The Ragnarok Software Development Environment

Henrik Bærbak Christensen

Department of Computer Science
University of Aarhus
DK-8000 Århus C, Denmark
hbc@daimi.aau.dk

Abstract. Ragnarok is an experimental software development environment that focuses on enhanced support for managerial activities in large scale software development, taking the daily work of the software developer as its point of departure. The main emphasis is support in three areas: *Management*, *navigation*, and *collaboration*. The leitmotif is the software architecture, which is extended to handle managerial data in addition to source code; this extended software architecture is put under tight version- and configuration management control and furthermore used as basis for visualisation. Preliminary results of using the Ragnarok prototype in a number of projects are outlined.

1 Introduction

Large-scale software development requires activities over a wide spectrum: In one end we find programming-near activities like implementation and debugging, in the other end managerial activities like release control, collaboration, project management, etc.

Ragnarok is a prototype software development environment that experiments with approaches focused on the managerial end of this spectrum. Often managerial aspects of the development process are neglected by developers in a hard-pressed development project: What pays the bills is the delivered system, and not accurately filled out time cards, proper software documentation nor a traceable evolution of the software. Still the accuracy by which these activities are performed is vital in insuring quality and overview in the development process. The goal of the Ragnarok project is that managerial activities become 'part-of' the every day development process instead of the often-felt 'stealing-time-from' the process.

The scope envisioned for Ragnarok is project teams of 2–6 persons and project sizes 2–20 man-years.

The purpose of this paper is to provide an overview of the Ragnarok environment and therefore the emphasis is more on examples than rigid formal descriptions. References will be made to more detailed and formal descriptions of the underlying models.

2 Motivation

Ragnarok limits its scope to three topics considered essential in successful software development:

- *Project- and source code management:* Managing a software project both means managing the project resources and tracing the evolution of produced software. In software production, development time is the critical resource and a work-break-down (WBD) structure is essential to organise and estimate planned and performed tasks. However, software designs evolve and it is therefore often a substantial effort to make sure that the WBD keeps pace with the design. This is a *tool gap* because different tools and procedures are used for programming in one end (programming environment/editors/etc.) and management (time cards/project management tools/etc.) in the other. Tracing the evolution is essential for two reasons: To provide accurate historic data for improving future development efforts; and to ensure the ability to recreate and compare milestones and releases accurately and swiftly.
- *Comprehension and navigation:* Overviewing and understanding large software systems and finding the correct piece of code in the thousands of files and libraries, are daunting tasks even in systems with a sound logical design. Explaining the design to newcomers can also be problematic [7].
- *Collaboration:* Software development is a team effort today. It is of vital importance that the team has a common understanding of the software and shares a reference frame in which the design can be discussed, documented, and reused. On a smaller scale, it is important that the team can collaborate on actual source code/document development without fearing loss of data and inconsistencies. Finally, other team members work in parallel and you have to be aware of their efforts to adjust your work situation accordingly.

The leitmotif in Ragnarok is the *software architecture* [30] or *logical design structure* [45]. By software architecture, we understand the hierarchy of abstractions that defines a logical software design. Abstraction and hierarchy are key concepts in designing, building, testing, and managing large software systems.

We believe it is possible to extend the scope of the architecture to management, navigation, and collaboration in a project. Many managerial tasks map well to the software architecture: For instance a reported bug can be associated directly to the class/library that contains it; the same goes for a change log describing how it was fixed, a time card listing the time spent on fixing it, etc. With respect to navigation the ability to locate pieces of code using the software design, instead of remembering long directory paths, is appealing. Collaboration is more naturally performed in terms of the architecture than in terms of loosely coupled sets of files.

The focus on managerial activities does not mean that programming-near activities are considered unimportant, but are currently outside the scope of the Ragnarok project. Ragnarok delegates programming tasks (edit/compile/etc.) to existing tools.

3 Proposal

The main contributions are the following proposals:

- *Annotated software architecture:* The proposal is to annotate the abstractions in a software architecture with additional information such as management data, work-break-down, reported bugs, test suits, release checklists, etc. Thereby the tool gap is minimised because design changes are automatically reflected in for instance the work-break-down structure.
- *Architectural software configuration management:* The proposal is a model for software configuration management (SCM) that minimises the gap between software design and configuration management by allowing developers to do configuration- and version control of the abstractions and hierarchy in the architecture. Furthermore, emphasis is put on traceability and reproducibility by unifying the concepts version and bound configuration.
- *Shared, activity mediating, software landscape:* Too often 'design' only exists in the minds of a few chief designers [7], or the documentation of it tends to be not 'quite' up-to-date. The proposal is to use modern graphical design notation like OMT or UML [43, 42] to create a visual, shared, design 'landscape' that mediates daily development activities like loading files into the editor, compiling, filling out time cards, logging changes, etc.

Finally, the Ragnarok project emphasises an experimental approach: Research prototypes have been implemented and are used in real development projects in order to verify the feasibility of the above proposals.

4 Annotated Software Architecture

In Ragnarok a design abstraction is embodied in a structured object denoted a **software component**. A software component has a name and a unique identity, CID. The physical implementation of an abstraction, a (possibly empty) set of code fragments, is stored in an attribute denoted the *substance*. For instance in a C++ project, a class foo could be represented by a software component `foo` with substance being the files {`foo.h`, `foo.cpp`}.

An abstraction is seldom an isolated entity but must be understood in its architectural context; abstractions are organised hierarchically by composition (aggregation/part-whole) and functionally interrelated by dependencies (association/use). Such *relations* are also stored as attributes in the software component; one for composition (a set of part/whole relations) and one for dependencies (a set of dependency relations.) Usually relations between components closely mimic the import/include declarations in the source code.

A software component also has a set of *annotations*. Each annotation is structured data for a specific dimension/aspect of the component. Examples include: Managerial annotations (like staffing: Who is responsible for implementing this component; budget: How many staff hours are budgeted for implementation, how many have been spent so far; estimated-time-to-complete etc.), quality assurance

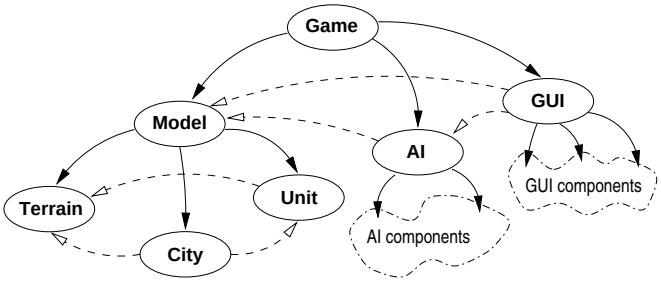

Fig. 1. Example of a software architecture. An ellipse represents an abstraction (software component) and the solid lines between ellipses are composition, dashed lines are dependencies. The 'clouds' denote unspecified sets of components.

annotations (checklists to be gone through in release situation, regression test suits), progress logs (what bug-fixes/enhancements have been carried out, by whom and when), etc.

Finally, a software component responds to messages and undergoes *transformations* like 'add substance', 'remove dependency, 'create part component', 'modify annotation', etc.

4.1 Graph Interpretation

The abstractions and hierarchy in an architecture are defined in terms of software components and their relations. This can be viewed as a directed graph: Components are nodes and relations are arcs. Fig. 1 exemplifies this by showing a feasible architecture for a small strategy game where a human player competes against computer controlled opponents for the control of a region of land, comprising various terrain and cities, by means of military units. The model has three major components: The underlying game play model, the artificial intelligence (AI), and the graphical interface (GUI). Our example concentrates on the model, therefore the model component is shown in more detail: Model is the class category/library containing classes for the fundamental game concepts: Terrain, City, and Unit. Composition is shown by solid lines while dashed lines indicate functional dependencies: Thus the library Model is composed of classes City, Unit, and Terrain, while City depends on Unit and Terrain, etc.

5 Architectural Software Configuration Management

Here we will briefly outline the software configuration management approach, denoted **architectural software configuration management**, used in Ragnarok. A more detailed description can be found in [9]. A formal description including consistency proofs and algorithms is provided in [11].

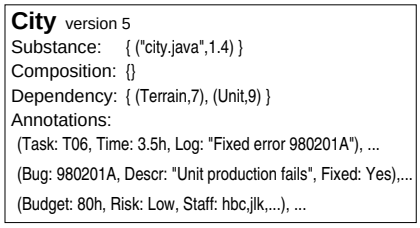

Fig. 2. Possible structure of software component City version 5 showing substance, relation sets, and some fictitious annotations.

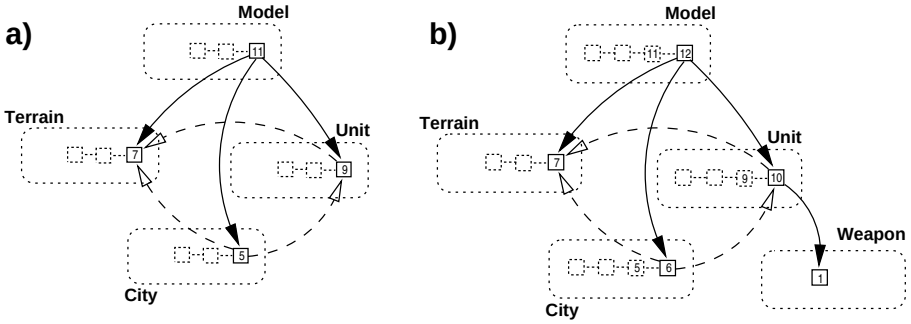

Fig. 3. Component 'Model' in version 11 (a) and version 12 (b).

5.1 Architectural Versioning

Conradi and Westfechtel [19] describe a SCM system as a combination of product space (often denoted 'workspace') containing evolving items, and version space (often: 'repository') storing states of the items evolution. In Ragnarok, the evolving item is the software component and each state is represented by a **software component version**—in essence reifying a version of an abstraction. A software component version is an immutable snapshot in time of the component: Its substance, relation sets, and annotations. The software component versions for a given component, CID, are arranged in a traditional version graph [48, 19].

An example of a fictitious internal structure of component City version 5 is shown in Fig. 2. Substance, relations, and example progress log-, bug report-, and budgeting annotations are hinted at.

An essential feature of the elements in the relation sets is that they are references to *specific* software component versions, not generic references. Figure 3 a) exemplifies this by showing component Model in version 11: The version group for a component is depicted as a box with rounded corners containing the component versions (small quadrant with the version number inside) organised in a version graph. Solid lines going out of a component version represent composition relations, dashed lines dependencies. Thus, for instance City version 5 does

not merely state a dependency to component Unit in general but to a specific version of it—namely version 9[1].

For the architecture to evolve, developers modify copies of component versions in a workspace (modifying substance and/or relations) before committing these to the version database.

Check-in and Check-out The architectural model emphasises bound configurations by means of a transitive closure check-in algorithm. To check in a new version, the algorithm recursively traverses all relation set references, depth-first, and creates new versions of all components along paths to modified components.

The check-out trivially reverses this process: The root component version is checked out, then the check-out is propagated recursively to all component versions referenced in the relation sets.

To illustrate this, consider a situation where an inner class, Weapon, is added to class Unit. After implementation and testing, a new version of Model is created in Fig. 3 b). The check-in is propagated to component Weapon, it substance stored and a new version identity, 1, established. Unit and City are also checked in; the new version of Unit adds a composition relation to Weapon, and City is indirectly modified as it lies on a path from Model to Weapon and must therefore update its dependency to Unit version 10. No new version of Terrain is necessary as it does not lie on a path to Weapon.

Versions are Configurations are Versions... As visualised in Fig. 3 component versions and the relations between them can also be viewed as directed graphs: Components versions are nodes and the references in the relation sets are arcs. Any component version is root in such a directed graph identifying a bound configuration of the abstraction and its context of relations and related abstractions at the time of check-in. Thus, the concepts version and bound configuration are unified. Even complex configurations are identified by a single component version, like e.g. 'Model version 12' in Fig. 3 b). Thus, configurations are first class objects and the evolution of configurations is trivially recorded and accessible.

Another important consequence of having specific references to component versions in the relation sets is that the software architecture itself is under strict version control. An *architectural diff* algorithm can recursively compute differences in the relation sets between two versions of a component and report abstractions added, deleted, or moved, and changed dependencies. This provides better overview of architectural changes than the traditional simple list of file contents differences.

5.2 RCM Prototype

The architectural software configuration management model is implemented in a subsystem in Ragnarok. This subsystem has been equipped with its own, sim-

[1] Hence the previous Fig. 1 was actually inaccurate because it showed generic relations.

ple, textual interface in a prototype called 'RCM'. RCM is inspired by the UNIX shell: The user can issue commands like check-in and -out, architectural diff's, display architecture and version history, etc., to the component which is current, just as commands in a UNIX shell affects the current directory. A `cd` like command is available for navigating the component architecture.

The current prototype implements component substance as a set of traditional files and delegates file-level versioning to RCS [47] so a substance snapshot is a simple set of pairs: (filename, RCS revision number).

A quick reference guide for RCM is provided on WWW [10].

5.3 Experience

RCM is currently used in three, real, small- to medium-sized projects whose main characteristics are given below.

	ConSys	*BETA Compiler*	*Ragnarok*
Used since	Mar. 96	Feb. 97	Feb. 96
Data	C++, SQL, binary	BETA, C, html	BETA, LaTeX
Platform	NT	Unix, NT	Unix, NT
No. developers	3	4	1
No. components	110	36	36
No. files	1070	250	160
No. lines (KLOC)	185 + binary	110	40

Table 1. Main characteristics of on-going experiments.

The main source of data is guided, open-ended [39] interviews of the developers on the BETA compiler [6] and ConSys [26] projects. A secondary source of data is automatically generated usage logs from RCM that have been analysed by simple statistical methods.

Results The results can be summarised as follows:

Model 'feels' natural: The user groups readily accept the software component to represent design entities and claim a close, if not one-to-one, mapping between their design and their software component structure. They 'think' SCM in terms of components rather than files/directories. This claim is supported by the usage logs data where file related commands are seldom used.

Emphasis on bound configurations: While this is considered important for release and milestone management, the developers more emphasised the feeling of 'security' in the daily development cycle as backtracking to working configurations is easy.

Traceable architectural evolution: The emphasis on tracing architectural change was valued; for instance the ConSys project has more than tripled its size in terms of number of components and files during the reported period.

Additional details can be found in [12].

5.4 Discussion and Related Work

The emphasis on bound configurations does not prohibit using selection rules to create new configurations in workspace, but rule based selection plays a much lesser part. For instance, RCM provides the inevitable 'get latest versions' rule as the only addition to the standard check-out procedure. Instead developers identify and communicate stabile libraries and subsystems through version identifications.

The current implementation of the substance attribute as a set of files makes handling fine-grained abstractions (like individual methods in a class) infeasible. A future implementation could overcome this limitation by implementing substance as sets of small code fragments in a program database instead.

The emphasis on architecture and bound configurations is similar to the ideas in COOP/Orm [34, 35, 37] and POEM [33, 32]. The prime difference is research focus: Where COOP/Orm and POEM have focused on supporting fine-grained abstractions but have not been used in real-life projects, the RCM prototype is more coarse-grained (using traditional files) but is used 'for real' as described in section 5.3. As such, we feel that our work complements and adds credibility to COOP/Orm and POEM by reporting that architectural models are feasible in practice.

Many traditional SCM systems [15, 41, 16, 8, 36, 5] rely on labels or *tags* for defining bound configurations: When creating, say a milestone, all file versions defining the milestone are tagged with a symbolic name. While tagged file version based systems are easy to understand, they suffer from a number of conceptual problems: Tagged file versions do not convey information about the evolution of the software architecture itself (e.g. how files and libraries are added or removed); and conceptually a tag is an *is-used-in* relation (stating that 'this version of this file *is-used-in*, say, release 4 of our system')—however developers more naturally think in terms of *uses* relations like: Release 4 *uses* graphics library version 1.14, which *uses* the window class version 1.22 etc.

Bendix' three-dimensional configuration management [4, chap. 6] also contains elements similar to our work. A crucial difference is that Bendix' model uses generic relations. Thus, bound configurations must be expressed as a set of selection rules that in our opinion leads to the tagging technique resulting in the same problems as mentioned above.

6 Software Architecture Visualisation

This section outlines the visualisation part of Ragnarok. A more detailed treatment can be found in [14] while an overview and reference guide for the Ragnarok prototype is provided on-line on WWW [13].

The fundamental purpose of the visualisation model is to make the architecture visually manifest in what is termed the **software landscape**. The landscape is shared and mediates development activities: 'Shared' because every developer in the team views and manipulates the same landscape, and 'mediating' because

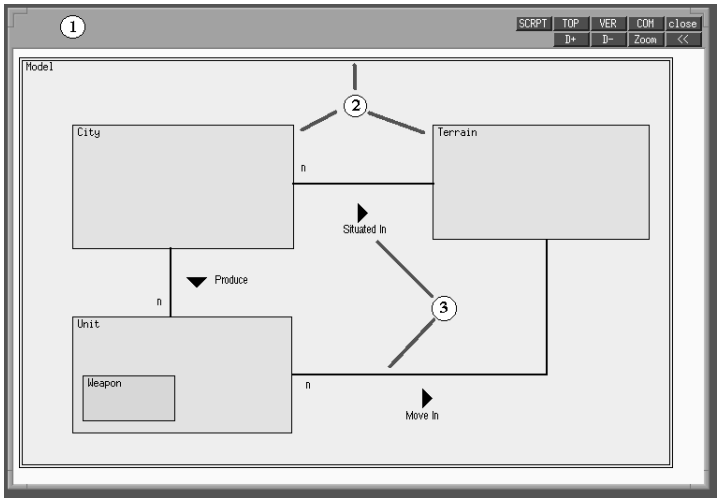

Fig. 4. A Ragnarok map (1) showing landmarks (2) and decorations (3) for the example game model design.

daily development tasks are performed by interacting directly with objects representing abstractions in the landscape.

The Ragnarok visual model is based on a *geographic space metaphor* [29, 20]. Objects in a geographic space are characterised by fixed positions over a long time-scale, like for instance buildings, trees, and streets in a city; as opposed to objects in a *desktop space* that are constantly moved around. Humans are apt at navigating in a well known physical environment: As Kuhn and Blumenthal note: 'Perception, manipulation, and motion in space are largely subconscious activities that impose little cognitive load while offering powerful functionality' [29]. By providing a geographical layout of the software design, we believe that humans fine sense of locality can be utilised.

6.1 Visual Model

The *landscape* is an infinite two-dimensional plane. The landscape serves as a space for geographically organising *landmarks* and *decorations* as exemplified in Fig. 4.

A landmark occupies a well-defined region of the landscape and represents a software component (and thereby an abstraction in the software architecture). The hierarchical structure is visualised by *spatial containment* i.e. the landmarks associated with components that are part of a component A are positioned inside the landmark of A. Fig. 4 shows how the core game model landmarks are nested inside the 'Model' landmark.

A decoration is simple graphics, like text, lines, polygons, images, etc., positioned in the landscape. Specifically the Ragnarok prototype provides basic

support for UML notation, the unified modelling language [42]. Thereby the software design can be further documented by stating associations, multiplicity, roles, etc., as seen on Fig. 4.

The landscape is not directly accessible but viewed and manipulated through *maps*. A map visualises a region of the landscape on the computer display. The region displayed is determined by the map's *view-parameters*: (O, w, h, s, A). O is the position of the map's top left pixel projected onto the landscape, w and h the map's physical (pixel) width and height, s the map's scale, and A the map's aspect. Hence a map w pixels wide and h pixels tall will display region $(O.x, O.y, O.x + ws, O.y + hs)$ of the landscape. This is denoted the *displayed region*.

The *aspect* A determines the appearance of landmarks in the displayed region by processing a subset of the data in the associated software component. For example in a management aspect map the management annotation data of a component may be processed to yield a colour code for estimated-time-to-complete, which is then used as the background colour for the landmark.

Global context Any number of maps can be created showing regions of the landscape in various aspects. However, many maps do not provide overview nor global context: How are the displayed regions positioned relative to each other? Therefore, a *map outline* is introduced. A map outline is a projection of the displayed region of one map (*detail* map) onto another map dedicated for showing global context (*world* map). To distinguish outlines in the world map, a given outline and the frame of the corresponding detail map have identical colours. The outline and the displayed region in the corresponding map are synchronised so any change in either of them is immediately reflected in the other.

Interaction The interaction model employed is *direct manipulation* [44, 24]. Maps, landmarks, and outlines can be moved, resized, and zoomed using simple mouse manipulations. Landmarks also mediate actions from the user to their associated components: For instance to compile a component, the user clicks the landmark bringing up a menu where she can select the compile action directly. The set of actions available on any given component depends on the aspect of the map it is shown in; e.g. in a version control map landmarks mediate actions like check-in and check-out to the underlying components, in a management aspect map the user can edit task lists, log spent staff hours, etc.

6.2 Ragnarok Prototype

Figure 5 shows a snapshot of the Ragnarok prototype. The example project is the Ragnarok project itself whose main characteristics were outlined in table 1.

The Ragnarok window is divided into four parts: In the upper left corner is the *world map* (1) with *outlines* (9) of open detail maps. The world map has a fixed position in the Ragnarok window for easy reference and overview. The lower left part contains the *log window* (2), which is essentially a running log of

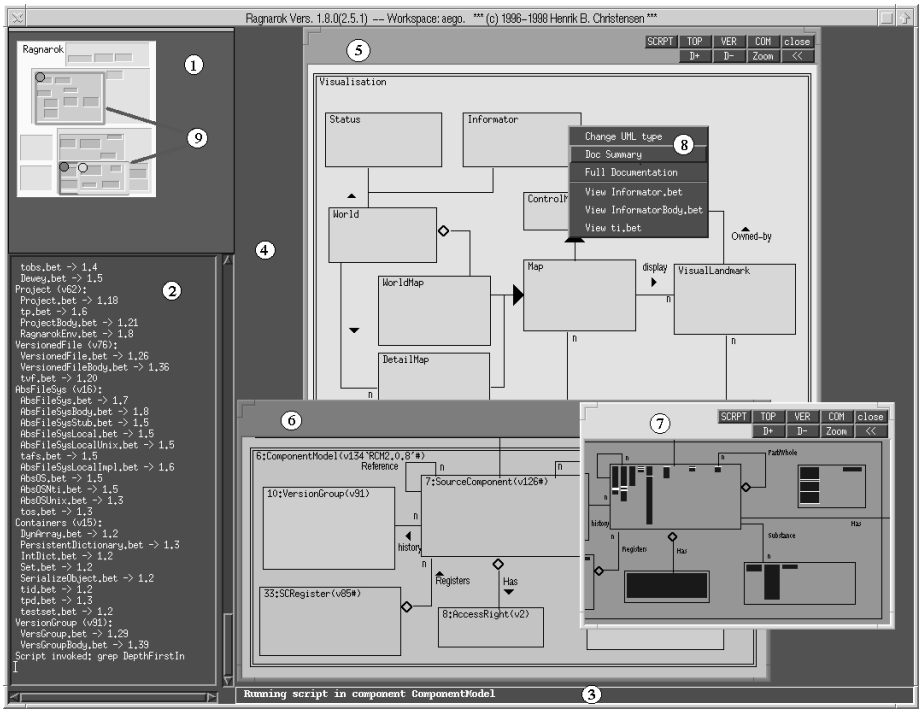

Fig. 5. Overview over the Ragnarok prototype window. The numbered parts are explained in the text. Note: The image in this and the following figures are best viewed in colour. Colour versions of the images can be found at http://www.daimi.aau.dk/~hbc/papers/nwper-images.html.

important operations, here a version control check-out operation. The bottom right corner contains the *status bar* (3), which displays warnings and status information. On the right is a large area (4) in which may reside multiple *detail maps* (5–7). Landmarks are generally displayed as simple, coloured, rectangles containing component name and possible additional information. Clicking any landmark brings up a context-sensitive menu (8) that lists available actions on the underlying component.

Version control An important collaboration aspect in version- and configuration management is to enable the individual developer to overview how his/her private copy of source code relates to the overall project code. A typical question is: 'Do I have the newest version of library X?'

The version control aspect visualises this in a compact form. Colour coding of landmarks is used to show the state of the developers local copy. Referring to Fig. 6 light red (medium gray on the figure) indicates components where newer versions exist. Gray (hatched) indicates that the local source code matches the newest. The colours light green (dotted) and bright red (dark gray) are used to

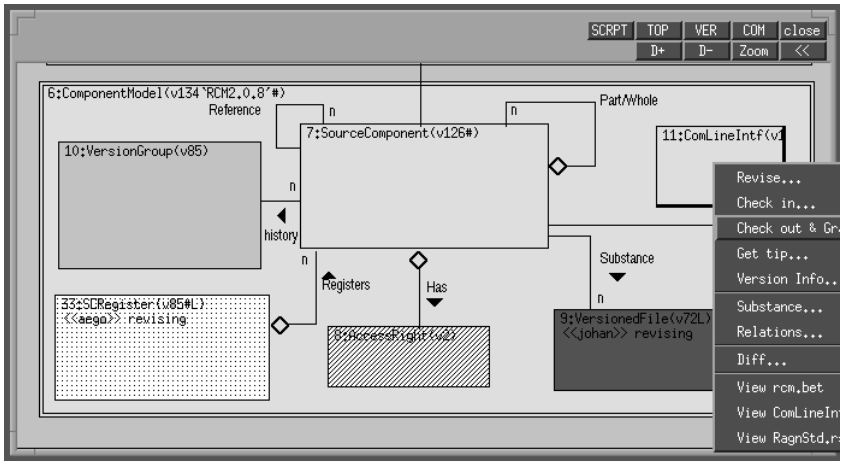

Fig. 6. A map showing the version control aspect.

convey information about currently ongoing work: Light green indicates that the developer himself is currently editing information in the component, bright red that some other, named, developer is working on it, and thus warns about potential conflicts if the developer decides to edit this component as well. Light yellow (light gray) indicates indirect changes because components depended upon have changed.

The context sensitive menu, half-visible, allows version control commands, check-in and -out, display version graph, source code access, etc., to be issued to the individual components.

Progress is instantly reflected in all running Ragnarok instances and thus the evolution of the software system is visible on-line: For instance if developer 'johan' checks in component 'VersionedFile' in Fig. 6 the component will immediately turn light red to indicate that a new version is available.

Topography This aspect minimises the amount of information presented to provide a compact overview, refer to the world map (1) in Fig. 5. Decorations are not shown, and landmarks are grayed according to their nesting levels and without text. The context-sensitive menu lists (source) files in the component and choosing one loads the file into an editor.

This aspect is the standard one for the world map providing convenient overview of the project landscape and a neutral background for outlines. An unanticipated but strong feature of the topography aspect world map is that in essence it is a compact and fast file browser: Any of the 160 source files in the Ragnarok project can be loaded into our emacs editor using *one* mouse-click in the world map (clicking the landmark brings up a pop-up menu with file names, releasing the mouse button over the wanted file tells emacs to load it.)

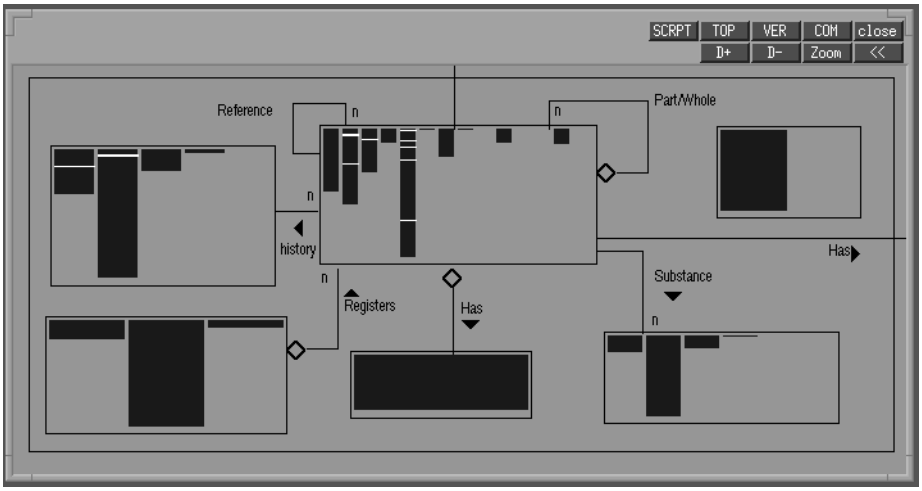

Fig. 7. A map in visual scripting aspect, showing the result of a grep for the string 'GetLockOwner'.

Visual script This aspect allows users to run scripts, written in the interpreted language TCL [38] on (parts of) the project and interpret the result spatially and visually. User actions like mouse clicks on landmarks or positions also result in user defined TCL functions being called.

As an illustration Fig. 7 shows the output of a `grep` script written in TCL. The user has supplied the search string 'GetLockOwner' as argument, and the TCL script is then executed on all files in all components in the selected configuration (here the same part as in Fig. 6.) The result is then visualised: The interior of landmarks is filled with black bars, each one represents a single file in the component. The bar height is a relative measure of the file size measured in lines. Each red line (white on the figure) in a bar shows that the search string occurs in the file at this relative position.

Furthermore, additional TCL scripts are bound to mouse events: Clicking and holding down the left mouse button near a red line pops up a text viewer displaying 20 lines around the position where the search string occurs in the file—releasing the mouse button again makes the text viewer disappear. This way one can quickly browse the occurrences and their immediate context without polluting the screen with numerous new windows. Double clicking a red line automatically loads the file into the editor centred on the matching line.

This visualisation of a recursive grep is compact and provides better overview than traditional textual recursive grep's. Furthermore the clustering, density, and distribution of red lines in itself give important information. For instance grepping for a function or class name may show misuses ('Now, why is there a call in the GUI library?') or high coupling ('Hey, look, this class pops up in every component in the system!') that are easily missed in a 300 line textual output.

By basing this aspects on user written scripts, Ragnarok provides a degree of tailorability to the context of a given project as developers can write scripts that provide custom visualisations. The TCL language has strong support for file handling and invoking external programs and it is therefore relatively simple to parse files (as done in the grep case above), or invoke profilers, run regression tests, extract relevant data from a project database, etc., and visualise the results of such external processing. The ability to bind user written TCL scripts to mouse events makes custom visualisations direct manipulable: Clicking a landmark that highlights an unsuccessful regression test run can load the test into an editor; clicking a landmark with project data can instruct the database to load the proper table/view, etc.

6.3 Discussion

Having a shared, mediating, design landscape has a number of benefits:

- *Overview and navigation:* Intangible 'design' is reified into concrete, manipulable, objects and overview and navigation is supported by tapping into humans fine spatial and visual perception. Focus shifts from traditional name-based search (remembering a sequence of directory names to find a file) to location-based search: Developers instead learn *where* the component containing relevant information is located.
- *Mediating, up-to-date, software design:* The software landscape becomes the focal point of daily activities: Editing a file in a component, compiling a library, editing a task list, reporting a bug, checking in a change, etc., are all performed directly through context-sensitive menus of the landmarks. This is direct and intuitive and the pressure to ensure a correct, up-to-date, design landscape is strengthened.
- *Collaboration:* A shared landscape provides a common reference frame for developers, managers, testers, and maintainers alike, easing discussion and allows everyday navigation language like 'look to the left' to be used when helping in locating components. The landscape is shared meaning progress and changes are immediately reflected in every team member's view of the landscape.
- *Visualisation framework:* Data in a software project is inherently multi-dimensional and traditionally these different dimensions are handled by different tools and organisational procedures and presented in many different formats. The software landscape provides a stable, unifying visualisation framework that allows diverse information to be visualised overlaid over the design landscape.

The different aspects described in section 6.2 of course only serve as a sample of uses. An important aspect currently not implemented is a management aspect. It is envisioned that developers log development progress and report actual work hours spent directly on the components. Thereby a management aspect map can on-line report estimated-time-to-complete or actual versus budgeted resource

allocations for instance using a colour coding scheme for the landmarks, and thereby highlight trouble spots.

Many other aspects can be envisioned: Defect reporting, possibly determined by automatically running regression test suits on modified components; visualising profiling information to identify bottlenecks in a system; release control, highlighting components with un-passed tests, etc.

6.4 Related Work

Pad++: Pad++ [3, 2, 40] is an innovative and powerful 2D visualisation system in which a user manipulate objects on an infinitely zoomable 2D surface and incorporates a very effective engine for panning and zooming. The underlying visual model employed in Ragnarok is similar to Pad++. The difference is the objects handled. In Pad++, objects reside directly on the Pad surface; in contrast landmarks serve as visual *representations* of the complex, multi-dimensional, data of the underlying component; they *are* not the actual data. Therefore, Ragnarok uses the *same* region to visualise *different* data in different aspect maps (like grep matches or version information). This aspect property of Ragnarok is essential because of the multi-dimensional nature of software.

CASE tools: Looking at e.g. map (5) in Fig. 5 Ragnarok may resemble a traditional CASE tool or UML diagram editor. However, the focus of CASE tools and Ragnarok is different. CASE tools are generally analysis and design tools with code generation features i.e. with strong support and focus on the early phases of a project and the programming task. The emphasis in Ragnarok is foremost on making design abstractions manifest and manipulable by assigning a geographic location and secondly on documenting the design using some notation like UML; once this is done it is used as a visualisation framework for project data which CASE-tools traditionally do not address.

7 Summary

We believe that the combination of annotated software architecture, an architectural SCM model, and the geographic space visual model provides support for the problems outlined in section 2:

- *Project- and source code management:* The ability to annotate the software architecture addresses the tool gap: We believe many tasks like documentation, handling time cards, bug reporting, and managing the work-breakdown structure can be integrated into the architecture which strengthens these tasks as being 'part-of'- instead of 'stealing-time-from' software production. Having a shared annotated architecture means that progress can be monitored in real time which is beneficial to both managers and developers. The underlying architectural configuration management model ensures tight version- and configuration control of the software architecture. The fact that annotations are version controlled as well means that very accurate historic data about the process can be extracted and analysed.

– *Comprehension and navigation:* The shared software landscape provides overview and documents the software design in the daily development environment. Therefore we believe the pressure is strengthened to ensure up-to-date diagrams. Location-based navigation is a strong alternative/supplement to traditional name-based search. Many different aspects of the software structure can be visualised overlaid on the same, stable, landscape easing comparisons; and navigational knowledge obtained performing one task can be carried on to the next.
– *Collaboration:* The software design landscape provides a common reference frame within the team and across different areas of expertise. The landscape immediately reflects changes to the underlying software architecture, like e.g. new versions, added/deleted abstractions etc, and thus provides overview over development progress. The underlying software configuration management model allows sharing and collaboration on source code and eliminates data loss and inconsistencies.

Kruchten advocates multiple views on architecture in the $4 + 1$ view model [28] including the logical, process, physical, and development view. Ragnarok is clearly based on the logical view and regards the development view as attributes of the logical, through the substance attribute of software components. One of the reasons that Kruchten needs a separate development view is indeed release- and configuration management concerns, a problem we believe our model overcomes. Still, aspects like process- and physical view as well as for instance the planning phase where one needs to view tasks on a time scale, are not supported and cannot be handled nor visualised well. However, we believe there are enough interesting and important aspects to make the approach worthwhile.

Navigation using the presented visual model relies on a stable landscape that is well known to the developers: If the landscape is constantly modified, spatial knowledge is virtually non existing—compare the effort of driving home from the office every day with the effort of driving in an unknown city. Consequently, navigation will become easier as the design stabilises. During the initial analysis and design phases, ideas and abstractions are fostered and discarded rapidly meaning many changes in the landscape; in these phases, a more traditional, search-based, navigation mechanism is a beneficial supplement.

8 Status and Future Work

The status of the Ragnarok project is:

– The architectural software configuration management model has proven to work well in the RCM prototype tool as described in section 5.3 and detailed in [12].
– The visual model, implemented in the graphical Ragnarok prototype, is currently used in the Ragnarok project itself and in a small sub-project in the Mjølner BETA system [1]. Preliminary experience is encouraging, especially concerning location-based navigation as described in section 6.2. It is

planned that the ConSys and BETA compiler groups will shift from RCM to the visual Ragnarok prototype during 1998.
- The annotation support is currently rather limited and therefore no significant experience can be reported yet.

Future work proceeds in a number of directions. A prime concern is to merge the current efforts in rather different areas, software configuration management (section 5) and visualisation (section 6), into a more cohesive whole. This will entail problems like how to visualise version graphs and compare ('diff') configurations using the geographic visualisation model, and how to version control the landscape itself. It is also important to verify the feasibility of an annotated software architecture: The user groups report a need for supporting progress tracking and bug report handling which are good candidates for initial support. This in turn requires more operational aspects to be implemented for visualising such information.

A lot of work also remains within the individual areas: The architectural software configuration model needs improved support for collaboration which we plan to do using 'cooperative versions' [22] i.e. 'micro'-versions intended for personal, daily, development activities. Concerning the visualisation model, a major effort is to design a flexible, user definable, aspect model that allows tailoring map aspects and visualisations to the need of the individual organisation and developer.

References

1. P. Andersen, L. Bak, S. Brandt, J. L. Knudsen, O. L. Madsen, K. J. Møller, C. Nørgaard, and E. Sandvad. The Mjølner BETA System. In *Object-Oriented Environments - The Mjølner Approach* [27], pages 24–35.
2. B. B. Bederson and J. D. Hollan. Pad++: A Zooming Graphical Interface for Exploring Alternate Interface Physics. In *Proceedings of ACM UIST '94*. ACM Press, 1994.
3. B. B. Bederson, J. D. Hollan, K. Perlin, J. Meyer, D. Bacon, and G. Furnas. Pad++: A Zoomable Graphical Sketchpad for Exploring Alternate Interface Physics. *Journal of Visual Languages and Computing*, 7:3–31, 1996.
4. L. Bendix. *Configuration Management and Version Control Revisited*. PhD thesis, Institute of Electronic Systems, Aalborg University, Denmark, Dec. 1995.
5. B. Berliner. CVS II: Parallelizing Software Development. In *USENIX*, Washington D.C., 1990.
6. http://www.daimi.aau.dk/~beta/.
7. U. Bürkle, G. Gryczan, and H. Züllighoven. Object-Oriented System Development in a Banking Project: Methodology, Experience, and Conclusions. *Human-Computer Interaction*, 10:293–336, 1995.
8. *PLATINUM CCC/Harvest Users Guide*.
9. H. B. Christensen. Context-Preserving Software Configuration Management. In Conradi [18], pages 14–24.
10. H. B. Christensen. *RCM 2.5 Quick Reference*. Department of Computer Science, University of Aarhus, 1997. http://www.daimi.aau.dk/~hbc/Ragnarok/rcm_quickref.html.

11. H. B. Christensen. A Formal Model for the Architectural Software Configuration Management Model. Technical report, Department of Computer Science, University of Aarhus, 1998. To appear in DAIMI PB series.
12. H. B. Christensen. Experiences with Architectural Software Configuration Management in Ragnarok. In *Proceedings of SCM-8: International Symposium on System Configuration Management*, Lecture Notes in Computer Science, Brussels, July 1998. Springer Verlag.
13. H. B. Christensen. *Ragnarok: Overview and Reference Guide Version 1.5.* Department of Computer Science, University of Aarhus, 1998. http://www.daimi.aau.dk/~hbc/Ragnarok/ragn_doc.html.
14. H. B. Christensen. Utilising a Geographic Space Metaphor in a Software Development Environment. In *Proceedings of EHCI'98, IFIP Working Conference on Engineering for Human-Computer Interaction*, Crete, Greece, Sept. 1998. Chapman and Hall.
15. http://www.rational.com/products/clearcase/.
16. Pc-based version control. http://www.silcom.com/~alobba/pc_vc.html.
17. R. Conradi, editor. *Software Configuration Management*, Lecture Notes in Computer Science 1235. ICSE'97 SCM-7 Workshop, Springer Verlag, 1997.
18. R. Conradi, editor. *Supplementary Proceedings: 7th International Workshop, SCM7*, May 1997.
19. R. Conradi and B. Westfechtel. Towards a Uniform Version Model for Software Configuration Management. In Conradi [17].
20. M. J. Egenhofer and D. M. Mark. Naive Geography. In Frank and Kuhn [23], pages 1–15.
21. J. Estublier, editor. *Software Configuration Management*, Lecture Notes in Computer Science 1005. ICSE SCM-4 and SCM-5 Workshops, Springer Verlag, 1995.
22. J. Estublier and R. Casallas. Three Dimensional Versioning. In Estublier [21], pages 118–135.
23. A. U. Frank and W. Kuhn, editors. *Spatial Information Theory / A Theoretical Basis for GIS*. COSIT '95, Lecture Notes in Computer Science 988, Springer-Verlag, 1995.
24. E. L. Hutchins, J. D. Hollan, and D. A. Norman. Direct manipulation interfaces. In D. A. Norman and S. W. Draper, editors, *User Centered System Design*, chapter 5. Lawrence Erlbaum, 1986.
25. IEEE Computer Society Press. *Proceedings of the 18th International Conference on Software Engineering*, 1996.
26. ISA. Consys. http://isals.dfi.aau.dk, 1996. ISA: Institute for Storage Ring Facilities, University of Aarhus.
27. J. L. Knudsen, M. Löfgren, O. L. Madsen, and B. Magnusson. *Object-Oriented Environments - The Mjølner Approach*. Prentice-Hall, 1993.
28. P. B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, Nov. 1995.
29. W. Kuhn and B. Blumenthal. *Spatialization: Spatial Metaphors for User Interfaces*. Geoinfo-Series, Department of Geoinformation, Technical University, Vienna, 1996. Reprinted tutorial notes from CHI'96.
30. D. A. Lamb. Introduction: Studies of Software Design. In *Studies of Software Design* [31].
31. D. A. Lamb, editor. *Studies of Software Design*, Lecture Notes in Computer Science 1078. ICSE'93 Workshop, Springer Verlag, 1996.
32. Y.-J. Lin and S. P. Reiss. Configuration Management in Terms of Modules. In Estublier [21].

33. Y.-J. Lin and S. P. Reiss. Configuration Management with Logical Structures. In *Proceedings of the 18th International Conference on Software Engineering* [25], pages 298–307.
34. B. Magnusson and U. Asklund. Fine Grained Version Control of Configurations in COOP/Orm. In Sommerville [46], pages 31–48.
35. B. Magnusson, U. Asklund, and S. Minör. Fine Grained Revision Control for Collaborative Software Development. In *ACM SIGSOFT'93 - Symposium on the Foundations of Software Engineering*, Los Angeles, California, Dec. 1993.
36. Microsoft (R) Corporation: Visual SourceSafe. http://www.microsoft.com/ssafe/.
37. S. Minör and B. Magnusson. A model for Semi-(a)Synchronous Collaborative Editing. In *Proceedings of Third European Conference on Computer-Supported Cooperative Work - ECSCW'93*, Milano, Italy, 1993. Kluwer Academic Press.
38. J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley Professional Computing Series, 1994.
39. M. Q. Patton. *Qualitative Evaluation Methods*. Sage Publications, Beverly Hills, Calif., 1980.
40. K. Perlin and D. Fox. Pad - An Alternative Approach to the Computer Interface. In *Proceedings of ACM SIGGRAPH '93*. ACM Press, 1993.
41. http://www.intersolv.com/products/pvcs-vm.htm.
42. Unified Modeling Language, version 1.0. Rational Software Corporation, Santa Clara/CA, Jan. 1997. URL:http://www.rational.com.
43. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall International Editions, 1991.
44. B. Shneiderman. Direct Manipulation: A Step Beyond Programming Languages. *IEEE Computer*, Aug. 1983.
45. I. Sommerville. *Software Engineering*. Addison-Wesley Publishers Ltd., 4 edition, 1992.
46. I. Sommerville, editor. *Software Configuration Management*, Lecture Notes in Computer Science 1167. ICSE'96 SCM-6 Workshop, Springer Verlag, 1996.
47. W. F. Tichy. RCS – A System for Version Control. *Software – Practice & Experience*, 15(7):637–654, July 1985.
48. W. F. Tichy. Tools for Software Configuration Management. In Winkler [49].
49. J. F. H. Winkler, editor. *Proceedings of the International Workshop on Software Version and Configuration Control*. B. G. Teubner, Stuttgart, Jan. 1988.