Directions in Modelling Large-scale Software Architectures

John Grundy and John Hosking

Department of Computer Science, University of Auckland
Private Bag 92019, Auckland, New Zealand
{john-g, john}@cs.auckland.ac.nz

Abstract

Modelling the software architectures of large systems is a challenging task. A wide variety of abstractions are required by developers to assist them in describing and analysing such architectures, including information about the components and relationships that comprise a system, the static and dynamic structure of the system, and the behavioural responsibilities of components in the system. In addition, good tool support is needed to provide modelling, analysis, design and code generation, and reverse engineering facilities. This paper analyses several architectural modelling approaches and their tools. Deficiencies with these current approaches are used to motivate a synthesised modelling approach and appropriate tool support.

1. Introduction

Software Architecture has become a major field of research and practice in Software Engineering. Topics of particular interest to both researchers and practitioners include architecture modelling and analysis, Architecture Description Languages (ADLs), architecture styles, design and implementation of software architectures, especially middleware technologies, architecture reverse engineering, and formal specification and refinement of software architectures [1, 14, 16, 17, 19, 22]. Developing system designs and implementations based on a formalised Software Architecture better enables developers to meet functional and especially non-functional requirements, allows other developers to more easily understand the rationale behind a system design, facilitates reuse of high-level architectural abstractions and "best practices", and provides a more structured development process for large systems and development teams.

Of particular interest to us has been the modelling, analysis and implementation of software architectures, particularly those for component-based, collaborative systems [6, 7, 8]. Key issues we have encountered to do with Software Architecture in such systems include:

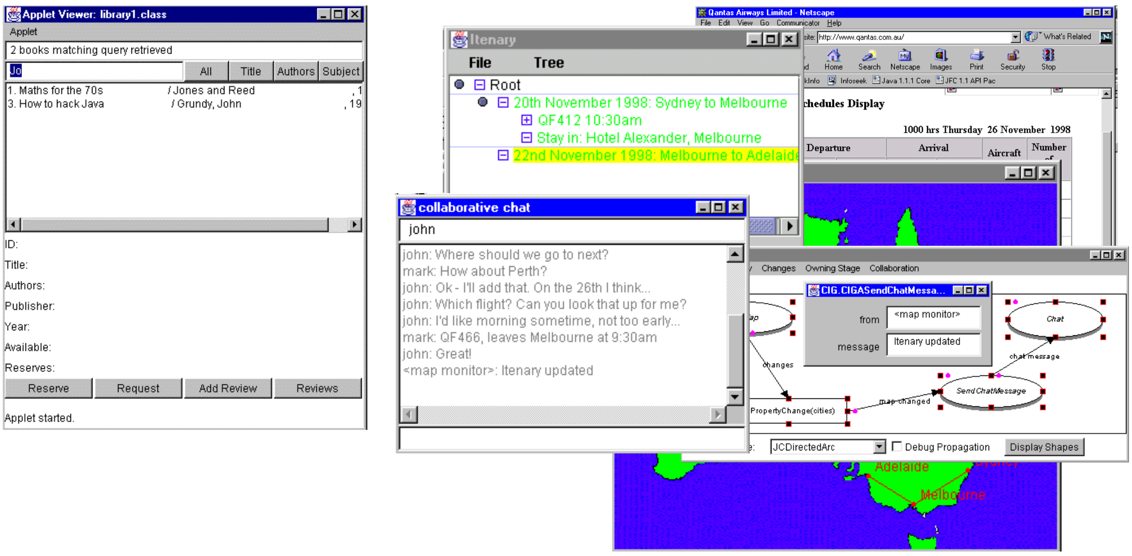
- Architecture Modelling. Determining the parts (components) the system is to be slit into, what data/functions these embody, how these components are connected, and static vs. dynamic behaviour of the components. Visualising such information about a system is crucial to understanding and developing a system based on the architecture [6].
- Architecture Analysis. After determining and modelling a system's architecture, it is necessary to reason about its expected performance, quality of service, cost, maintainability, etc. [9, 10].
- Architecture Refinement. Developers need to design solutions based around the architectural abstractions embodied in a software architecture model, implement these designs using suitable middleware technologies and evaluate the resultant system [7, 11].

This paper investigates these Software Architecture modelling and analysis issues with respect to current modelling approaches. The following section introduces two e-commerce applications and an example Software Architecture which might be chosen to realise these systems. Key requirements when modelling, analysing and refining such Software Architectures are outlined in the following section. Several brief case studies review some Software Architecture modelling techniques. A critique of these modelling techniques and their support tool's modelling, analysis and refinement capabilities is given. The final sections present a proposed synthesis of these Software Architecture techniques into a new modelling approach for large-scale system architectures. A brief outline of necessary tool support is presented.

2. An Example System

We have been developing a range of component-based, collaborative systems, including CASE tools, meta-CASE tools and collaborative Information Systems [6, 7, 8]. Figure 1 illustrates two prototype e-commerce systems we have developed. On the left is a screen shot from an on-line purchasing system, allowing customers to search for, order and pay for books via the WWW. Staff use traditional database-oriented forms and reports to access the database, along with other desktop applications. On the right is a screen shot from a collaborative travel itinerary planner, allowing a travel agent and customer to collaboratively plan a trip [7]. This provides a variety of interfaces including structured itinerary editor, map visualisation, collaborative chat and desk top applications like Netscape™ and MS Word™. The travel itinerary planner was implemented using Java, SQL Server™ and several customised software agents, built and deployed by end users.

Figure 2 shows a possible Software Architecture that might be suitable for such e-commerce applications. A DBMS server (e.g. SQL Server™) running on a dedicated machine provides high performance, remote data management for various client programs. An HTTP server provides customers access to HTML, CGI and applets, as well as staff access to intranet documents and services. An application server provides applets server-side processing functionality embodying business rules. Both customers and staff may have various desktop applications which share files via a web-based interface. In the collaborative itinerary planner, users can also build and deploy software agents from plug-and-play components, developing custom task automation, notification and tool integration agents.



(a) On-line book purchasing. (b) Collaborative travel itinerary planner.

Figure 1. Example e-commerce systems.

A variety of different architecture models could be envisioned for this type of application, each with different performance, quality of service, security, cost and other characteristics. For example, all server-side processing and data management could be run on a single machine, reducing cost but also performance and robustness. Alternatively, multiple server-side machines could support mirrored HTTP servers, internet-dedicated vs. intranet-dedicated hosts, multiple application server hosts and so on, all increasing robustness and performance, but also increasing implementation complexity and cost. Many details not shown below need to be reasoned about, including how processes running on multiple machine communicate via suitable middleware services, what objects make up each process/program, which parts of the system are static and which dynamic, what performance etc. characteristics different processes and machines have, concurrent processes, shared data management, and so on.

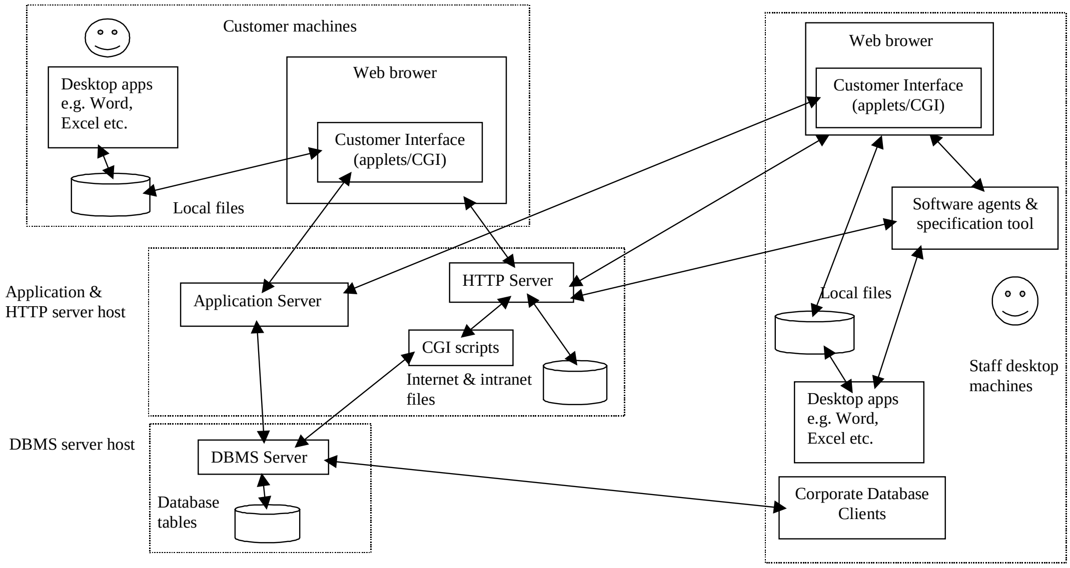


Figure 2. An example distributed Information System Software Architecture.

3. General Software Architecture Modelling Requirements

When developing systems such as the on-line book ordering system or the travel itinerary planner, software engineers need to model appropriate Software Architectures, reason about the suitability of candidate architectures, implement these architectures, and evaluate the resulting system to ensure all system specifications are met. When modelling, analysing and refining Software Architectures, developers thus have a range of requirements of the modelling approach and tools used, which we have outlined below.

- High-level Components. Software Architecture is concerned with high-level system components, rather than low-level design objects. Thus dividing a system into a manageable number of important components is fundamental in developing useful architecture models. Different kinds of system components exist, ranging from simply distinguishing client-side and server-side components, to distinguishing machines and devices from data management, processing, user interface, and device interface processes. Refinements of high-level components to intermediate architectural components and ultimately to design-level classes is usually necessary to manage the complexity of large system architectures.
- Inter-component Relationships. Approaches to relating components are a crucial part of any Software Architecture model, and many different kinds of relationship might be distinguished. Modelling might range from indicating high-level relationships, describing actual data, control and/or event interchange, to detailed connection approaches, such as sockets, shared memory, remote method invocation or remote object middleware. Grouping related components into aggregates and modelling connections between aggregates helps manage cognitive complexity of large-scale system architectures.
- Dynamic vs. Static Structure. Many Software Architectures include a need to model dynamic structural characteristics, such as run-time addition or modification of components and component relationships. This is particularly important in systems where structural evolution is inherent [15], and user-enhancable systems, such as those using plug-and-play components and software agents [8].
- High-level Behavioural Characteristics. Software Architecture models need to capture information about high-level processing performed by components, the exchange of information between components, synchronisation of access to shared data and processing, replication of data, event subscription and notification models, parallel processing and real-time response characteristics. Such behaviour characterisation would normally be at a higher level than object-oriented design models, but more detailed than analysis and non-functional requirement specifications.

- Requirements and Design links. Software Architectures are not developed in isolation. A particular architecture is motivated by functional, or more often non-functional, requirements captured about a system. Similarly, a Software Architecture model is of little use unless it is refined into a detailed system design and implementation. Thus parts of an architecture model need to be related back to the system requirements they embody and to the design-level artifacts that realise them.

In addition to a Software Architecture modelling approach satisfying the above requirements, developers need tool support to make such modelling techniques feasible for deployment. We have outlined the main requirements we have identified for tool support for Software Architecture development below:

- Modelling. Different modelling notation editors are required to allow developers to describe the range of architecture information identified above. As developers often sketch out parts of architectures roughly and then refine them over time, quite flexible modelling tools are necessary. Multiple views used to capture different perspectives on a system's architecture model, and consistency management between these views, are essential. The ability to reuse different architecture styles and patterns, and package various parts and abstractions from an architecture for reuse, are necessary to support long term architecture improvement.
- Analysis. Many different characteristics of a candidate architecture need to be analysed to ensure a model is sensible and meets requirements. Tools available to a developer might include those for analysing performance and quality of service, using ranges of performance values and QoS parameters. The correctness of a model can be checked with regard to consistent and complete structural and behavioural characteristics.
- Implementation. Support for refining an architecture model into a design and subsequent implementation is necessary, along with reverse engineering support enabling developers to deduce architecture models from existing designs. Design model and potentially code generation are possible from architecture models. Care needs to be taken to ensure a wide range of design and implementation tools are catered for, by using standardised interchange file formats where possible.
- Evaluation. Support for handling feedback from evaluation of developed systems should be included in any architecture modelling tool. Such feedback can be used not only to refine an architecture but to record performance, QoS, security, integrity and other run-time data, which may later be used to guide future model development and analysis.

4. Case Studies

In this section we review some related work into Software Architecture modelling, analysis and refinement. This is not intended to be a comprehensive literature review, but to highlight particular contributions of interest which are used in the following sections to help motivate and develop a new architecture modelling approach. Comprehensive reviews of recent Software Architecture research and practice can be found in various recent works on Software Architectures [1, 19, 20].

4.1. UML Deployment Diagrams

The Unified Modelling Language (UML) [2, 3] provides a variety of notations for modelling software systems. These include class diagrams, collaboration and sequence diagrams, and use case diagrams. For architectural modelling, UML deployment diagrams are most suitable for capturing information and deployment of system processes on machines. Component diagrams give further information about the grouping of design objects into programs. Many tools exist for supporting UML modelling, the most popular being Rational Rose ™ [18]. Figure 3 shows an example of a UML deployment diagram built using Rose for the book purchasing system introduced in Section 2. Deployment diagrams are quite simple, with two main kinds of entity ("processor" (machine) and device), with processes deployed on machines able to be listed. Machine and device inter-connection is via simple association links. Deployment diagram entities and links can be annotated with notes to document further information about them, and process concurrency characteristics can be specified. Code can not be generated from deployment diagrams, and processes in these diagrams do not directly relate to design-level classes or their methods. No consistency management between deployment diagrams, UML class diagrams and resultant code implementing a system is supported in Rose or most other UML-based CASE tools.

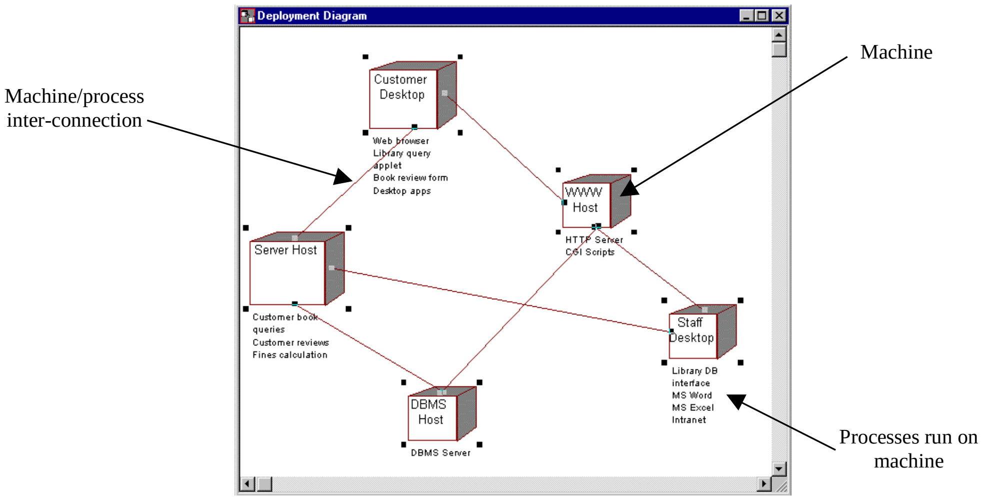


Figure 3. Example of a UML Deployment Diagram.

4.2. JComposer Component Diagrams

JComposer is a CASE tool for developing component-based systems [6]. Figure 4 shows an example of a JComposer component diagram. Such diagrams provide a variety of abstractions for modelling components, relationship components, links, and event filtering and actioning components. A variety of links can be used, including structural association and aggregation, generalisation, event flow, and "usage" (which includes method calling or more general inter-component relationship identification). Both low-level components, corresponding to design-level objects, and higher-level "aggregate components" can be modelled in JComposer. Complex inter-component relationships can themselves be modelled as components, with a variety of characteristics specified. Code can be generated to implement low-level components, and design-level component refinements and requirements-level component abstractions linked to architectural components. Thus JComposer describes the static structure of components and the dynamic behaviour of component instances for component classes that make up a system. Sophisticated consistency management between these different levels of abstraction is supported.

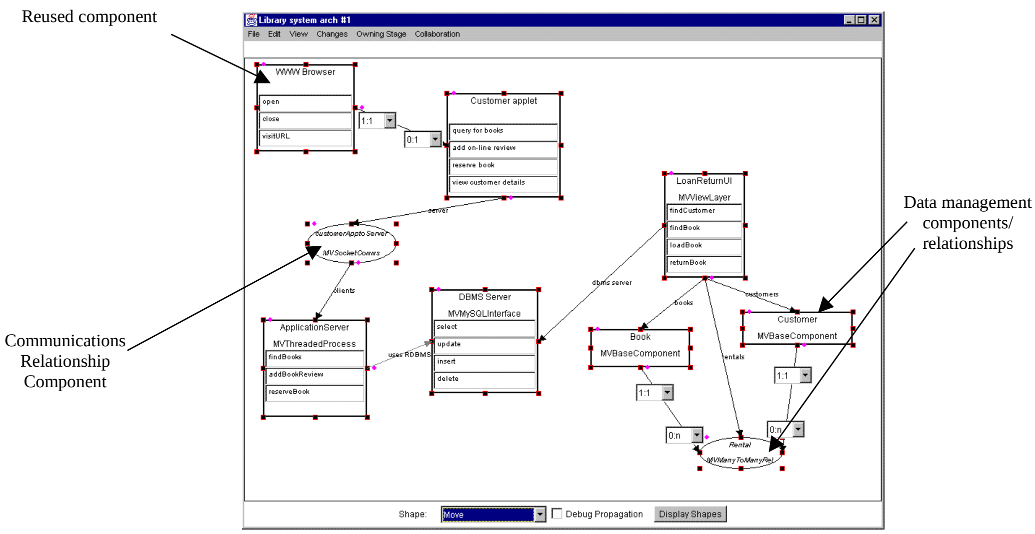


Figure 4. Example of a JComposer Component Diagram.

4.3. Serendipity-II Agent Specification Diagrams

Serendipity-II is a process management environment, built using JComposer, which provides a variety of visual process modelling views [8]. Serendipity-II also provides views for interactively speciftying software agents, similar to JComposer component views but where actual running software components are created and composed [7, 13]. Serendipity-II was used to construct the itinerary planner prototype outlined in Section 2 by composition of reusable software agents. Figure 5 shows two Serendipity-II agent specification views for part of the travel itinerary planner. This describes how travel itinerary update events are detected by an agent which updates a map visualisation, and how map visualisation events are detected by an agent which notifies other users via the collaborative chat component. Serendipity-II supports the notion of components, event filtering components, and event actioning components. Event propagation between components, and inter-component method calling, are indicated by links between component representations. As users construct Serendipity-II agent specification views, appropriate software agents are found or created, and as links are constructed or modified graphically, appropriate inter-component relationships are established or modified. Various additional component parameters are set via dialogues.

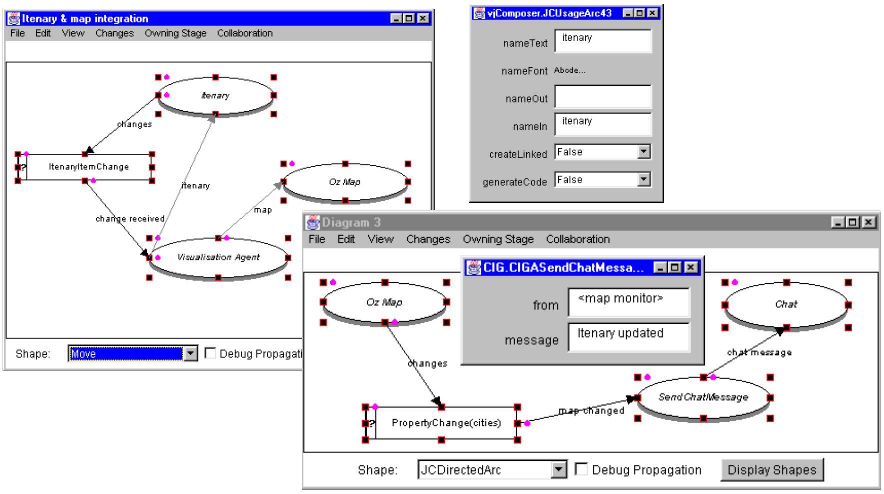


Figure 5. Serendipity-II agent specification views.

4.4. Clockworks MVC Diagrams

Clockworks provides an integrated environment for developing programs using the Clock language [5, 21]. Clockworks components correspond to design-level classes, like UML class diagram classes and some JComposer components, which are composed of layered Abstract Data Types. A Model-View-Controller approach is used to define views of the model components in a layered, tree-based manner. “Server” components thus act as the root of a Clockworks MVC tree, with “client” components defined as composites built on top of this model. Figure 6 shows an example of a Clockworks architecture diagram for the travel itinerary planner from Section 2. The Itinerary component is the model, which would be managed on a server machine, with ADTs providing access to itinerary item structures. Two views are defined for the itinerary, one for travel agents and customers to edit the itinerary in a tree-structured way, and one for map visualisations. Each view is composed of sub-components providing different parts of the itinerary views, and each runs on a client PC.

Clockworks specifies Clock component (class) structures and aspects of component behaviour. Code is generated from Clockworks specifications, including quite complex design refinements of architectural abstractions (such as code for client-side caching, shared ADT access and concurrency controls) [15]. ADTs can be annotated to represent replication, for example the itinerary items can be replicated. Client-server splits are identified on tree branches, with components above the split on the server machine and those below on the client. Various data caching options can be specified, including pre-send caching, pre-fetch, simple caching and no caching. Concurrency control annotations can be used to indicate concurrent access.

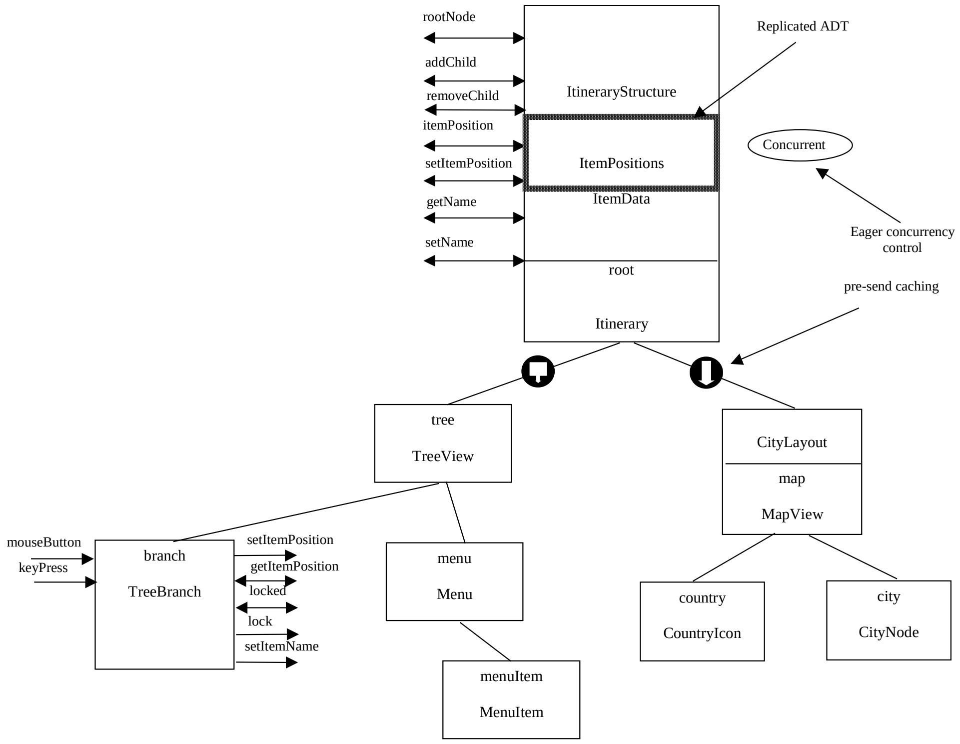


Figure 6. Example of a Clockworks Architcture Diagram.

4.5. ViTABaL Toolie Diagrams

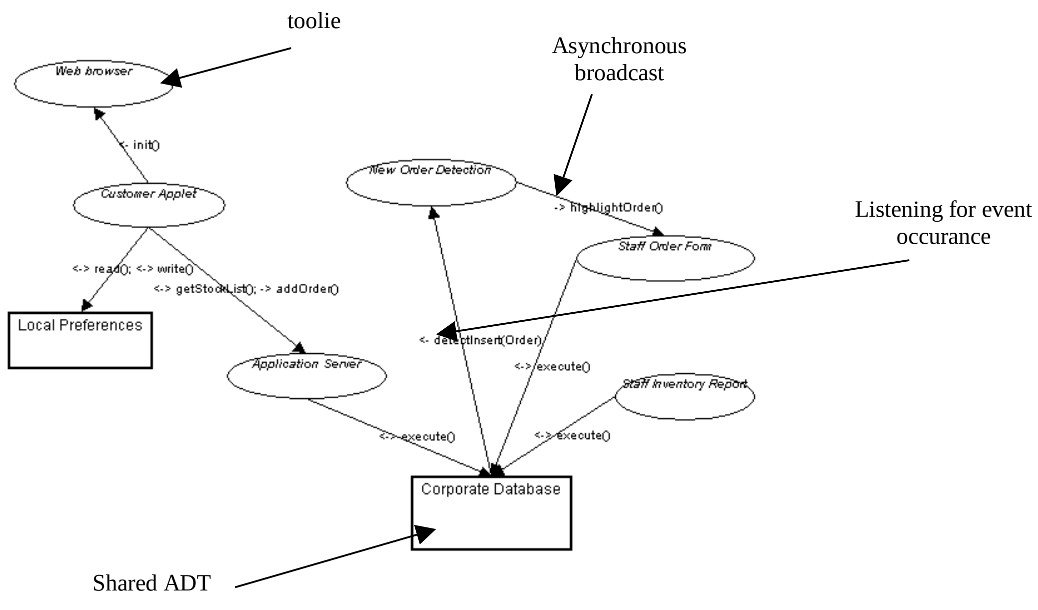


Figure 7. ViTABaL toolie and ADT diagram.

ViTABaL (Visual Tool Abstraction Language) is a notation and environment for modelling systems based on the tool abstraction paradigm of Garlan et al [4, 9]. Components are either "toolies", which embody functionality, or "abstract data types", which embody data management. Toolies typically share a pool of ADTs, and communication between toolies is either via the ADT pool or direct communication. Figure 7 shows a ViTABaL Toolie diagram describing the book purchasing program from Section 2.

ViTABaL provides a range of inter-toolie message passing abstractions, including synchronous and asynchronous broadcasting and requests, and before-action and after-action event monitoring. ViTABaL also allows toolies to be decomposed into smaller aggregate toolies, with local ADTs and inter-toolie communication. Dynamic addition and removal of toolies is possible. Unlike UML, JComposer and Clockworks, manipulating ViTABaL diagrams corresponds to manipulating live architecture component instances. A set of toolie monitoring and debugging tools are fully-integrated with ViTABaL toolie diagrams. Inter-toolie communication code is dynamically generated from toolie diagrams rather than coded manually.

5.6. PARSE-DAT Process Graph Diagrams

PARSE-DAT provides a notation, editing tools and analysis method for specifying and simulating dynamically-structured software architectures for distributed and parallel processing systems [Refs]. The PARSE notation uses a set of symbols to represent static architectural components, including function, data and control servers (processes) and a variety of path inter-connections between processes. It also provides symbols to represent dynamic parts of a software architecture i.e. nodes and connections that can be created and modified at run-time. Figure 8 shows a PARSE process graph for the collaborative travel itinerary planner from Section 2. In this example, notification agents and their monitoring/actioning connections shown as dynamic (as end-users can create and remove these dynamically, as can other software agents).

PARSE-DAT provides an environment for modelling architectures using PARSE. A PARSE process graph can be translated by PARSE-DAT into a formal specification using π-calculus. This can then be analysed to verify the correctness of the PARSE process graph specification. An integrated modelling and analysis environment is provided by PARSE-DAT, with feedback from architecture analysis into the modelling process.

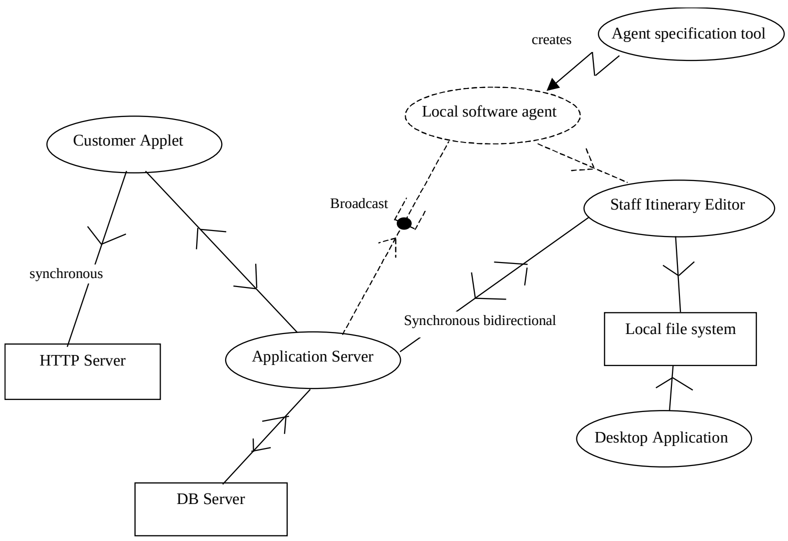


Figure 8. Example PARSE Process Graph Diagram.

5.7. JComposer Aspect Diagrams

A recent enhancement of JComposer has been the addition of component "aspects", used to categorise and describe the provided services and required services of components, including both their functional and non-functional characteristics [12]. Aspects include persistency management, distribution strategies, security and transaction processing models, user interfaces and component configuration support, and for each such aspect a component may provide or require several aspect details. Figure 9 shows a JComposer aspect diagram for the book purchasing application, where JComposer component aspects are visualised and inter-relationships specified. Provided aspect details are indicated by a "+" and required by a "-". Aspect details have properties further specifying their characteristics, which may include performance, robustness, security, integrity and other non-functional requirements. Aspect information can be modelled at the requirments-level for abstract components, at software component object design-level, can be encoded in component implementations and be accessed by users and other components at run-time. JComposer supports the modelling of component aspects, refinement of aspects, encoding generation and consistency management between aspects at different levels of abstraction. Aggregate aspects can be used to describe provided and required services of multiple, related components, including a whole component-based system.

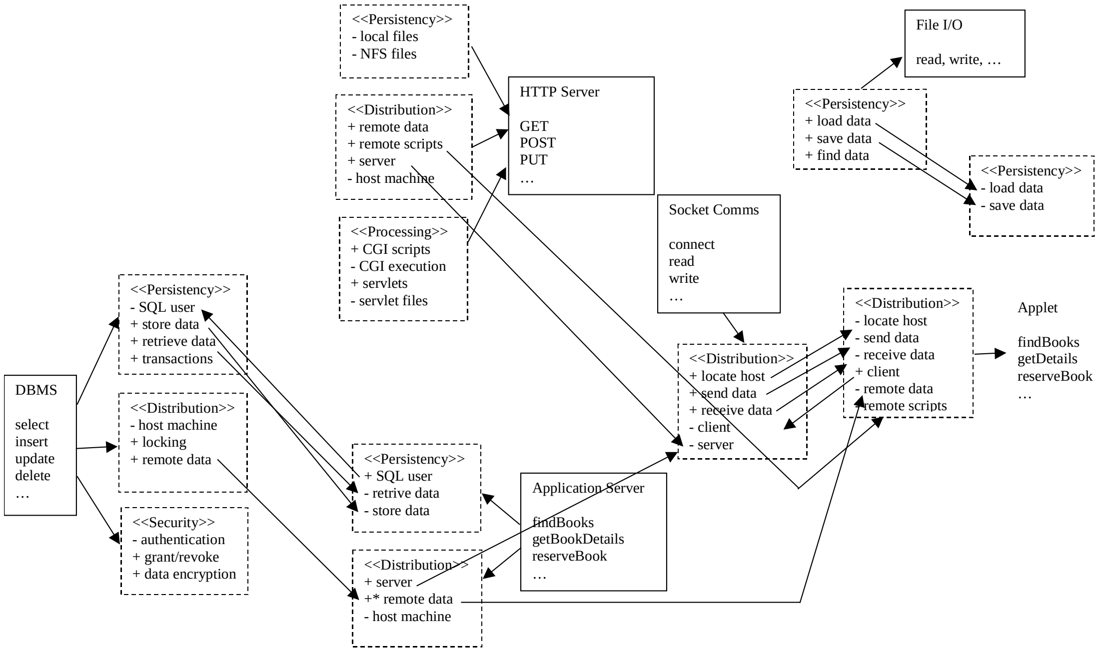


Figure 9. Example of a JComposer Aspect Diagram.

6. Critique of Modelling Approaches

Each of the architectural modelling approaches identified in the previous section offers a different way to specify characteristics of a system architecture. Table 1 summarises these different approaches with respect to the requirements for architecture modelling outlined in Section 3.

The UML provides a wide range of system modelling capabilities, with deployment diagrams providing a reasonably high-level approach to architecture modelling. There isn't a way to group machines and processes, however, and specifying information about the inter-relationships between processes is very limited. It appears only the static structure of architectures can be described and visualised, and very limited information about system behaviour can be captured. Limited links and consistency between deployment diagrams and other UML artifacts exist. Modelling support in tools like Rose is reasonable, but no analysis support exists i.e. validating deployment models, suggesting suitable models etc. No support exists for realising deployment diagram-specified information nor feeding back information from testing the architectures specified.

JComposer component diagrams are generally lower-level than deployment diagrams, though aggregate components can be used to model similar abstractions to processes. No information about machine deployment is directly supported in such views. A much richer range of inter-process structural and dynamic information can be modelled, similar to UML class and collaboration diagrams but with richer event modelling. Dynamic structural configurations are not modelled visually. Usage and event links can model "aggregate" method and event inter-change between components, and relationship components can be used to model complex architectural facilities like middleware. Links and consistency management with requirements-level and design-level entities are provided. Very few analysis tools for component configurations exist, especially about processing concurrency and dynamic configurations, but . Code generation is supported, though refinement to design-level components must be done first. Feedback from a component visualisation tool can be utilised when modifying architectures.

Serendipity-II diagrams tend to be at a similar level to JComposer diagrams, though there are some annotations indicating machines distributed agents run on and their inter-process communication middleware [13]. Dynamic system architecture composition is supported by Serendipity-II, though most modelling work tends to focus on lower-level, detailed component event and message inter-change. New composite agents can be built from reusable parts and repeatedly instantiated and parameterised, fostering reuse of higher-level abstractions. A good range of reusable components is provided for users, though their tailorability is often limited. Some limited, distributed monitoring of agents is supported.

	UML	JComposer	Serendipity-II	Clockworks	ViTABaL	PARSE-DAT	Aspects
Components	Medium-level machines and processes	Low to medium-level data and behaviour components	Low to medium-level components ("agents")	Low level model and view components	Medium to high-level toolies and ADTs	Medium to high-level servers	Low to high-level components and component aspects
Relationships	Very limited	Structural, with dynamic event exchange and method calling	Event & message propagation, association	Structural, with some dynamic events and messages	Message-passing association and semantics	Inter-process data communications	Provides and requires associations
Structure	Static	Static	Static (though run-time specified by user)	Static	Mainly static, run-time specified by user	Static and dynamic	Mainly static with some dynamic
Behaviour	Limited	Low-level: good; Limited at higher levels	Low to medium level event propagation, associations	Good for medium-level behaviour	Good for medium and high level behaviour	Good for medium- high level process communication	Non-functional: good; Functional: limited
Links	Limited links & consistency	Some links & consistency to both requirements & code	Limited	Code links & consistency good	Code links & consistency good	Formal specification can be generated	Good: consistent metaphor throughout
Modelling	Good	Very Good	Good	Very Good	Very Good	Very Good	Basic support
Analysis	Very limited	Limited	Basic configuration validation	Some for annotations	Consistency checking	Very Good	Basic configuration validation
Implementation	Code generation & reverse engineering from class diagrams	Good – integrated environment	Reuse of existing components	Good – integrated environment	Very good – well-integrated environment	Limited	Limited – some code generation & reverse engineering
Evaluation	Testing tool; no feedback	Component visualisation tool	Limited debugging support	Some debugging support	Integrated testing & monitoring tools	Limited	Run-time aspect information used

Table 1. Comparison of approaches.

Clockworks provides similar capabilities to UML class diagram and JComposer component diagrams, except for the addition of various architecture-related annotations. These provide additional information for architecture modellers in regards to specifying component and relationship characteristics, and some dynamic behavioural characteristics. No dynamic component structure visualisation support exists in Clockworks, and limited behavioural specification at the visual level. Good integrated tool support exists, with code generation and debugging tool support. Limited analysis tools check the suitablility and compatibility of architectural annotations on Clockworks models.

ViTABaL, in general, attempts to describe larger architectural entities than Clockworks, JComposer and UML class diagrams ("toolies" vs. software components or objects), although large toolies can be built from smaller ones which are more like conventional design-level objects. ViTABaL supports both low-level behavioural specification between toolies and ADTs and higher-level, abstract associations. Some limited support for dynamic system structure is supported in the visual notation, and annotations indicating which machine a toolie is to be run on are used. Consistency management between toolie views and toolie method implementation code is supported, including automatic code generation for complex inter-toolie method and event propagation via middleware abstractions. Modelling of toolie views is done "live" in ViTABaL, with users manipulating actual running architectural entities. Some limited analysis support is provided including structural and basic behavioural validation, and some concurrency annotation and validation support. Monitoring and debugging tools are tightly integrated with the modelling tools, and monitor information can be used to improve architecture structure.

PARSE-DAT provides a mainly process-oriented view of architectures. Inter-process communication strategies are well-represented, along with basic data management, process management and control process discrimination. Good support is provided for dynamically structured systems, unlike many other approaches. Limited detail about structural and behavioural relationships is captured. PARSE-DAT supports editing and analysis of PARSE process graphs, with a greater degree of formal specification and verification than most other techniques. Limited support for design-level refinement and architecture implementation and evaluation is provided.

Component aspects provide an additional perspective on architectural entities unlike those of the preceding systems. Aspects offer various perspectives on architecture component provided and required services, with the ability to reason about related components via their aspects. Some support for modelling, analysing and refining aspects is provided by JComposer. Run-time use of aspects is supported by the JViews software architecture and its implementation framework.

7. A Synthesised Architecture Modelling Approach

Issues	Facilities		
	OOD Capabilities	Low-level Software Architecture	High-level Software Architecture
Components	UML/JComposer/Clockworks software components; Serendipity-II dynamic, running components.	UML machines; ViTABaL toolies; aspects; PARSE servers	ViTABaL toolie groupings; aggregate aspects; machine groupings; Process groupings
Relationships	UML/JComposer/Clockworks generalisation, association. JComposer event propagation. UML method calling. UML state transition.	JComposer usage links; ViTABaL toolie and PARSE server message exchange; aspect provides and requires relationships; aggregate message/event links; link annotations. Middleware characteristics.	High-level structure, data/control exchange, usage links; aggregate provides and requires associations; link annotations. High-level middleware characteristics.
Structure	Static and dynamic component and relationship characterisation.	Machine, process and process group, toolie, and server data management and various interconnection specification. Dynamic parts and relationships. Middleware support ORBs.	Machine and toolie groupings; dynamic groupings and relationships.
Behaviour	Method and event exchanges. State transitions.	Tool/process/server-level dynamic behaviour. Grouping of methods, events, state transitions. Concurrency and parallel processing characteristics.	Overview of data and control movement. Overview of concurrency/parallel processing. Overview of communication strategies.
Links	OOA specifications, detailed design classes and code modules.	Individual OOA specification objects, program and process entities, OOD classes, non-functional requirements.	Low-level architectural entities, non-functional requirements. Possibly grouped OOA objects.

Table 2. Some desired Architecture Modelling capabilities.

Overall the notations and tools described in Section 4 provide a good range of architecture modelling approaches. Many support particular aspects of architecture modelling well, but either do not address others or address them in unsatisfactory ways. A synthesis of these modelling techniques can be envisaged which combines their various features to provide a more complete modelling notation for large software architectures. Table 2 summarises the various capabilities that might be chosen to support such modelling. Note that an architecture modelling approach would normally be from a static view of a system i.e. JComposer, UML, PARSE and Clockworks-style specification of classes/processes, rather than a ViTABaL and Serendipity-II view and manipulation of actual running objects. However, support for both is needed in general – architects may work with static specifications whicha refined into OO designs and implementations, but the ability to view and modify running architectures via suitable abstractions is also necessary.

Figure 10 illustrates the kind of modelling capabilities that might be used based on the combination of features identified above. From our work in developing architectures for several large software tool environments and collaborative systems, we have determined that it is important to provide a range of abstractions for modelling architectures. These range from high-level views capturing information about the overall architecture and important composite components, to low-level views specifying all architectural refinements and information about concurrency controls, data access and replication, data and operation specification and so on. As the various notations overviewed in this paper have some overlap in terms of their modelling capabilities, a new, consistent set of notational symbols would be required so that a consistent set of abstractions is available to modellers.

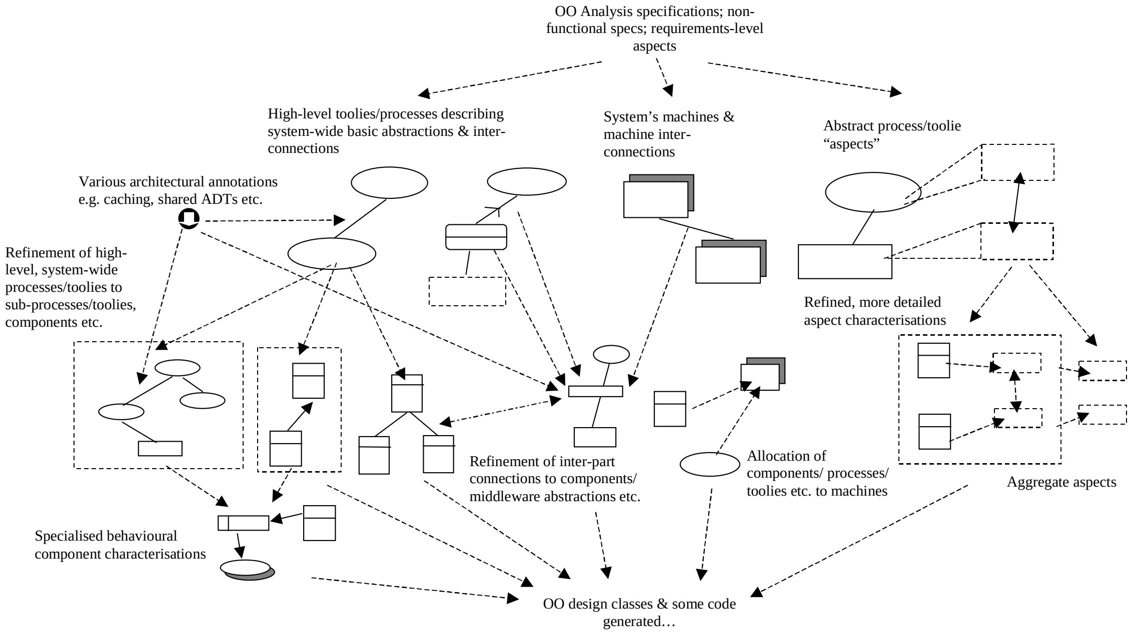


Figure 10. Multi-level architecture modelling using a synthesis of approaches.

8. Tool Support

Good tool support is necessary to make such a synthesised modelling approach feasible. Such a modelling tool needs to support various views of a software architecture, ranging from high-level, basic structural and behavioural abstractions to detailed component connectivity, data management and behavioural specifications. Guideance support, including reusable architectural styles and patterns, Analysis tools like those of PARSE-DAT, ViTABaL and JComposer aspects are required to enable developers to formally reason about their architectures. OO designs and implementation code generation should be supported by the tool, in conjunction with existing CASE and development environments. Results of using architectures, such as actual performance characteristics, robustness, security and maintainability, should be captured and associated with various architectural abstractions. Reverse engineering of existing architecture models needs to be supported by the tool, to enable developers to reuse and evolve their existing system architecture models and infrastructure.

Multiple Views

Multiple views of a large system's architecture should provide representations of parts of the architectural characteristics at varying levels of abstraction and focusing on different parts and aspects of the architecture. For example, high-level views of the overall system might use ViTABaL and PARSE-DAT style toolies/processes, UML machines and JComposer aspects to characterise overall system characteristics. High-level annotations of components and links might indicate abstract characteristics of each. Refinements of such views might focus on structural inter-relationships (sub-toolies, sub-processes, components, component trees etc.), behavioural inter-relationships (message passing details, middleware support structures, event subscription and notification details etc.), depictions of important concurrency and parallelism, assignment of processes etc. to machines, refinements of aspects, aggregate aspects and aspect inter-relationships, and model-view-controller style structures. These “mid-level” architectural views would be refined to OOD-level classes and perhaps some implementation-level code, detailing the realisation of architectural decisions.

All editing tools should support appropriate notational representations and highlight links between different views and refinements of the architecture. Some details about specifications at various levels would require dialogue input, though use of appropriate annotations would be desirable on the visual representations. Formalised encodings of parts of the architecture, such as aspect properties, constraints on data and operation usage (pre/post-conditions, invariants etc.), sequencing of operations and so on would likely require textual encodings. Consistency management between views is essential, as many parts of the synthesised notation contain overlaps with other parts. Flexible consistency management is more likely to be useful, with a lot of consistency checking on-demand rather than always enforced i.e. more analysis-style checking than editing-level checking. Appropriate basic syntactic and semantic editing checks should be enforced where appropriate though. Supporting “sketching” of incomplete parts of architectures without undue validity checking would likely enhance modellers ability to experiment and informally document architectural ideas before rigid checking is applied.

7.2. Analysis Tools

Two main kinds of analysis checks could be envisaged: basic consistency checking applied to the model at differing levels of abstraction, and formal reasoning about various architectural properties (e.g. concurrency deadlock potential, potential points of failure, expected performance given required data management and transaction processing parameters, potential security anomalies, etc.). Model consistency checking must ensure information specified about the architectural model from various perspectives (views) defines a complete architecture specification, is internally consistent and all required services of architecture components are satisfied. Model analysis should assist modellers in determining whether their models meet the non-functional and functional specifications for the system, and that the architecture has appropriate expected performance properties (processes won’t deadlock, data won’t be corrupted, security won’t be breached and so on). The use of formalisms like state machine models, as employed in ViTABaL, and π-calculus, as employed in PARSE-DAT, should be deployed where possible, to ensure such architectural analysis is well-grounded. In addition, wizard-like guideance for modellers, using various metrics and recommended practices, could give more informal feedback on the likely suitability of an architecture model.

7.3. Design and Code Generation

Developers will want to generate design-level classes and perhaps implementation-level code based on their architecture models. For large systems, such design and code generation alleviates a great deal of tedium and error-prone hand-coding, but also ensures consistency between model and design/implementation. A key to ensuring the success of design/code generation tools is to allow various third-party CASE tools and development environments to use generated information. This suggests the use of standardised representations of design information, such as XMI-based UML encodings, and target language code. IDLs, as provided by middleware systems like CORBA, also offer an encoding which is more portable than custom tool repository formats.

7.4. Reverse Engineering Support

In addition to generating designs and some code, an architecture modelling tool needs to support reverse engineering. This is for two reasons: to support round-trip engineering where design and code changes can be reverse-engineered back into the architecture model, and to support the modelling and evolution of existing systems. A complexity with the second case is identifying appropriate architectural abstractions from OO design-level classes. Some common architectural patterns or styles might be recognised by a tool, but modeller intervention is probably necessary in general to capture all architectural features, particularly higher-level abstractions.

7.5. Architecture Performance Measurements

When modelling architectures a number of metrics and formal analysis techniques could be deployed to assist developers as outlined above. In general, additional information about the actual performance characteristics (speed, robustness, integrity, security, quality etc.) would be very useful to base analysis reasoning on. For example, if a modeller wanted to compare socket-based inter-process communication speed and reliability to CORBA-supported remote operation invocation, various actual system performance measurements relating to the use of these different architectural abstractions and implementing component implementation would be useful. A modelling environment that captured such characteristics by monitoring running systems could offer improved guidance and analysis support to developers, and the results used by developers to validate a system meets non-functional requirements. Storage of such results against reusable architectural entities would facilitate more accurate and appropriate reuse in future architecture modelling situations.

9. Conclusions

The different architecture modelling approaches overviewed in this paper offer various facilities to developers, most of them complementary and from differing perspectives. A synthesis of these, and possibly other, approaches would seem to offer a more holistic method for developing software architecture specifications for large systems. Important features of such an approach include support for multi-level architecture component and relationship modelling, static and dynamic structure modelling, a range of behavioural modelling facilities, and appropriate linkage to requirements and design artifacts. Development tool support should include multiple views and consistency management, various formal and semi-formal analysis techniques, design and code generation as well as reverse engineering support, and the capture of actual performance data for use when analysing and evaluating architectures.

We are currenty developing a unified notation for modelling software architectures based on a synthesis of the approaches surveyed in this paper. An important characteristic of this approach is the use of multiple levels of architectural model refinements and a common notation at each level to express architectural elements. A prototype CASE tool supporting this approach is planned for development, by extending the JComposer CASE environment to add some of the architecture modelling facilities outlined in the previous sections. This has the dual advantage of using JComposer's metaCASE facilities and allowing linkage of requirements-level artifacts and design-level artifacts in JComposer to the architecture-level artifacts. The addition of more formal Architecture Description Language properites to this modelling approach may enhance its overall analysis capabilities. Interchange of ADL and architecture modelling notation models may enhance the ability of developers to work with appropriate notations and approaches in different domains and for different perspectives of systems.

References

1. Bass, L., Clements, P. and Kazman, R. Software Architecture in Practice, Addison-Wesley, 1998.
2. Booch, G. Jacobsob, I. And Rumbaugh, The Unified Modeling Language User Guide, Addison-Wesley, 1999.
3. Fowler, M. UML Distilled, Second Edition: A Brief Guide to the Standard Object Modeling Language, Addison-Wesley, 1998.
4. Garlan, D., Kaiser, G.E., and Notkin, D., " Using Tool Abstraction to Compose Systems," *COMPUTER*, vol. 25, no. 6, 30-38, June 1992.
5. Graham, T.C.N., Morton, C.A. and Urnes, T. ClockWorks: Visual Programming of Component-Based Software Architectures. *Journal of Visual Languages and Computing*, Academic Press, pp. 175-196, July 1996.
6. Grundy, J.C., Mugridge, W.B., Hosking, J.G. Static and dynamic visualisation of component-based software architectures, In *Proceedings of 10th International Conference on Software Engineering and Knowledge Engineering*, San Francisco, June 18-20, 1998, KSI Press.
7. Grundy, J.C., Mugridge, W.B., Hosking, J.G., and Apperley, M.D., Tool integration, collaborative work and user interaction issues in component-based software architectures, In *Proceedings of TOOLS Pacific '98*, Melbourne, Australia, 24-26 November, IEEE CS Press.
8. Grundy, J.C., Hosking, J.G., Mugridge, W.B. and Apperley, M.D. An architecture for decentralised process modelling and enactment, *IEEE Internet Computing*, Vol. 2, No. 5, September/October 1998, IEEE CS Press.
9. Grundy, J.C., Hosking, J.G. ViTABaL: A Visual Language Supporting Design by Tool Abstraction, In *Proceedings of the 1995 IEEE Symposium on Visual Languages*, Darmsdart, Germany, September 1995, IEEE CS Press, pp. 53-60.
10. Grundy, J.C. Human Interaction Issues for User-configurable Collaborative Editing Systems, In *Proceedings of APCHI'98*, Tokyo, Japan, July 15-17 1998, IEEE CS Press, pp. 145-150.
11. Grundy, J.C. Engineering component-based, user-configurable collaborative editing systems, In *Proceedings of the 7th Conference on Engineering for Human-Computer Interaction*, Crete, Greece, Sept 14-18 1998, Kluwer Academic Publishers.

12. Grundy, J.C. Aspect-oriented Requirements Engineering for Component-based Software Systems, In *1999 IEEE Symposium on Requirements Engineering*, Limmerick, Ireland, 7-11 June, 1999, IEEE CS Press.
13. Grundy, J.C. Visual specification and monitoring of software agents in decentralized process-centred environments, *International Journal of Software Engineering and Knowledge Engineering*, Vol. 9, No. 4, September, 1999.
14. Kazman, R., and Carriere, S.J. Playing Detective: Reconstructing Software Architecture from Available Evidence, *Automated Software Engineering*, Vol. 6, No. 2, April, 1999, 107-138.
15. Liu, A. Dynamic Distributed Software Architecture Design with PARSE-DAT, In *Proceedings of the 1998 Australasian Workshop on Software Architectures*, Melbourne, Australia, Nov 24, Monash University Press.
16. Mowbray, T.J., Ruh, W.A. *Inside Corba: Distributed Object Standards and Applications*, Addison-Wesley, 1997.
17. O'Neil, J. and Schildt, H. *Java Beans Programming from the Ground Up*, Osborne McGraw-Hill, 1998.
18. Quatrani, T. Visual Modeling With Rational Rose and Uml, Addison-Wesley, 1998.
19. Shaw, M. and Garlan, D. Software Architecture : Perspectives on an Emerging Discipline, Prentice Hall, 1996.
20. Szyperski, C.A. *Component Software: Beyond Object-oriented Programming*, Addison-Wesley, 1997.
21. Urnes, T. and Graham, T.C.N. Flexibly Mapping Synchronous Groupware Architectures to Distributed Implementations. In *Proceedings of Design, Specification and Verification of Interactive Systems*, 1999.
22. Witt, B.I., Baker, F. T. and Merritt, E. W., *Software architecture and design: Principles, models, and methods*, Van Nostrand, 1994.