Swinburne Research Bank
http://researchbank.swinburne.edu.au

Grundy, J., & Hosking, J. (2003). SoftArch: tool support for integrated software architecture development.

Electronic version of an article published as
International Journal of Software Engineering and Knowledge Engineering, 13(2), 125–151.

Available from: http://dx.doi.org/10.1142/S0218194003001238

Copyright © 2003 World Scientific Publishing Company.

This is the author's version of the work, posted here with the permission of the publisher for your personal use. No further distribution is permitted. You may also be able to access the published version from your library. The definitive version is available at http://www.worldscinet.com/.

A preprint of article published in International Journal of Software Engineering and Knowledge Engineering vol. 13, no. 3, © World Scientific 2003.

SOFTARCH: TOOL SUPPORT FOR INTEGRATED SOFTWARE ARCHITECTURE DEVELOPMENT

JOHN GRUNDY[1, 2] AND JOHN HOSKING[1]

Department of Computer Science[1] and Department of Electrical and Electronic Engineering[2], University of Auckland
Private Bag 92019, Auckland, New Zealand
john-g@cs.auckland.ac.nz

Abstract

A good software architecture design is crucial in successfully realising an object-oriented analysis (OOA) specification with an object-oriented design (OOD) model that meets the specification's functional and non-functional requirements. Most CASE tools and software architecture design notations do not adequately support software architecture modelling and analysis, integration with OOA and OOD methods and tools, and high-level, dynamic architectural visualisations of running systems. We describe SoftArch, an environment that provides flexible software architecture modelling using a concept of successive refinement and an extensible architecture meta-model. SoftArch provides extensible analysis tools enabling developers to analyse their architecture model properties. Run-time visualisation of systems uses dynamic annotation and animation of high-level architectural modelling views. SoftArch is integrated with a component-based CASE tool and run-time monitoring tool, and has facilities for 3rd party tool integration through a common exchange format. This paper discusses the motivation for SoftArch, its modelling, analysis and dynamic visualisation capabilities, and its integration with various analysis, design and implementation tools.

Keywords: software architecture, software tools, architecture modelling and analysis, software visualisation

1 Introduction

Many software modelling notations and tools have been developed over time [1, 2, 3, 4]. Due to the increasing complexity of software systems there has been an increasing emphasis on software architecture modelling in CASE tools in addition to the more conventional object-oriented analysis (OOA) and object-oriented design (OOD) modelling [5, 6, 3, 4]. Various design notations have been developed, including those of UML [7], PARSE [4], JViews and aspects [8, 9], tool abstraction [2], and Clock [1, 10]. Support tools include Rational Rose [11], JComposer [9], PARSE-DAT [4], ViTABaL [2], SAAMTool [3] and Argo/UML [12]. The Unified Modelling Language (UML) [7] uses a combination of class, collaboration, component and deployment diagrams. Clockworks and JComposer use annotated component diagrams [1, 9]. PARSE-DAT and ViTABaL use process diagrams. Several systems, including SAAMTool [3], Argo/UML [12] and Visper [13], use various kinds of structural architecture component diagrams.

Most of these systems provide only partial software architecture modelling solutions, supporting some aspects of architecture modelling supported e.g. basic structure, limited dynamic behaviour and event models, or dynamic process creation [15, 16]. Most only capture limited knowledge about an architecture's properties and the characteristics of architecture elements. Few provide analysis tools to help developers reason about their models and ensure OOA requirements are met and architecture components refined to suitable OOD abstractions [12, 15]. Few support OOD and/or implementation code generation from architecture-level abstractions, and few support reuse of previously developed models and patterns [10, 12]. Almost none are sufficiently extensible to allow new architecture abstractions and analysis tools to be added, and most architecture representations in tools have poor or no integration with related analysis, design and implementation abstractions. High-level dynamic visualisation of algorithms and design-level call graphs and dataflow have been used for many years [17, 18, 19, 14, 20] to provide a mixture of views of running program information. Most of these approaches focus on object or algorithm-level dynamic visualisation techniques, rather than architectural component visualisation. Limited architecture-level visualisations have been developed, together with approaches to visualise running systems [18, 19]. However most dynamic visualisations bear little or no relation to static architecture visualisation (design) notations, making them hard to understand and interpret.

We describe SoftArch, an environment providing new approaches for software architecture modelling, analysis, visualisation and tool integration. Architects use an extensible visual notation to describe and refine software architecture models. Detailed properties of architecture elements and element groupings capture knowledge of architectural characteristics. A collection of extensible "analysis agents" constrain, guide and advise architects as they build and refine these models. Visualisation of running system architectures using high-level abstractions in SoftArch is supported. SoftArch has been integrated with process management, analysis, design and implementation tools, using a variety of tool integration techniques, to "value-add" to a software designer's overall tool set by providing support for complementary, integrated architecture development.

In the following sections we motivate the need for SoftArch and review current support for architecture modelling, analysis and dynamic visualisation support. We then overview the facilities of SoftArch, focusing in turn on its static architecture modelling, architecture analysis, and dynamic architecture-level visualisation support. We briefly discuss the design and implementation of SoftArch, focusing on its integration with other tools (CASE, programming environments and run-time systems). We conclude with a summary of SoftArch's contributions and directions for future research.

2 Motivation

Software architecture development has become an increasingly important part of the software lifecycle, due to the increasing complexity of software being constructed [21, 7, 22]. Software developers need to carefully describe and reason about the architectures of complex, distributed information systems, which are often comprised of a mix of new and reused components. A good, extensible and maintainable architecture often makes the difference between successful and failed projects. Much more time

tends to be spent on architecture development than previously, and many more options exist for developers [21].

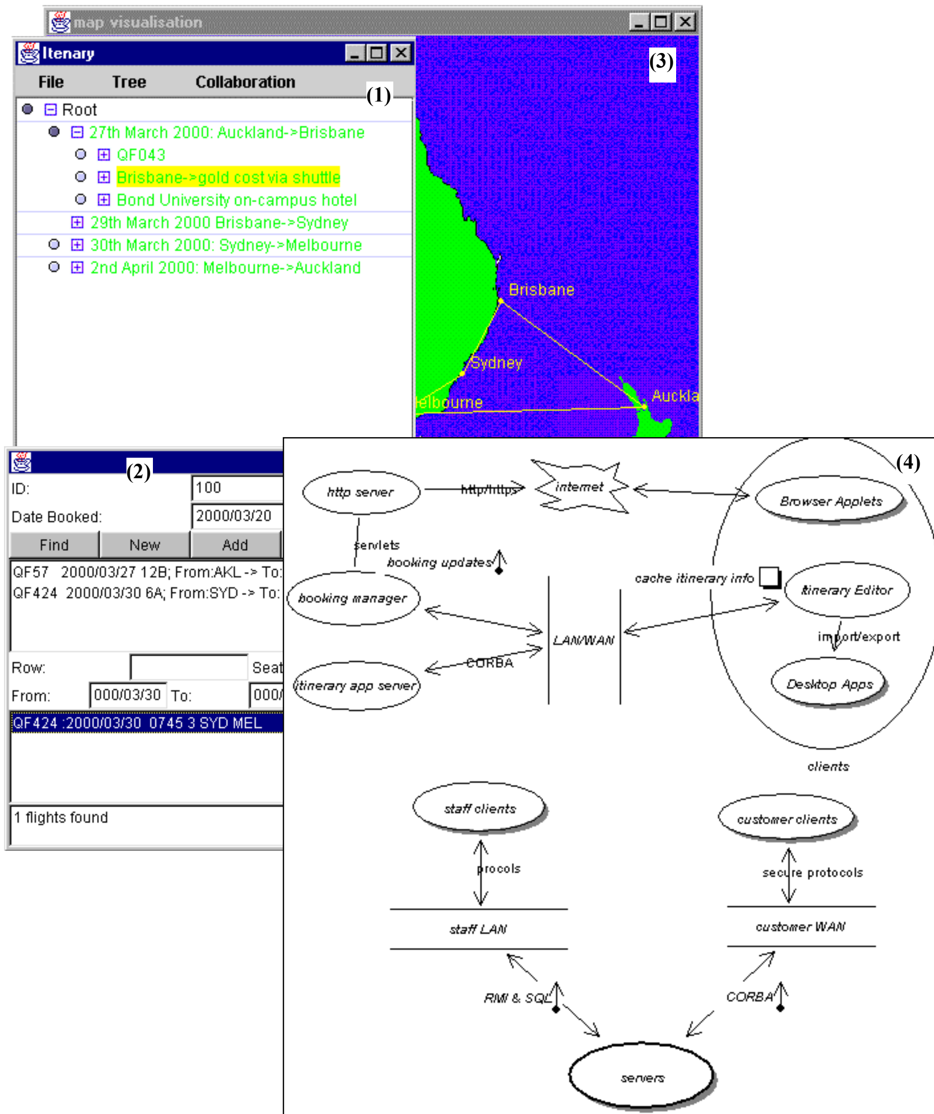


Fig. 1. (a) An example system and (b) two views of parts of its software architecture from the SoftArch environment.

Consider a simple E-commerce system, a screen dump from which is shown in Fig. 1 (a). This is a collaborative travel planning system which provides itinerary views (1), flight bookings (2) and a travel map visualisation (3) [23]. Fig. 1 (b) shows two high-level views of parts of the software architecture for this system from our SoftArch

design tool. The top view shows how specific components of the client-side and server-side processes are inter-related. The bottom view shows how customer and travel agent clients access the centralised server processes. These views are described using our SoftArch visual architecture description notation [5, 24, 22]. In order to design and build such a system, developers need to carefully model the software architecture and refine it to a suitable OOD model, ensuring it meets all system functional and non-functional specifications.

We define a software architecture as the organisation of the software elements of a system, together with relationships to the hardware and networking required to run and support communication between these software elements. Like most researchers we characterise software architectures as comprised of various components (groups of functional abstractions) and connectors linking components [5, 22]. Each has both functional (data and behavioural) and non-functional (e.g. performance, reliability, security, integrity, etc) properties. Most OOD techniques, like the UML, focus solely on detailed functional system definition. However, many software architecture description languages (ADLs) aim to associate both functional and non-functional properties of a specification with architectural elements, so these can be reasoned about [21, 5, 24, 16]. An architecture description should help developers to meet a system specification's functional and non-functional requirements, and a rich variety of architectural views may be useful (data allocation, processes and process inter-connections, subscribe-notify and event-passing approaches, host machines, processes and networking, and so on). Thus when designing the architecture for a system like the Travel Planner outlined above, developers typically require support to:

- represent processes (clients, servers, databases etc), machines (client and server hosts etc), data and other architectural components (database tables, files, etc) [1, 11]
- represent inter-component relationships, such as structural relationships, data usage, message passing, event subscription/notification, message order, concurrency and so on [16, 4, 12]
- represent additional architectural characteristics related to those above, such as data and control functions, data replication, caching, concurrency control, security mechanisms, communication protocols, etc [1, 2]
- model and reason about both static architectural connections and dynamic behaviour of related architecture components
- capture both functional and non-functional characteristics of each of these architectural features [7, 1, 16]

In addition, a system's software architecture can be viewed from many levels of abstraction, from high level (e.g. client-server; staff clients vs customer clients; server processes; multi-tier architecture) to architecture-implementing object-oriented design (OOD) classes and inter-class relationships (e.g. "TravelItineraryClient", "FlightManager", and "CustomerTable"). In any non-trivial architecture there normally exists many refinement steps from OOA specifications and high-level views of the system's architecture to detailed OOD-level class and object abstractions. Refinements of system functional and non-functional properties, using multiple levels of software architecture abstractions, thus preserves traceability from OOA specifications to low-level OOD design implementation approaches.

Fig. 2 illustrates the development process and relationships between OOA, software architecture, and OOD and implementation-level software artefacts we aim to support with SoftArch. Architects construct architecture designs at varying levels of detail to realise an OOA specification, eventually producing parts of an OO design (to be completed and implemented using e.g. CASE tools and programming environments). In addition, often existing designs and code must be reverse engineered into higher-level architectural models, which themselves may need to be reverse engineered into OOA specifications. Ideally an architecture design tool should support traceability from high-level to low-level architectural abstractions. It should also aid developers in validating the correctness of their use of architectural abstractions. Architectural design views at different levels of abstraction should be able to help developers analyse how an implemented, running system using the architecture behaves.

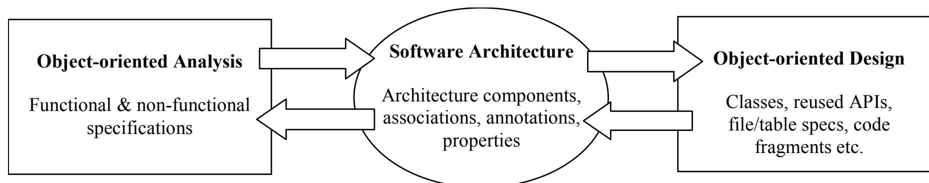


Fig. 2. Transformation of OOA model to OOD model via Software Architecture.

A tool to support architecture modelling, refinement, validation and to utilise static architecture design information to aid running architecture performance analysis should:

- Allow architects to use an extensible set of architecture modelling abstractions i.e. different kinds of components, connectors and component/connector annotations. Architects need to use a wide range of suitable abstractions when designing architectures, and need to extend these for different problem domains. Each of these architecture element types will have a variety of characteristics the designer may specify (e.g. name, location, performance characteristics, required security support, and so on).
- Support modelling the system at differing levels of architectural abstraction, from very high-level to parts of a detailed OO design. Ideally a number of visual abstractions will be provided along with detailed architectural data entry.
- Provide architects with assistance reasoning about and validating complex architecture designs. This should include checking the characteristics of related architecture elements to ensure usage constraints and non-functional properties are consistent and compatible.
- Support visualisation of implemented architecture designs using high-level design abstractions. This allows architects to link implemented system performance results with the architectural abstractions the implementation is based on
- Be able to exchange data with related tools e.g. CASE tools, programming environments, monitoring tools.

3 Related Work

Existing software architecture notations and support tools generally lack comprehensive support for architecture modelling, refinement, analysis and OOA/D linkage [15, 16]. Commonly used software modelling notations like the UML [7] provide views of classes, components and machines. Such notations suit low-level architectural representation reasonably well, but do not provide for higher level architectural oversight [16, 1]. Deployment diagrams in UML offer a view of machine and process assignment and inter-connection, but this is the only high-level specifically architectural view in UML, and is quite limited. Commonly used CASE tools, such as Rational Rose [11] and Argo/UML [12], also lack notational abstractions for designing large system architectures [15]. In addition, most CASE tools lack adequate support for refinement of OOA to OOD and architecture models and for maintaining traceability between multiple levels of system abstractions. Few provide adequate template or reusable model support and few capture architecture-related design rationale [16].

Most component engineering tools, such as JComposer [9], Borland JBuilder and that of Wagner et al [25], provide little in the way or architecture modelling support, focussing primarily on design- and implementation-level detail. The latter is necessary when developing systems, but too low-level for large system architecture development. Few support capture of multiple perspectives on architecture models and different levels of abstraction and refinement relationships. JComposer [9] and MET+[25] provide component views with some higher level associations and properties like event exchange visualised. Argo/UML [12] does provide a small amount of additional architecture-oriented notation, notably C^2-style communication "buses", but this is inadequate for large system design.

Some tools and notations have been developed specifically for software architecture modelling or have had more conmprehensive architecture modelling capabilities added. Examples include PARSE-DAT [4], ViTABaL [2], Clockworks [1], SAAMTool [3], and JComposer architectural aspects [8]. These typically support only limited kinds of architectural abstractions. PARSE-DAT focuses on process-oriented views of architectures, ViTABaL on tool-based abstraction and SAAMTool on structural composition. ClockWorks [1] uses component diagrams but with additional architecture "annotations", representing caching, concurrency and replication. Clockworks supports some code generation from these annotations to help automate realisation of such facilities from their visual specifications. PARSE-DAT provides reasonably high level views of processes and inter-process communication [4] but lacks support for OOD or for code generation, and is limited to basic process views. Most other architecture modelling approaches also focus on basic process and/or program structure (such as SAAMTool) [3]. Most tools that provide architecture notations lack support for dynamic visualisation of realised systems using equivalent notational representations.

Few CASE or other tools provide architecture model analysis and verification mechanisms or integration and reverse engineering support. PARSE-DAT, ViTABaL and ClockWorks provide some analysis support, but limited to specific kinds of domains. Argo/UML provides design critics but these mostly focus on low-level OOD model evaluation heuristics. Argo's critics cannot currently be extended in any way by users, which is problematic if new architectural modelling features need to be added to the environment. Specialised analysis tools, such as those for CSP [26], allow the

validation of (limited) architectural models via formal analysis. Some Architecture Description Language support tools, such as those for Wright [21] and Rapide [27], also focus on formal specification of architectural styles and support reasoning about the characteristics of such styles. However our key interest is not so much in the characteristics of certain architectural styles or approaches, but in supporting developers in modelling and validating the use of such styles/approaches on development projects.

Dynamic visualisation of systems is useful for developers to understand system correctness (i.e. to debug them), and to understand higher-level system behavioural characteristics that can not be easily determined from static architecture design views and analyses. Various tools support object visualisation and object structure querying, but lack higher level abstractions [18, 3]. Others support higher-level visualisation over object graphs, generating call graphs, map visualisations and 3D visualisations [28, 19], but these focus at only the object level, and are hard to scale and interpret for large, distributed applications. Various program visualisation systems have been developed, many offering high-level animations and visualisations of algorithms and object structures. These include VisualLinda [29], Rose/Architect [6], The Software Bookshelf [30], and PvaniM [31], and those using 3D call graphs and object trees [19, 18]. While these visualisations are useful, they typically bear no relation to static architecture modelling languages and views, and are thus difficult to formulate and interpret. ViTABaL [2] provides dynamic views of reasonably high-level system components ("toolies") and their relationships but developers must construct these views only from running components, limiting its usefulness.

4 Overview of SoftArch

The above deficiencies in current CASE and related approaches to software architecture design motivated us to develop the SoftArch environment. SoftArch provides an extensible visual notation for software architecture modelling support and an environment that allows models to be constructed and refined. Analysis agents guide, advise and/or constrain architects, and templates allow reuse of a variety of software architecture refinements. A visualisation facility reuses architecture modelling views to provide high-level visualisation of the dynamics of running systems. Fig. 3 outlines these basic SoftArch capabilities.

Architects build up software architecture designs drawing on a set of extensible meta-model architecture element types (components, connectors and annotations) (1). These element types describe possible kinds of architectural components, connectors and annotations, and the properties of these elements constrain the use of such entities. For example, for E-commerce systems like the travel planner, thin-clients, web servers, http requests, databases and their inter-connections are common modelling elements a designer draws upon to model important parts of their particular problem domain.

SoftArch supports the notion of refinement of software architecture elements and groups of elements into successively more detailed, numerous and lower-level element groupings (2). Properties of high-level architectural components constrain the kinds of refinements and properties at lower levels of detail. For example, a high-level travel system component such as “Customer Clients” might be refined to “Map Visualiser”, “Itinerary Editor”, and “Desktop Applications”. A conceptual group of system functionality such as “Itinerary Management” might be refined to “Itinerary Clients”,

“Itinerary Servers”, “Database Server”, “Web server” with associated connectors and annotations.

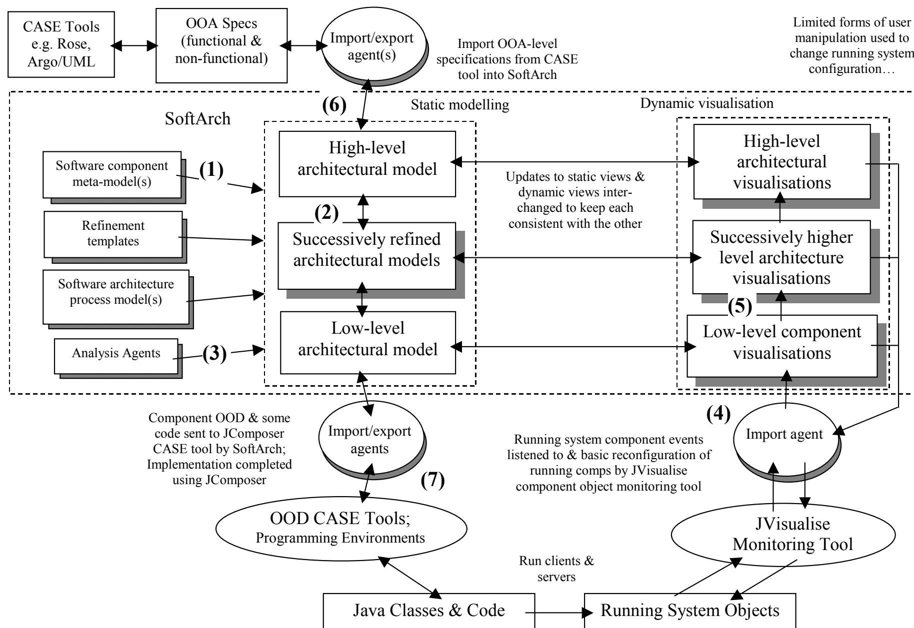


Fig. 3. Overview of SoftArch architecture design modelling, analysis and dynamic visualisation approach.

Analysis agents monitor architecture model changes and advise architects on model correctness i.e. if various meta-model specified constraints between element types and property values are being adhered to (3). These give the architect feedback (in various ways) as they model and refine a system. This feedback is typically unobtrusive, though architects can request immediate notification of constraint violation or can manually request agents run model checking.

Understanding the behaviour of the architecture of a system like the travel planner when it is running is challenging. To help architects validate the run-time properties of their architectures, we capture low-level object events (method calls, property changes, component events) from running systems that are forwarded to SoftArch (4). In SoftArch, OOD-level architecture components are located based on event annotations and information about the running system is passed to their abstractions (i.e. higher-level components). Static SoftArch visualisation views are copied and annotated to convey this running system information to developers e.g. to highlight created/not created processes, indicate number/size/timing of messages between components etc (5).

We deliberately designed SoftArch not to be a complete CASE tool, but rather to share information with CASE tools and programming environments. Import/export tools support linkage between SoftArch and OOA, design and implementation tools. OOA models allow software architects to capture functional and non-functional requirements

in SoftArch and ensure software architecture models meet these, or at least are annotated with this information (6). For example, travel planner functional requirements might be imported from Rational Rose™ OOA descriptions. Partial OOD models and some code fragments (implementing socket protocols, database access, ORB API calls etc.) are exported from bottom-level architecture components e.g. to OOD CASE tools or programming environments, like JBuilder™, to implement the travel planner system (7). Reverse engineering of OOD models into SoftArch allows developers to abstract higher-level architectural models from their code.

5 Static Architecture Modelling

In this and the following sections we describe and illustrate the static architecture modelling, architecture analysis and dynamic architecture visualisation capabilities of SoftArch. Consider again an architect wanting to design a software architecture for the travel planning system from Section 2. The architect needs some (ideally extensible) architectural abstractions to work with and a visual modelling notation to represent these abstractions in multiple views of the architecture. These views provide perspectives on the architecture at varying levels of detail.

5.1. Modelling Notations and Meta-Model

SoftArch uses the concepts of architecture *components*, *associations* (connections) between components, and *annotations* on components and associations. The types of component architects might use include processes, data stores, data management processes (e.g. database servers), machines and devices, and OOA and OOD-level objects and classes. Associations include data usage associations, event notification/subscription, message passing, and process synchronisation links. Annotations include data used, events passed, messages exchanged, protocol used, caching, replication and concurrency information, process control information, ports and so on. Each of these architectural elements can have associated properties. These could include information on services, security approaches, data size, transaction processing speed, data, message and event exchange details, and so on. Property values can be simple numbers, enumerated values, strings or value range constraints.

Architectures are made up of complex, inter-connected *elements* (i.e. components, associations and annotations). Visualising these inter-connected parts provides architects with key viewpoints on their architecture's design. To give this perspective, we provide architects with a visual architecture modelling language to represent architecture elements. Fig. 4 (a) shows some of the basic notational elements in our architecture modelling visual language. This notation has been developed over several years to represent various architectural abstractions in both our teaching and research projects [2, 8]. We chose this visual language for architecture modelling to enable developers to capture a wide range of features, to be relatively simple yet expressive, to be relatively easy to extend as needed, and to be able to tailor the appearance of visual elements to their needs. A wide variety of notational symbol and display characteristics can be changed by architects, such as iconic appearance, size, colour, shading etc. A UML-style representation of architecture using deployment and component diagram-like icons is also supported [11, 12], though we have found that these lack sufficient expressive power and diversity for most architecture design.

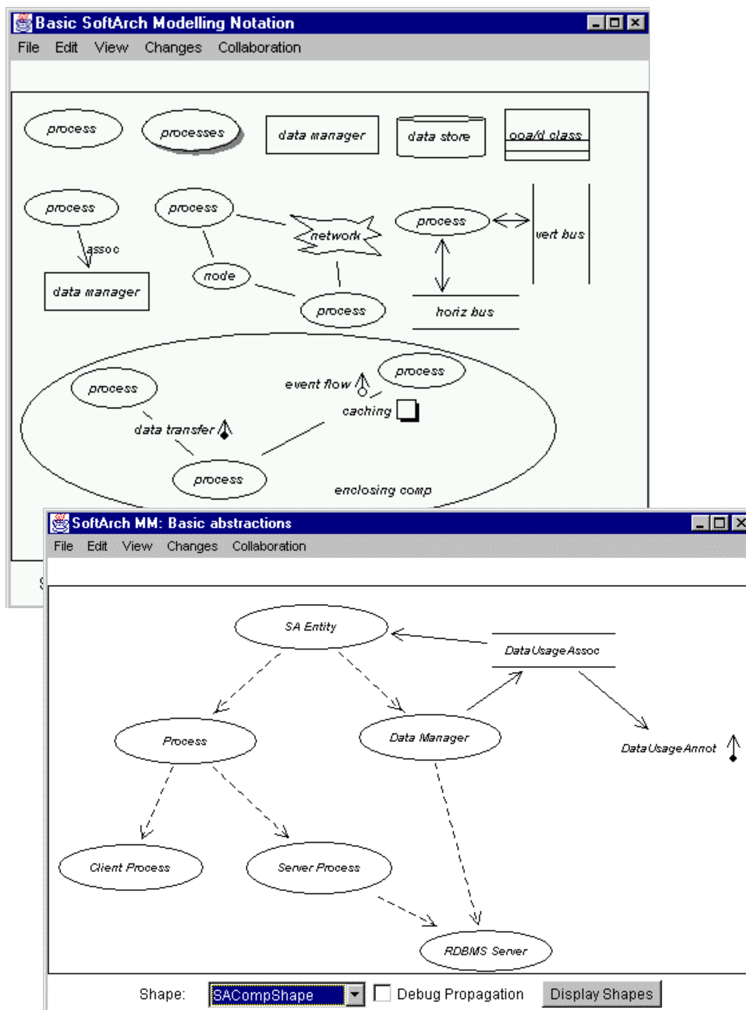


Fig. 4. (a) Some examples of our SoftArch visual modelling notation, and (b) part of a SoftArch meta-model.

The nature of software architecture design means architects often want to incorporate new modelling abstractions (new types of components, associations and annotations) into their models for different problem domains, or to better capture important abstractions. A way of doing this is to allow architects to extend the modelling abstractions available (and visual notations used to represent these) within SoftArch. We use a software architecture meta-model to describe all of the types of components, associations, annotations and properties of these different elements available for use by an architect. To enable architects to easily extend this meta-model SoftArch provides a simple visual language to describe the meta-model, illustrated in Fig. 4 (b). Ovals represent architecture component types, horizontal bars inter-component association types, and labelled vertical arrows annotation types. Dashed,

arrowed lines between types indicate refinement e.g. a process can be refined into a client or server process. Solid arrowed lines indicate association relationships e.g. a data manager may have data usage relationships with any architecture element. For example, when developing the travel planning system, an architect may find a useful architecture abstraction is missing e.g. web server, servlet, desktop application, http protocol etc. In order to use this abstraction during modelling the architect may choose to add this element type to the meta-model, refining it from an existing, more generic element, and linking it to other elements. Properties and constraints can be specified for the element type to enable detailed description and reasoning about its correct usage by analysis tools.

5.2. Architecture Modelling Example

To illustrate the use of the SoftArch notation and environment for architectural modelling, consider the modelling of the travel planning application described in Section 2.

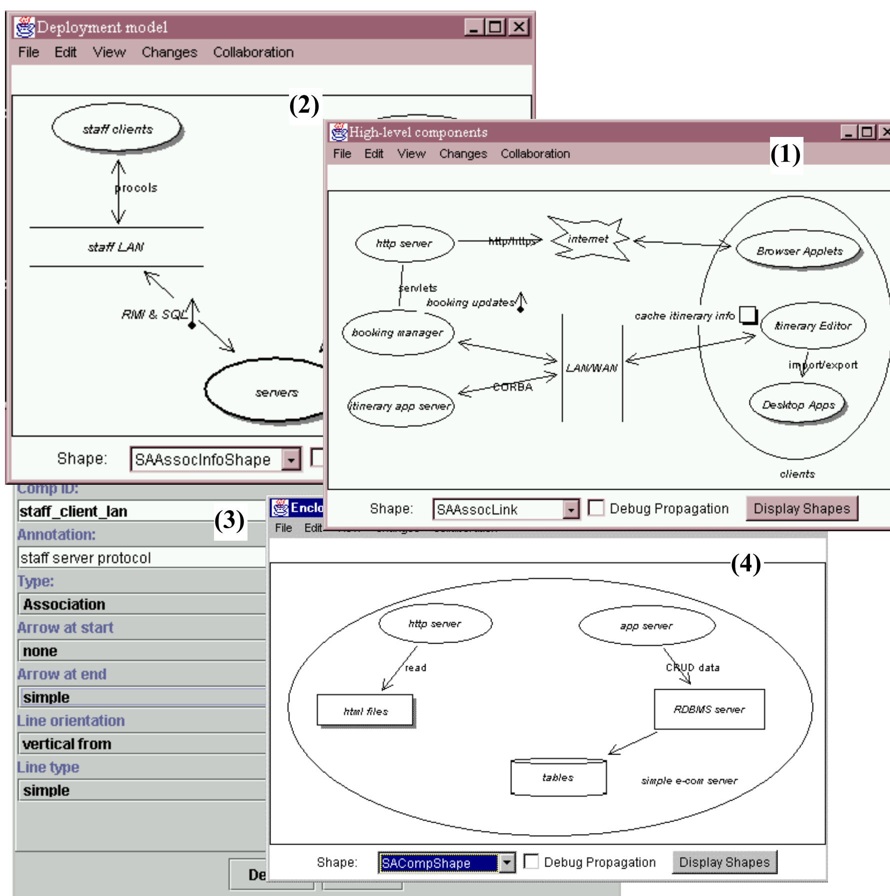


Fig. 5. Examples of architecture modelling in SoftArch.

To begin with, an architect initially imports an OOA functional and non-functional specification from a CASE tool into SoftArch or defines this information directly into SoftArch itself (using simple OOA-level class and function element types). The architect then sketches out a high-level architectural model for the planned system, ensuring the general characteristics of this model meets the OOA specification. For example, the travel planner has to support a number of concurrent users, customers require a web interface, travel planner components need to communicate with both client-side desktop applications and server-side enterprise applications, and various data processing, network and host machine characteristics need to be adhered to by the architecture (performance, reliability, cost and so on).

Fig. 5 (1) and (2) show two such high-level views of the travel itinerary planning system architecture. In (1) the architect has represented the parts of the system as three groups of "processes" – "staff clients", "customer clients" and "servers". They have indicated the staff client applications are connected to the servers via a LAN association, the customer clients via a WAN (i.e. internet) association. Annotations add further information such as the expected protocols for communicating with the servers. In (2), the architect has represented the major server-side and client-side processing components making up the system and high-level associations and annotations between these. Such views allow software architects and system designers to describe the basic architectural approach of the system using simple architectural elements. Some elements may appear in more than one view, and some views may show both structural characteristics and dynamic event/message/data exchange. In Fig. 5 (3) the architect is viewing/setting properties associated with an association between staff clients and the enterprise servers, which may include visual appearance and non-functional properties of the element.These architecture diagrams are not always built from scratch. Reusable template views, such as that shown in Fig. 5 (4) provide a means for them to reuse best-practice or common architectural structures. In this example the architect considers reusing a simple server-side "e-commerce" system organisation, made up of http, application and RDBMS servers and associated data. Architects can copy any view for reuse as a template and may select an appropriate template and have SoftArch copy this into their project, automating linking of abstract elements to new refined elements copied from the template. Change management between templates and copied views is supported [32].

5.3. Architecture Refinement

Once an architect has designed high-level architectural views capturing the essence of their system architecture, they usually wish to flesh this high-level architecture out in more detail, ultimately down to partial OOD-level class and relationship abstractions. Such refinement allows a system to be visualised from multiple perspectives, some showing basic architectural elements, others detailed views of parts of a system, with traceability supported between high-level and low-level abstractions. There are three ways to refine an architectural model in SoftArch: enclosing components within another (all enclosed elements are refinements of the encloser), adding sub-views for an element (all elements in the view are refinements of the view owning element), and specifying explicit refinement links between elements.

For example, the architect may decide to further specify what “staff clients” are required, and so create a sub-view for the “staff clients” component in Fig. 5 (1). Fig. 6 (1) shows this sub-view. All components in this view, except for “staff lan”, are refinements of the higher-level architecture component (“staff clients”) which owns the sub-view. This allows architects to “hide” this level of detail and drill down to it by double-clicking the “staff clients” icon to show its refinements. A component may have several sub-views, with refined components shown in more than one sub-view. In this example, “staff clients” is refined to “staff booking client”, “staff itinerary editor” and “staff desktop apps” processes. Annotations indicate a CORBA protocol supports booking client to server communication, the itinerary client caches itinerary data and itinerary update events are "pushed" to the itinerary editor. The “staff lan” component is shown in this refinement for context (what the “staff clients” sub-components are connected to) but the “<…>” component name annotation indicates it’s a linkage component and not a refinement component in this sub-view.

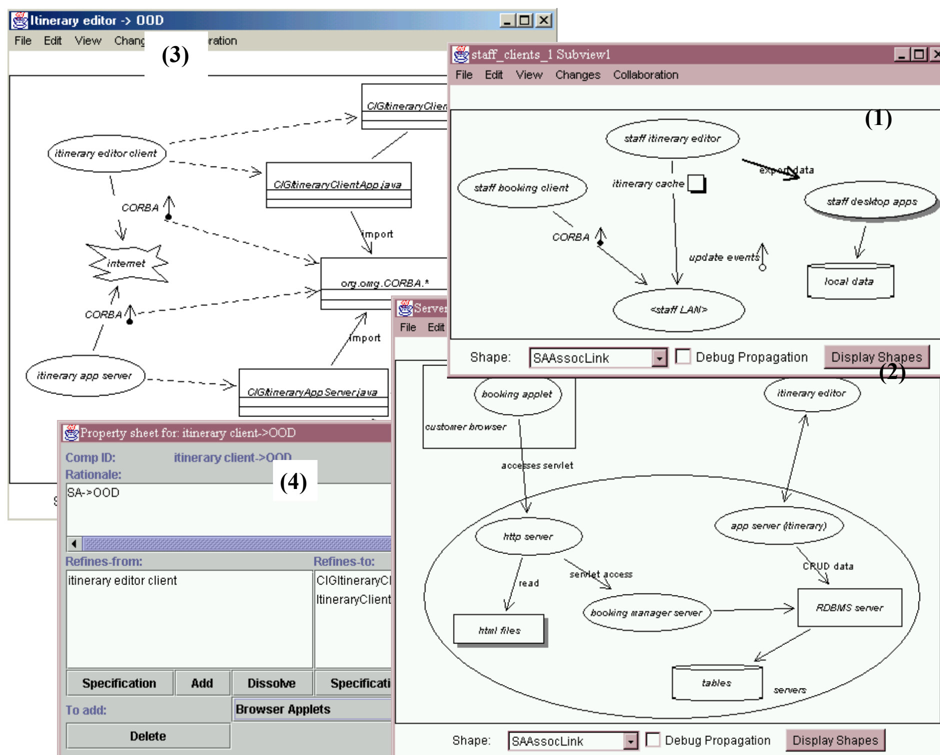


Fig. 6. Example architecture refinements in SoftArch.

The software architect has made two further refinements in this example. In diagram (2), the “servers” component from Fig. 5 (1) has been refined by using it to enclose more specific server-side components, relationships and annotations. All enclosed components, associations and annotations are refinements of the “servers” component. The architect has chosen to use the enclosing of other components so the context of the refinement is shown on the diagram with the refined components. Note

that this refinement has reused the template architecture from Fig. 5 (4), copying the template and the architect renaming and adding to it for use in this modelling application. In diagram (3), several architecture components describing the itinerary management part of the system, on the left hand side, have been refined to OOD-level class components (using UML class diagram notation) on the right hand side. This was done by the architect adding explicit refinement links (the dotted arrows). The dialogue shows refinement information for the itinerary_client to OOD classes refinement. OOA-level classes and services can also be described in SoftArch (usually imported from a CASE tool rather than defined in SoftArch itself) and refinement links from OOA classes to architectural components can be made.

SoftArch provides a number of additional modelling facilities our architect may choose to use. These include dynamic behaviour representation, showing components, connectors and dynamic information annotations including data, message, control and event flow/relationships and timing. We have also incorporated some PARSE-DAT and ViTABaL [4, 2] dynamic component assembly constructs, allowing architects to model architectures whose components and connectors evolve at run-time. Our software architect can also model the provided and required services of components and connectors, allowing us to check (from static design models, at least) that these are met. Dynamic properties of components can be modelled and compared against actual performance and other run-time collected measurements.

6 Architecture Analysis

Supporting modelling of software architectures and refinements is not sufficient to enable software architects to produce quality, consistent architecture models for complex systems. Software architecture analysis tools are also needed, including support for checking such things as: all components are linked to others, all components are suitably refined from OOA-level specifications to OOD-level class realisations, sensible and consistent associations and annotations have been used, valid property values have been set, and provided and required services between linked components are consistent.

For example, our architect may have specified a variety of views of their architecture as outlined in the previous section. However, a number of problems may be present in these architecture designs:

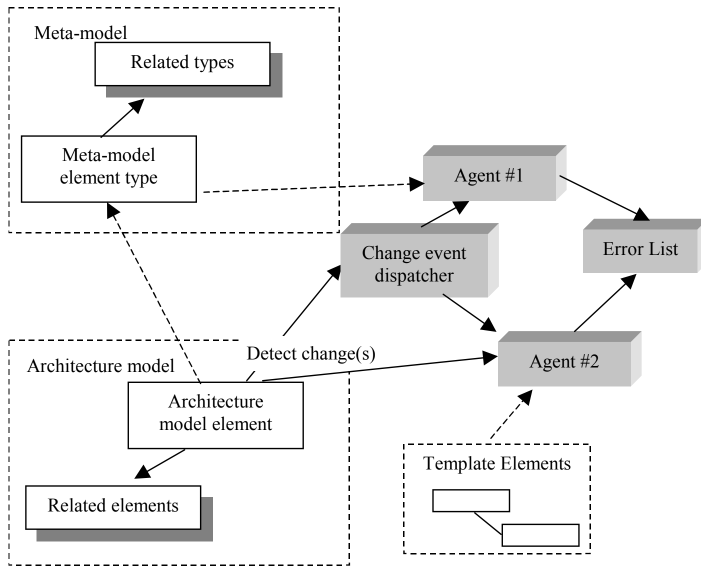
- Some architecture elements may not be associated with others or may be associated in invalid ways.
- Some elements may not have all required property values specified for their types.
- Some architecture elements may not be refined from higher-level components or refined to lower-level components. This indicates inconsistency between refinement levels.
- Some elements may require specific kinds of services/properties from related elements, but these are not provided.
- Some architecture elements may be invalidly refined from or to others.
- Some elements may provide services or properties that are not used by related components and should be.

Our meta-model architecture modelling types specify various types of validity information (valid associations, properties and property value ranges, refinement

relationships and so on). In addition to checking such architecture element type usage correctness, the architect may want to be provided with feedback on the use of various architecture elements or parts of models in various situations i.e. usage guidelines.

In order to assist architects in static validation of their architecture models SoftArch provides an extensible set of model analysis agents. These monitor changes to an architecture model and give feedback in various ways to the software architect - immediate report of error; unobtrusively adding error notes to an "error list"; or producing a report when the architect requests one. Fig, 7 shows basic approaches our model checking agents use to detect and report on errors. Some agents e.g. Agent #1 simply detect a change to a model element (or changes to elements related to that model element) against meta-model type information. For example, an agent may check if the component has all required property values set, is refined from another element, or has required associations to/from other kinds of elements (and these are valid). The agent reports any discrepancies between the meta-model element type specifications and the model element instances it can find. Agents can subscribe to changes from single architecture model elements or groups of related elements. Agents can filter out changes they are not interested in e.g. an association checking agent only detects "establish or dissolve relationship" events. We use a change event dispatcher to detect model changes based on element type, forwarding these to subscribing model checking agents.

Other model checking agents, such as Agent #2, may detect changes to one or more components and then compare their inter-connectivity and property values against the agent's "correctness" template, reporting any error(s). These templates are simply SoftArch architecture modelling templates, like Fig. 5 (4), with a property for all components, associations, refinement links, annotations saying if they are optional/required, and additional element property constraints. The checking agent validates the changed model elements by comparing them against the template. For example, a "valid E-commerce architecture" checking agent may check for the presence of a web server, application server, database and suitable connections, possibly with suitably constrained element properties.



Fig, 7. Analysis agent control, reporting and visual specification.

Fig. 8 illustrates how our software architect can control agent behaviour and view agent error message reports via an agent control panel (shown on the left). In this example, the architect has all of the agents configured as "critics" i.e. agents watch model changes and add messages to a critic message dialogue (shown at the bottom). Several example messages are shown, indicating various discrepancies between meta-model type specifications and actual usage of architecture elements in the model. The architect can ignore these and continue modelling, select one of these and correct it, or correct a number of these errors and leave others. Inconsistent architecture models can be modified with inconsistencies tracked in this way.

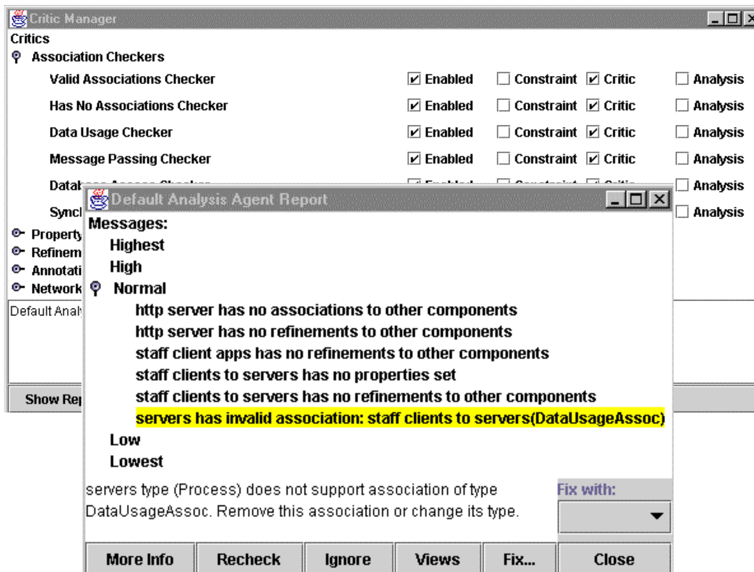


Fig. 8. Example of architecture design critics.

7 Dynamic Architecture Visualisation

Once an architecture model design is complete, the software architect typically exports partially specified OOD-level components to a CASE tool and/or programming environment. Developers complete the implementation using these tools. Software architects may then want to analyse the actual behaviour of their implemented architecture, or to analyse a reverse-engineered architecture, using SoftArch-style abstractions. For example, after completing travel system implementation our architect may want to study the behaviour of their architecture in practice, possibly to identify and correct problems, possibly to record performance and other characteristics for use when developing other architectures in the future. In order to support dynamic architecture analysis we have developed support for monitoring and visualising performance information within SoftArch. The approach we use is to capture running system events (such as method calls, object creation, time to complete method call etc), and forward these events to SoftArch, tagging them with information about the lowest-level SoftArch model element they are associated with (OOD class/method, low-level architecture abstraction etc). We then aggregate these events within SoftArch i.e. associate events with SoftArch elements and then "pass them up" refinement hierarchies, summing them at each level in the hierarchy. This gives a multi-level analysis of overall running system performance measures. We present this aggregated performance information to the architect using annotated design diagrams (though architects can also develop diagrams specifically for aggregating performance information in different ways).

For example, after starting up the itinerary editor servers and one itinerary client staff application, Fig. 9 (1) shows a dynamic visualisation using a top-level architectural view in SoftArch. This visualisation represents the number of components created so

far. The servers component is dark (five server-side objects created), staff clients is lightly shaded (one staff application is running) and customer applets very lightly coloured (no objects of types that are refined from this high-level component running on non-staff hosts have been created). This kind of visualisation is useful for software architects to determine what processes in an architecture have so far been created, and to determine relative densities of objects etc in their realised architecture. Views can be animated to show density increasing as a system runs. As with other visualisations, scaling is used to show relative object and process creation densities, data transfer totals and number of events exchanged, and so on. The architect can change properties of the visualisation (colour, scaling, elements updated, frequency of update etc) as they require.

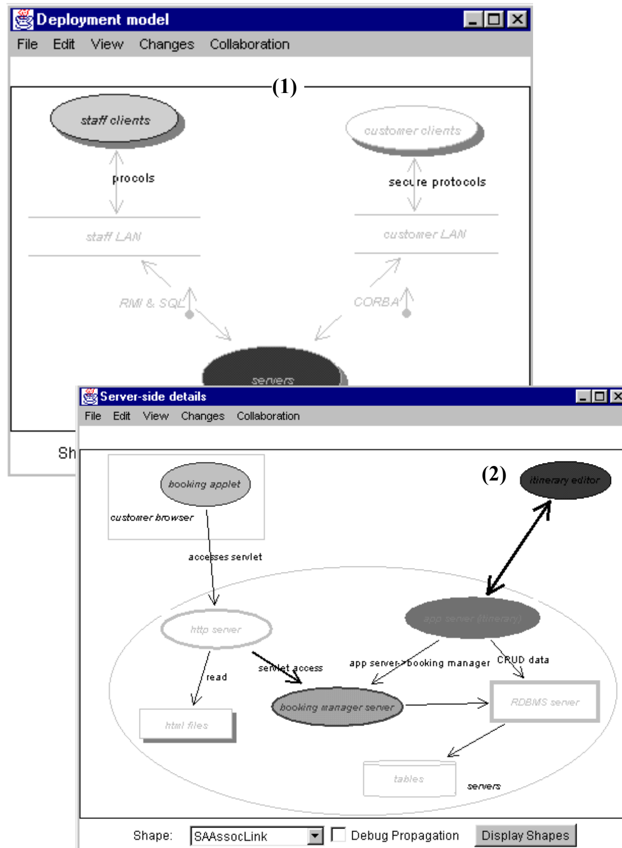


Fig. 9. Dynamic visualisation of running system in SoftArch.

Fig. 9 (2) shows the server-side view of the itinerary planning system, annotated to illustrate relative method calling and event propagation densities. This helps our architect identify bottlenecks and performance problems. Association line thickness for indicates method/event propagation density across the relationship (however implemented). Border thickness of component icons indicates number of methods/events in, while background colour indicates internal method/event calling. This visualisation shows the architect that the booking applet and booking manager

server are moderately used and perform a limited number of internal calls. The Itinerary editor and its application server perform many more internal calls, and in this scenario generate more client<->server interactions. The RDBMS is moderately heavily used (caching in the application servers reduces its load), while the http server's servlet is moderately used and performs few internal calls.

Sometimes the software architect wants point-value information about specific architecture element performance measures, as shown in Fig. 10 (1). This shows data for the application server objects at a snap shot in time. Summarised information, in this example a bar graph of number of method calls to the "itinerary app server" component, an object making up the application server process, has been generated by exporting data to MS Excel™, shown in Fig. 10 (2). Developers can request that inter-component communication tracing information be recorded for selected architectural elements, and can review these. Fig. 10 (3) shows an example of such information presented in a dialogue. Each method invocation of the application server has been recorded, and the developer can examine this trace.

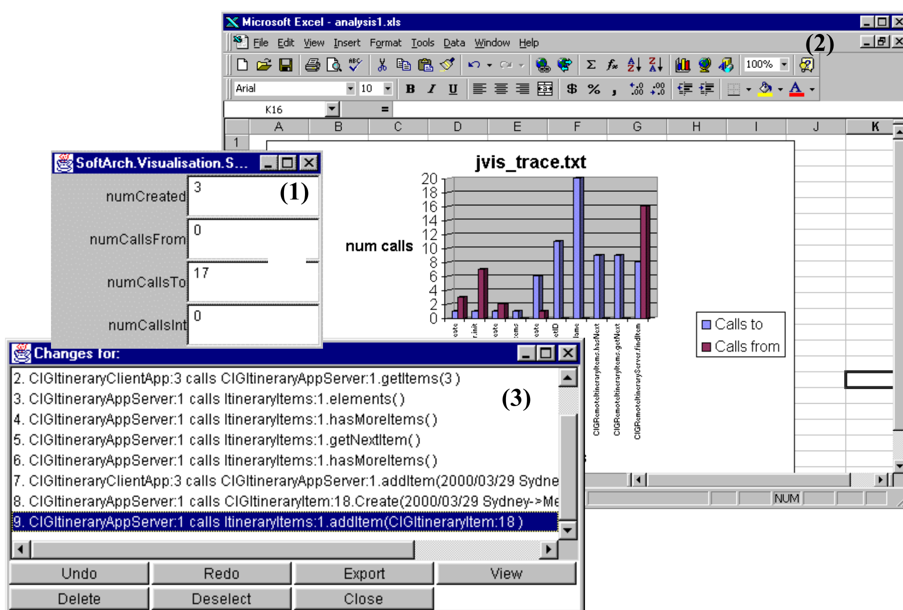


Fig. 10. Detailed and summarised performance information.

8 Design and Implementation

Figure 11 illustrates the basic architecture of SoftArch which comprises:

- The SoftArch modelling and meta-modelling tools. These provide a set of meta-model architecture element type abstractions, reusable model templates and software architecture model views.

- Serendipity-II workflow system [32]. This provides process models and project plans for guiding use of SoftArch, and supports visual plug-and-play of analysis agent parts for use by SoftArch.
- JComposer component-based CASE tool [9]. JComposer provides abstractions for modelling the specifications and designs of software components, and we use JComposer to complete design and implementation of SoftArch architecture models. JComposer also allows reverse-engineering of architecture designs from existing component code (currently Java files).
- Import/export tools for communicating with 3rd party CASE tools. To date, we have built XMI-based tools for communication with Argo/UML and a comma-separated value data exchange tool with MS Excel™.
- The JVisualise dynamic component debugging tool [9]. JVisualise allows us to monitor running system events and aggregate these into SoftArch. It also allows us to perform limited manipulation of running component-based systems.
- Communication/event handling by the JViews software bus [23].

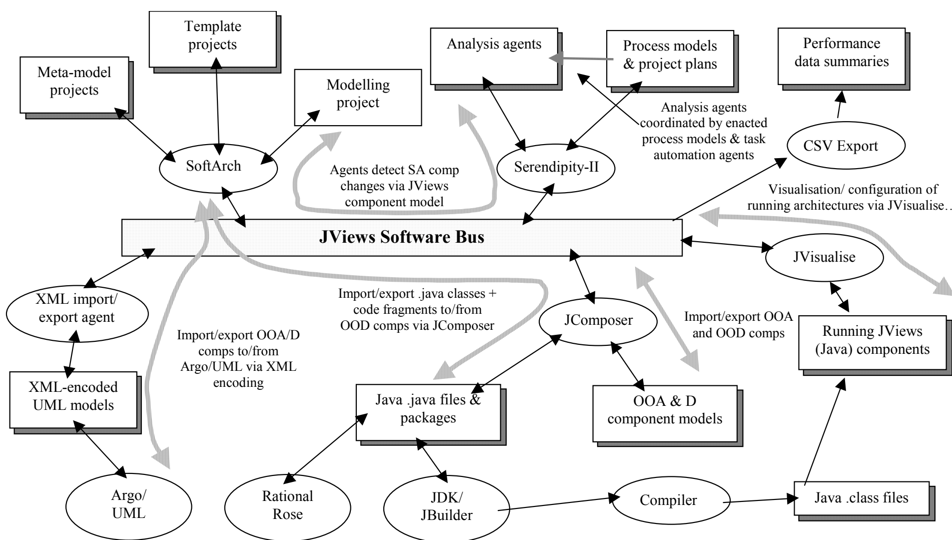


Figure 11. Architecture of SoftArch and related tools.

SoftArch's meta-model, model and visual editing views are implemented using our JComposer tool's meta-CASE framework, which generates classes that specialise our JViews component-based architecture for multi-view, multi-user environment construction [9]. SoftArch is thus a component-based system and able to be integrated with other component-based tools by JViews component facilities. SoftArch provides multiple views of software architecture models with a centralised repository and view consistency mechanism. It provides a variety of collaborative work facilities, including synchronous and asynchronous editing of views, version merging and configuration management.

SoftArch maintains a set of meta-model projects that define the allowable components, associations, annotations and property types for a model. A set of reusable refinement templates (which are copiable SoftArch model views) allow reuse of common architectural refinements. A modelling project holds the model of the software architecture currently under development, including all views, architectural elements and projects. This information is represented as JViews software components using a three-level architecture, as illustrated in Figure 12. The bottom layer is a canonical representation of data; the middle view information shown in each view; and the top a set of user interface components forming an editor and icons. The extensible meta-model can be visually extended allowing new modelling abstractions and constraints to be dynamically added and reused. The templates can be copied and instantiated into a model for reuse, with changes able to be propagated between model and source template. The graphical user interface icons and connectors used by SoftArch are user-tailorable, allowing architects to change and extend the appearance of their modelling views. This is particularly important if architects add new meta-model abstractions - they can also extend their appearance of the SoftArch visual icons to distinguish new modelling element types from others.

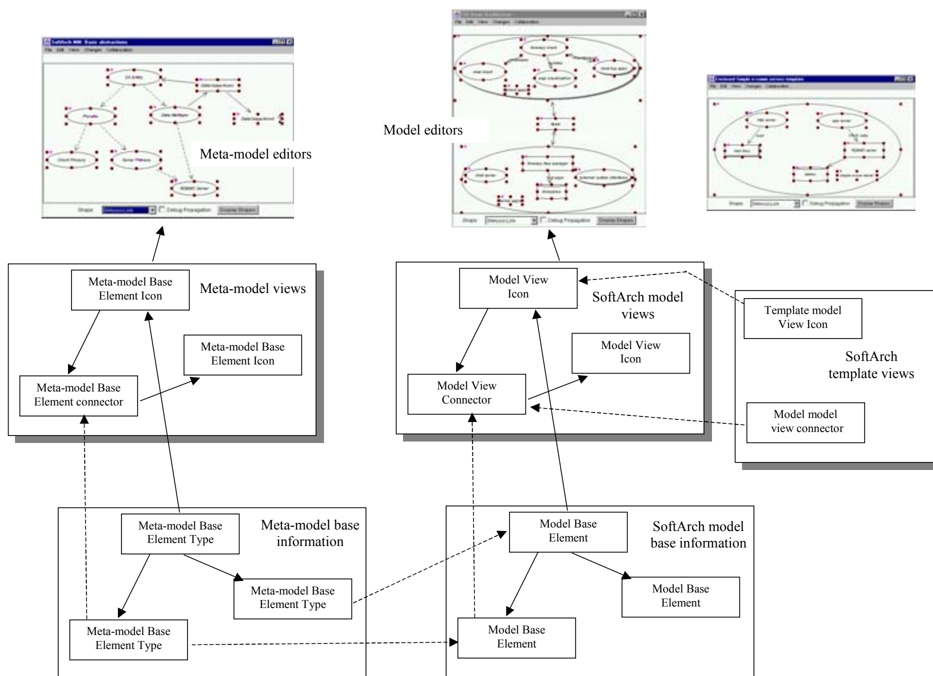


Figure 12. 3-level architecture of SoftArch.

Many tools exist which provide object-oriented analysis and design capabilities. Our own JComposer is one such example, but others include CASE tools like Rational Rose [11] and Argo/UML [12]. We originally planned SoftArch as an extension to JComposer, but decided it would be more useful as a stand-alone tool, that could ultimately be used in conjunction with other, 3rd party CASE tools. SoftArch requires constraints from an OOA model, particularly non-functional constraints like

performance parameters, robustness requirements, data integrity and security needs and so on. These constrain the software architecture model properties that needs to be developed in order to realise the specification. These also influence the particular architecture-related design decisions and trade-offs software architects need to make. Similarly, a SoftArch architectural model is little use on its own, but needs to be exported to a CASE tool and/or programming environment for further refinement and implementation. Some code generation can potentially be done directly from a SoftArch model description e.g. some middleware and data management code. When reverse engineering an application, an OOD model will need to be imported into SoftArch and a higher-level system architecture model derived from it. Ultimately an OOA specification may be exported from SoftArch to a CASE tool. Thus SoftArch must support OOA and OOD model exchange with other tools, and ideally some code generation support.

JComposer and SoftArch interact to achieve OOA import and OOD design export and code fragment generation for SoftArch. Generated .java class source code files can be used in tools like JDK and JBuilder, and changes reverse engineered back into JComposer and then into SoftArch. We initially used a JComposer component model as the source for SoftArch OOA-level specification information. JComposer allows not only functional requirements to be captured, but has the additional benefit of requirements and design-level component "aspects", which are used to capture various non-functional requirements. We developed a component that supports importation of basic component and aspect information into SoftArch from a JComposer model, using JViews' inter-component communication facilities to link SoftArch and JComposer. However, rather than add OOD and code generation support to SoftArch itself, we leveraged existing support for these in JComposer. SoftArch uses JComposer's component API to create OOD-level components (classes) in JComposer, and instructs JComposer to generate code for these to produce .java files. We have prototyped a data interchange mechanism to enable SoftArch to exchange OOA and OOD models with Argo/UML using an XML-based encoding of UML models.

We use our JVisualise component monitoring tool to request running components send it messages when they generate events. SoftArch instructs JVisualise to send it these low-level component monitoring events, which are mapped onto SoftArch OOD-level architecture elements using JComposer-generated Java class names. SoftArch allows users to view information about running components using higher-level SoftArch views, as OOD-level components in SoftArch must have refinement relationships to higher-level architecture elements in these views. We extended JComposer-generated OOD models and code to include additional monitoring components to intercept data and communication messages and to annotate these with the source SoftArch elements to which low-level generated component events are related. JVisualise uses these to provide its event and message monitoring and control support.

9 Experience

We have used SoftArch to model a number of small and medium-sized system architectures. These have included the travel planner discussed here, a business-to-customer on-line retailing system, an on-line video store library system, and an on-line micro-payment system. We have also used it to help reverse-engineer the architectures of several component-based, distributed systems. We have used SoftArch as the

architecture modelling tool component of SoftArch/MTE, a performance test-bed generator which takes SoftArch models and generates performance test-bed code [33]. This includes the generation of JSP, ASP, CORBA, Enterprise JavaBean and .NET implementations of SoftArch-modelled systems, with stubs generated for clients, servers and database access code to allow architecture and middleware performance analysis to be automatically carried out.

We have carried out two usability evaluations of SoftArch to assess its support for architecture development. These involved a combination of graduate students, researchers and industry practitioners modelling system architectures, and in the second scenario generating performance test-bed code for analysis. These evaluations have demonstrated SoftArch provides a number of useful facilities not found in comparable CASE or development environments. These include the ability of designers to refine architecture models in various ways that supports much richer architectural representation and reasoning and traces architectural design decisions clearly from OOA to OOD. Analysis agents that provide incremental feedback to architects while tolerating varying amounts of inconsistency during architecture design allow for more flexible architecture development while providing continuous validation guidance. Visualisation of running systems using high-level abstract views provides much easier to understand performance information and more rapid feedback than most other approaches. When coupled with our performance test-bed generator component, the SoftArch architecture visualisation support can also be reused to provide high-level visualisation of automatically-generated performance analysis tests. Users of SoftArch liked its flexible design notation but commonly suggested using "more UML-style notations". They also found the import/export of architecture models between different tools e.g. Argo/UML, SoftArch and JBuilder, to be cumbersome. The current visual language used to define architecture design critics was found to be too difficult for most users of SoftArch surveyed.

We are working on characterising a wider range of architectural components for various domains, such as for embedded systems, real-time systems and E-commerce systems, and defining meta-models, notational symbols and reusable templates to enable easier modelling of such systems. We are building further analysis agents to make better use of architecture component performance measurements, giving developers improved estimates of likely run-time behaviour of architecture models. We plan to incorporate more structured architectural properties, characterised using our aspect-oriented engineering method [8], allowing developers to characterise their architectural component and association characteristics using systemic aspects and for SoftArch to perform consistency analysis of inter-aspect relationships. The JComposer-based code generation and monitoring is quite effective at providing developers with high (and low) level dynamic architectural component performance and trace information. However, it requires our JViews-based component architecture be used to realise running distributed applications. We are currently working on a code generator that uses XML-encoded UML models to generate parts of systems and 3rd party profiling tools whose traces will be acquired and aggregated by SoftArch to provide run-time performance information. We plan to use this code generation and profile aggregation to allow developers to rapidly develop middleware test bed systems to validate architectural model performance properties via rapid architecture prototyping and performance analysis.

10 Summary

Modelling, validating and visualising complex system architectures is a challenging development activity. SoftArch provides a set of tools enabling architects to model rich knowledge about their architectures using an extensible set of architectural abstractions and visual notations. The extensible SoftArch meta-model allows developers to define new, specialised architectural components for particular domains, while the tailorable visual notations allow developers to represent their architectures in a wide variety of ways. Architectural component refinement allows developers to refine their architectures from analysis-level specifications to design-level implementation descriptions in a controlled and traceable fashion. Architecture analysis in SoftArch uses basic consistency checks and comparison to best-practice element usage templates to help inform developers of the quality of their models. This provides support for proactive architectural refinement during modelling. Dynamic visualisation of running systems is supported by aggregating captured trace events and displaying this information by annotating static modelling view components. This approach presents developers with run-time architecture performance metrics in a context they can readily interpret.

Acknowledgements

Support for this research from the University of Auckland Research Committee and the New Zealand Public Good Science Fund is gratefully acknowledged. The helpful comments of the anonymous referees on earlier drafts of this paper are also appreciated.

References

1. T.C.N. Graham, C.A. Morton, and T. Urnes, ClockWorks: Visual Programming of Component-Based Software Architectures. Journal of Visual Languages and Computing, (July 1996), 175-196.
2. J.C. Grundy, J.G. Hosking, ViTABaL: A Visual Language Supporting Design by Tool Abstraction, In Proc. of the 1995 IEEE Symposium on Visual Languages, Darmsdart, Germany, September 1995, IEEE CS Press, pp. 53-60.
3. R. Kazman, Tool support for architecture analysis and design, In Proc. of the Second Int. Workshop on Software Architectures, ACM Press, 94-97.
4. A. Liu, Dynamic Distributed Software Architecture Design with PARSE-DAT, In Proc. of the 1998 Australasian Workshop on Software Architectures, Melbourne, Australia, Nov 24, Monash University Press.
5. L. Bass, P. Clements, and R. Kazman, Software Architecture in Practice, Addison-Wesley, 1998.
6. A. Egyed and P. Kruchten, Rose/Architect: a tool to visualize architecture, In Proc. of the 32nd Hawaii Int. Conf. on System Sciences, January 1999, IEEE CS Press.
7. G. Booch, J. Rumbaugh, and I. Jacobson, The Unified Modelling Language User Guide, Addison-Wesley, 1999.
8. J.C. Grundy, Supporting aspect-oriented component-based systems engineering, In Proc. of 11th Int. Conf. on Software Engineering and Knowledge Engineering, Kaiserslautern, Germany, June 16-19 1999, KSI Press, pp. 388-395.
9. J.C. Grundy, W.B. Mugridge, J.G. Hosking, Static and dynamic visualisation of component-based software architectures, In Proc. of 10th Int. Conf. on Software Engineering and Knowledge Engineering, San Francisco, June 18-20, 1998, KSI Press.

10. T. Urnes and T.C.N. Graham, Flexibly Mapping Synchronous Groupware Architectures to Distributed Implementations. In Proc. of Design, Specification and Verification of Interactive Systems (DSV-IS'99), 1999.
11. T. Quatrani, Visual Modeling With Rational Rose and Uml, Addison-Wesley, 1998.
12. J. Robbins, D.M. Hilbert, and D.F. Redmiles, Extending design environments to software architecture design, Automated Software Engineering 5 (July 1998), 261-390.
13. N. Stankovic and K. Zhang, K. Towards Visual Development of Message-Passing Programs, In Proc. of 1997 IEEE Symposium on Visual Languages, IEEE CS Press.
14. B. Shizuki, M. Toyoda, E. Shibayama, and S. Takahashi, Visual Patterns + Multi-Focus Fisheye View: An Automatic Scalable Visualization Technique of Data-Flow Visual Program Execution. In 1998 IEEE Symposium on Visual Languages, Halifax, Canada, September 1998, IEEE.
15. J.C. Grundy and J.G. Hosking, Directions in modelling large-scale software architectures, In Proc. of the 2nd Australasian Workshop on Software Architectures, Melbourne 23rd Nov 1999, Monash University Press, pp. 25-40.
16. J. Leo, OO Enterprise Architecture approach using UML, In Proc. of the 2nd Australasian Workshop on Software Architectures, Melbourne 23rd Nov 1999, Monash University Press, pp. 25-40.
17. M. Beaumont, and D. Jackson, Visualising Complex Control Flow. In 1998 IEEE Symposium on Visual Languages, Halifax, Canada, September 1998, IEEE.
18. T. Hill, J. Noble, Visualizing Implicit Structure in Java Object Graphs, In Proc. of SoftVis'99, Sydney, Australia, Dec 5-6 1999.
19. S.P. Reiss, A framework for abstract 3-D visualization, In Proc. of the 1993 IEEE Symposium on Visual Languages, IEEE CS Press.
20. R.J. Walker, G.C. Murphy, J. Steinbok, and M.P. Robillard, Efficient Mapping of Software System Traces to Architectural Views, In Proc. of CASCON'2000.
21. R. Allen and D. Garlan, A formal basis for architectural connection, ACM Transactions on Software Engineering and Methodology, July 1997.
22. M. Shaw and D. Garlan, Software Architecture : Perspectives on an Emerging Discipline, Prentice Hall, 1996.
23. J.C. Grundy, W.B. Mugridge, J.G. Hosking and M.D. Apperley, Tool Integration, Collaboration and User Interaction Issues in Component-based Software Architectures, In Proc. of TOOLS Pacific'98, Melbourne, Australia, Nov 28-30 1998, IEEE CS Press.
24. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, P. Pattern Oriented Software Architecture : A System of Patterns, Wiley, 1996.
25. B. Wagner, I. Sluijmers, Eichelberg, D. and P. Ackerman, Black-box Reuse within Frameworks Based on Visual Programming, In Proeedings of the. 1st Component Users Conf., Munich, July 1996, SIGS Books, pp. 57-66.
26. A. Parashkevov and J. Yantchev, ARC - A Tool for Efficient Refinement and Equivalence Checking for CSP, In Proc. of the 1996 IEEE Int. Conf. on Algorithms and Architectures for Parallel Processing, Singapore, June 11-13, 1996.
27. D.C. Luckham, L.M. Augustin, J.J. Kenney, J. Veera, D. Bryan, and W. Mann, Specification and analysis of system architecture using Rapide, IEEE Transactions on Software Engineering **21**(April 1995), 336-355.
28. J.G. Hosking, Visualisation of object-oriented program execution, In Proc. of 1996 IEEE Symposium on Visual Languages, IEEE CS Press.
29. H. Koike, T. Takada, and T. Masui, VisuaLinda: A Framework for Visualizing Parallel Linda Programs, In Proc. of the 1997 IEEE Symposium on Visual Languages, IEEE CS Press.
30. P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Mueller, J. Mylopoulos, S. Perelgut, M. Stanley, M. and K. Wong, The Software Bookshelf, IBM Systems Journal 36 (November 1997), 564-593.

31. B. Topol, J. Stasko and V. Sunderam, PVaniM: A Tool for Visualization in Network Computing Environments, Concurrency: Practice & Experience **10** (1998), 1197-1222.
32. J.C. Grundy, J.C., Hosking, J.G., Mugridge, W.B. and Apperley, M.D. An architecture for decentralised process modelling and enactment, IEEE Internet Computing 2 (September/October 1998), IEEE CS Press.
33. J.C. Grundy, Y. Cai, Y. and A. Liu, Generation of Distributed System Test-beds from High-level Software Architecture Descriptions, In Proc. of the 2001 Int. Conf. on Automated Software Engineering, San Diego, Nov 25-28 2001, IEEE CS Press.