that the implementor of the generic user tasks decided to reference the product family's events and action types according to their string name. When GUCS generates code, each of the events and actions defined in the family/device plane are looked up by that name. Although this look-up operation does not impair the system's operation, this example of unnecessary duplication of identity might justify a GUCS redesign. The point is that we have detected a potential intermodule design flaw just by inspecting an abstraction of the system.

Figure 4 shows that all user tasks use two method modules—a General one and a Table one. As they both seem to be the primary method suppliers, we could ask ourselves whether the implementor has strictly separated methodological information in table form from its counterpart in function defini-

tion form. However, such separation is not the case; inspecting the code reveals that the method table is a supplier of device-independent user-interaction building blocks in table form that are part of the method, because these interactions are common to any product family. Since some parts of Figure 3 are not shown, due to the chosen viewing angle, examining an architecture involves active browsing of the model. A complementary view of the same 3D model, as in Figure 4, offers a view of the family-kite plane and its relations.

Experimentation with (even informal) reasoning

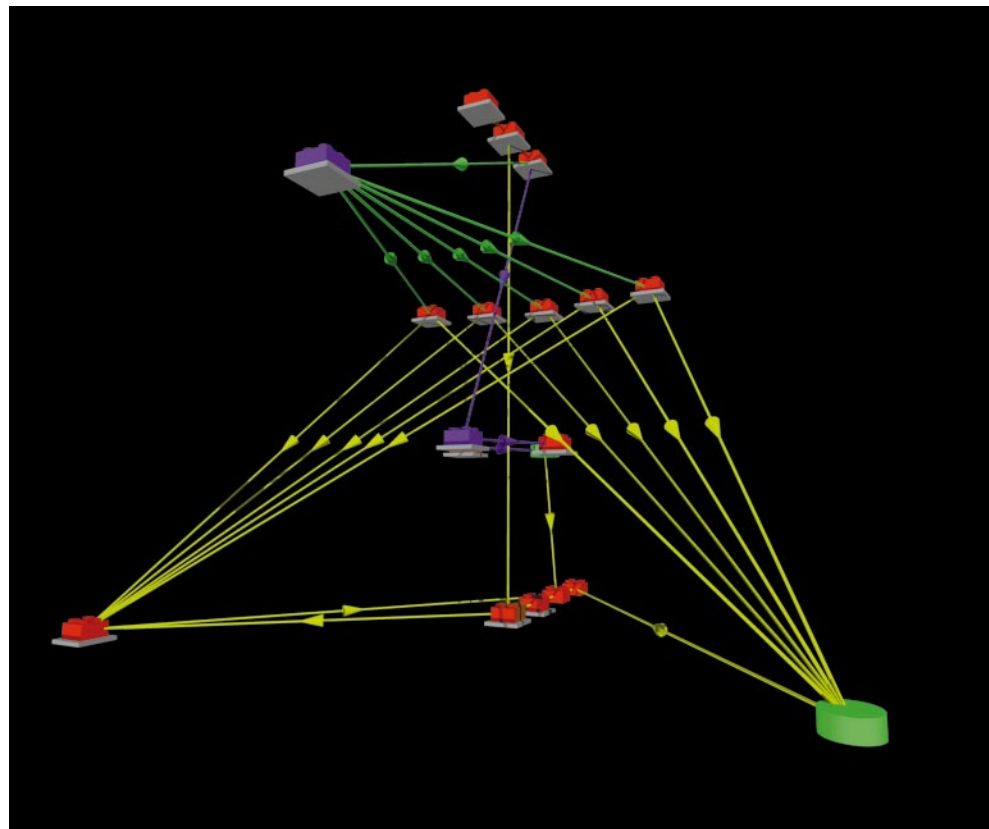


Figure 3. Back view of import relations, environment modules suppressed

about architecture on the basis of 3D abstracted pictorial information usually leads to the discovery of properties affecting relations among multiple modules. Whereas exploring 3D images seems useful for examining an architecture, system designers could also benefit from 3D before or during their design work.

More to Come

Walking through or rotating a complex design is exciting, almost like having the design in your hands and are options designers will increasingly want more of. However, as our example GUCS application

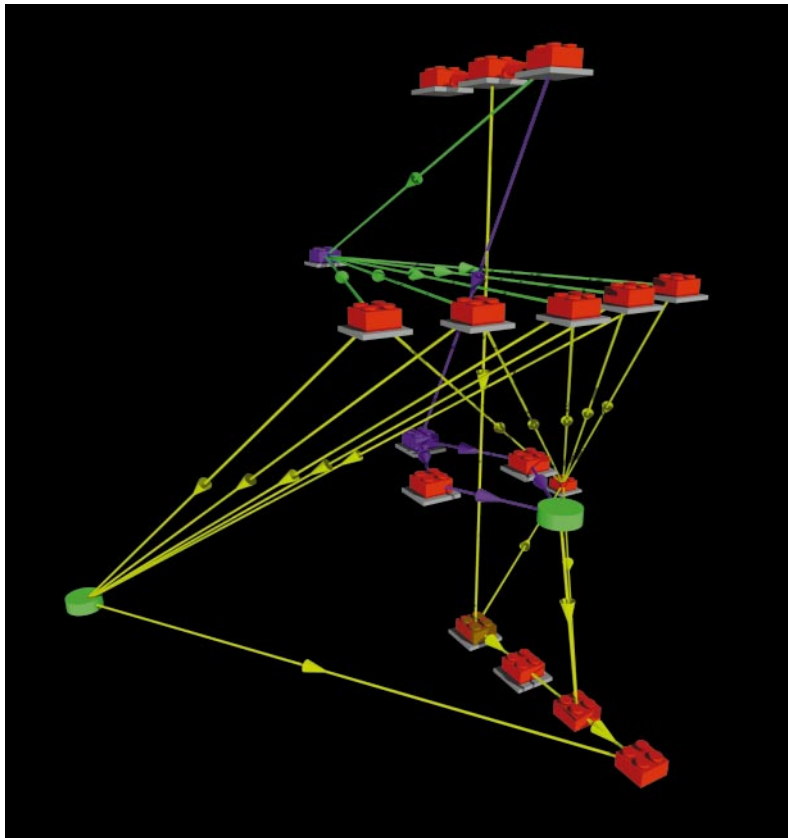


Figure 4. Front view of import relations, environment modules suppressed

does not yet fully exploit 3D visualization, we should also want to point out a few more techniques we've been experimenting with. One is to display several fundamentally different relations at the same time, so we can, for example, add a plane concerning a system's electrical aspects. This plane contains software-controlled hardware functions connected by green arrows for signal paths, red arrows for specific kinds of supply lines, and more. Other planes represent the layers of software control and the usual import relations connecting the software modules.

Another concerns checking specific rules about the organization of a file system. The files and directories appear as 3D shapes, connected by arrows for the links and directory-of relations. Shapes have different colors depending on whether the file or directory violates one or more of the rules. All design objects can be hyperlinked to other relevant design documents, to the object's own internal design world, even to sounds, movie fragments, and more. Our tool does not yet generate such links, but such features could be added (VRML supports links).

Our work is not the last word in design analysis

and presentation. But in view of the fast-changing world of virtual reality and because of the increasing importance of managing design complexity, we felt we had to present it as it is today. **C**

REFERENCES

1. Boasson, M. The artistry of software architecture. *IEEE Software* 12, 6 (Nov. 1995), 13–16.
2. Carmichael, I., Tzerpos, V., and Holt, R. Design maintenance: Unexpected architectural interactions. In *Proceedings of the International Conference on Software Maintenance (ICSM95)* (Nice, France). IEEE Computer Society Press, 1995.
3. Chen, Y., Nishimoto, N., and Ramamoorthy, C. The C information abstraction system. *IEEE transactions on Software Engineering* 16, 3 (March 1990), 325–334.
4. Feijs, L., Krikhaar, R., and Van Ommering, R. A relational approach to support software architecture analysis. *Soft. Prac. Exp.* 28, 4 (April 1988), 371–400.
5. Feijs, L., and van Ommering, R. The theory of relations and its applications to software structuring. IST report RWB-510-re-95011, Philips Research, Eindhoven, The Netherlands, 1995.
6. Harris, D., Reubenstein, H., and Yeh, A. Reverse Engineering to the Architectural Level. In *Proceedings of the 17th International Conference on Software Engineering (ICSE-17)*, ACM Press, New York, 1995.
7. Lipschutz, S. *Set Theory*. Schaum outlines series, Schaum Publishing Co., New York, 1980.
8. Murphy, G., Notkin, D., and Sullivan, K. Reflecting source code relations in higher-level modules of software systems. Tech. Rep. 94-09-03, Dept. of Computer Science and Engineering, University of Washington, Seattle, 1994; see www.cs.washington.edu/research/tr/techreports.html.
9. Plasmeijer, R., and van Eekelen, M. Concurrent Clean Language Report, Version 1.1, University of Nijmegen, The Netherlands, 1996; see www.cs.kun.nl/~clean.
10. QA Systems. *QAC TM 3.1 User Guide and Reference Manual*. Programming Research Ltd., 1993; see www.prqa.co.uk.
11. Schmidt, G., and Strohlein, T. *Relations and Graphs*. Springer-Verlag, New York, 1993.
12. Van Ommering R. *TEDDY User's Manual*. Tech. Rep. 12NC 4322 2730176 1, Philips Research, Dept. for Information and Software Technology, Eindhoven, The Netherlands, 1993.

LOE FEIJS (feijs@natlab.research.philips.com) is a senior scientist in Philips Research Laboratories, Eindhoven, The Netherlands, and a professor in the Eindhoven University of Technology
ROEL DE JONG (jongr@ce.philips.nl) is a scientist in Philips Research Laboratories, Eindhoven, The Netherlands.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.