Component-based perspective on software mismatch detection and resolution

A.Egyed, N.Medvidovic and C.Gacek

Abstract: Existing approaches to modelling software systems all too often neglect the issue of component-mismatch identification and resolution. The traditional view of software development over-emphasises synthesis at the expense of analysis – the latter frequently being seen as a problem which only needs to be dealt with during the integration stage towards the end of a development project. The paper discusses two software modelling and analysis techniques, all tool supported, and emphasises the vital role analysis can play in identifying and resolving risks early on. This work also combines model-based development (e.g. architectural modelling) with component-based development (e.g. COTS and legacy systems), and shows how their mismatch-detection capabilities complement one another to provide a more comprehensive coverage of development risks.

1 Introduction

To be competitive, a developer's usage of commercial-off-the-shelf (COTS) packages has become standard, at times being an explicit requirement from the customer. The idea of simply plugging together various COTS packages and/or other existing/in-house-developed parts (as described in the megaprogramming principles [1]) is, however, often trivialised, as are the side effects that may occur by plugging or composing these packages together [2]. Likewise, the development of architectural and design-modelling languages (e.g. Unified Modelling Language [3]) has proceeded with little emphasis on new challenges imposed by component-based development.

This work introduces a two-tiered development methodology that combines independently created component-based and model-based development approaches [4]. The combined approaches complement one another in their ability to detect a larger (more comprehensive) number of mismatches that may happen between various component choices. The combined approaches also complement one another in the speed with which they provide mismatch feedback. More comprehensive mismatch analysis has the benefit of an increased likelihood that incompatibilities between components are identified before the actual software system is built, potentially preventing great loss of cost and effort. Rapid mismatch detection has the additional benefit of a fast exclusion of nonviable component configurations (architectures) for a desired system, again reducing the cost and effort required in analysing a potentially large set of components.

The approach described in this paper performs (i) a high-level component analysis followed by (ii) a more comprehensive component-enabled architectural analysis, excluding unfeasible architectural options along the way. Both analysis approaches detect mismatches and thus help human decision makers in eliminating unfeasible architectural options. The analysis approaches act as filters where unfeasible options are removed before the next analysis step. The first filter is fast and can provide feedback within hours, but is not very comprehensive. The second filter is more comprehensive and thus slower, but more reliable. To ensure continuity between them, a synthesis technique was also created to share modelling information (e.g use of analysis results of the first filter as an input to the second filter).

The fact that the analysis filters trade-off precision with speed has another beneficial side effect. During software development, we are (at times) confronted with both the lack of information and its abundance. When we deal with new issues, variables, changes etc. (e.g. new requirements) we usually lack information and the ability to analyse its impact seems limited. Conversely, once some aspect has been investigated, we are often confronted with a plethora of information spread across multiple models (e.g. documents, design models, implementation). The dual-filtering process can handle both situations. The first filter is found to be most effective early on in the software life cycle—as early as requirements negotiation—where choices of components, their features, and configurations are still very vague. The second filter requires more detailed information that becomes available later on. Architectural-analysis techniques [5] used in the second filter frequently also support refinement, making it possible to use the second filter recursively for even more detailed mismatch analyses if desired.

The contributions of this paper are an incremental, rapid and comprehensive approach to component-based software modelling, with a strong emphasis on component-

IEE Proceedings online no. 20000915

DOI: 10.1049/ip-sen:20000915

Paper first received 14th June and in revised form 1st November 2000

A. Egyed is with the Teknowledge Corporation, Distributed Systems Group, 4640 Admiralty Way, Suite 231, Marina Del Rey, CA 90292, USA
E-mail: aegyed@acm.org

N. Medvidovic is with the Computer Science Department, University of Southern California, 941 W. 37th Place, SAL 338, Los Angeles, CA 90089-0781, USA
E-mail: neno@usc.edu

C. Gacek is with Fraunhofer IESE, Sauerwiesen 6, 67661 Kaiserslautern, Germany
E-mail: cristina.gacek@iese.fhg.de

mismatch detection and consistent refinement. In the following, this paper will introduce and discuss the two-tier development approach. To complement the discussion, a nontrivial example is used to illustrate the approach. The two sections discussing the individual pieces of this approach are structured by giving an initial high-level and generic overview, followed by a demonstration in context of the example, and concluded with a follow-up discussion on how other scenarios may require a different treatment than that given in the example. Owing to the limited space, references are provided whenever the details cannot be discussed in this paper.

2 Approach

Simply speaking, a typical software-development process is seen as consisting of requirements modelling, evaluation of architectural options, architectural modelling, designing, and finally coding (Fig. 1).

Section 4 deals with the exploration of different possible realisations of the requirements. At this level, functional details are not yet (fully) known. Instead, the focus is on coarse-grain modelling, often also involving outside components such as COTS or legacy systems. Analysing architectural options tends to evaluate the suitability of components to work together and solve the proposed problem.

Since it would be too expensive to model each architectural option in detail, only a few suitable options should be selected for further modelling and evaluation. Section 5 [5, 6] focuses on modelling the interior workings and interactions of all to-be-developed parts of a proposed system. Architectural modelling partitions the system into coherent subsystems and describes the presumed interactions between those subsystems and involved external components (e.g. COTS, humans etc.). Analysis of architectures focuses on the evaluation of roles and responsibilities of architectural components as well as their interaction protocols.

Section 6 refines architectural modelling by using lower-level constructs conceptually closer to the implementation. Thus, the design supplements the architecture by modelling how roles, responsibilities, and interaction protocols are actually realised. The design typically also extends the architecture by further subdividing architectural components into smaller pieces and providing additional details for them. Analysis of designs primarily focuses on evaluating the consistency between the design and its architecture so that the design faithfully realises the architecture and is internally consistent.

The two-tier development approach has a strong emphasis on mismatch detection. Mismatches are introduced during software development and evolution at various levels while combining models (diagrams) into system representations, and components into systems. In [4] it has been shown that component integration and model integration are two complementary activities. Components (e.g. libraries, COTS, legacy) are used as building blocks to create more complex components or systems. Component-based development decreases subsystem dependency and increases (component) reuse. Likewise, models (diagrams) are used as building blocks for creating more complex representations of systems. Model-based development decreases modelling complexity and increases the separation of concerns.

The primary benefits of using this approach to component-based development are the combination of early risk assessment of mismatches (e.g. incompatibilities) between (off-the-shelf) components, the exhaustive modelling of components (and their interconnections) for detailed mismatch analysis, and the well defined refinement process for implementing component wrappers (e.g. 'glue code') and additional functionalities not captured by OTS components. The approach is supported by several techniques that, together, cover the three stages of the lifecycle discussed above. The techniques also augment one another in their ability to detect larger sets (and more detailed types) of component mismatches.

In the following sections the two stages of our component-based development approach are discussed. Each section describes what information is available at that stage, what the goals are, and what approaches have to be followed to reach those goals. It will be illustrated how mismatches can be identified during each stage, in the context of an example. The tool support that is available for each stage will be discussed.

3 Example system

The proposed example is a hospital system that extends a legacy application and incorporates some COTS capabilities. The COTS component *Medication DB* is a comprehensive database about existing medications and their side effects. It is used in the hospital system to identify automatically whether a newly introduced medication for a patient conflicts with previous or current medications.

The current legacy system covers parts of the proposed capabilities (see shaded area in Fig. 2). However, the legacy system needs to be extended to support automated medication analysis. In particular, elements of the legacy system dealing with patients and treatments require a more precise definition of their activities. The legacy system describes treatment information in plain English. For an automated analysis of treatments and side effects, as well as automated report generation of treatment options and results, the information-capture part of the legacy system needs to be enhanced to become more systematic.

4 Evaluating architectural options

Early on in any software engineering effort, several architectural options may be considered. These are based on the set of given requirements and must be evaluated, supporting a reduction in the number of options, and then further

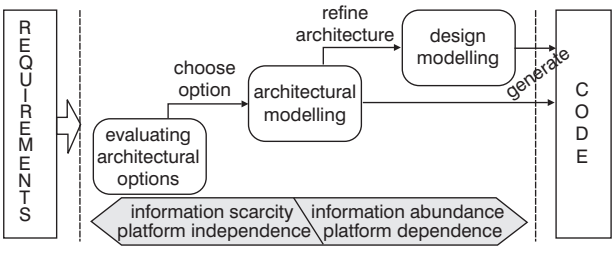


Fig. 1 *Modelling framework*

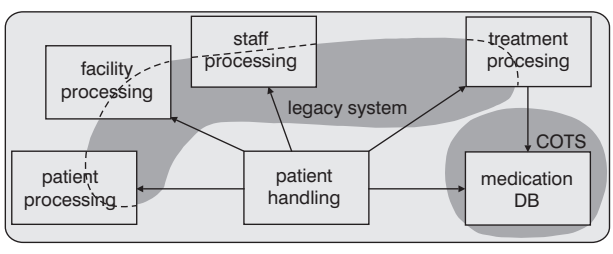


Fig. 2 *Overview of proposed Hospital System*

refined [7]. These architectural options are high-level descriptions of components and their expected interactions. Among others, they must be evaluated to determine architectural mismatches they may entail. The goals described in this section are to model components with their architectural features and to minimise mismatches between them.

4.1 Finding conceptual features to describe components

The existence of architectural mismatches among various parts of a system may seriously hinder a component-based software-engineering effort [2]. Architectural mismatches are caused by inconsistencies between two or more constraints of different architectural parts being composed. Architectural mismatches may vary considerably in terms of the kind of impact they have. Some may easily be avoided by the simple use of a wrapper, whereas others may be extremely expensive to handle. This illustrates the importance of early risk assessment when reuse possibilities are being considered.

To perform early risk assessment during component composition, the Architect's Automated Assistant approach and tool (AAA) are used [8]. AAA is a mismatch-detection approach that supports rapid evaluations of components with respect to the potential incompatibilities among them. AAA takes as input a high-level description of the (sub)systems to be composed and the kinds of interactions expected among the components in the subsystem. Based on those descriptions it is possible to infer mismatches among the components. Subsystem descriptions can be given in terms of architectural styles or specific architectural features. Some of the component features supported by AAA are:

(*a*) Backtracking: does the component support backtracking while trying to solve a problem?
(*b*) Concurrency: is the component multithreaded or not?
(*c*) Control unit: is there a special subcomponent within the component responsible for arbitrating which components are to execute at any given point in time? If so, is this a central control unit or are there distributed ones?
(*d*) Distribution: is the component mapped to more than one hardware node or not?
(*e*) Encapsulation: does the component include some private data and/or control elements?
(*f*) Layering: does the component use some layering with respect to control or data connectors?
(*g*) Pre-emption: is pre-emption required in the subsystem, i.e. are there tasks that must be interrupted and suspended to start or continue running another task?
(*h*) Reconfiguration: does the component support online reconfiguration or does it require offline intervention?
(*j*) Re-entrance: reentrant code can have multiple simultaneous, interleaved, or nested invocations which will not interfere with each other. Does the component contain reentrant parts?
(*k*) Response times: does the component have some predictable, bounded or unbounded response-time requirements? Is it cyclic (i.e. does it contain a cycle that will run indefinitely)?
(*l*) Supported data transfers: how are data transferred within the component? Is it via shared variables, explicit data connectors (such as pipes), and/or shared repositories?

Other features supported by AAA and not included here are component priorities, dynamism and triggering capabilities. For a detailed description of the complete feature set as well as their relevance for architectural mismatches refer to [8].

The various component features are gathered by interviewing the people that were or will be involved in their development and maintenance. Re-engineering existing parts may also be of help, as long as not too much effort is invested. The same approach is also used for COTS packages, since vendors may be willing to give out at least general information on these characteristics—they do describe the system and often help describe APIs, without giving away secrets that could reduce the vendor's competitive advantage. Features whose values are unknown are simply reported as such.

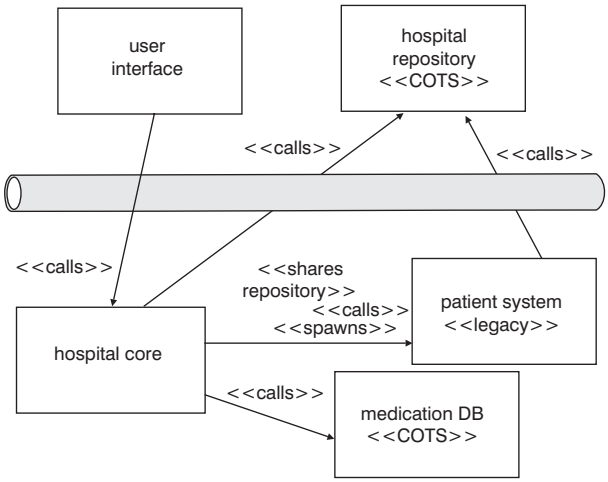
An architectural style defines a family of systems based on a common structural organisation [5]. It constrains both the design elements and the formal relationships among the design elements [6]. The set of constraints imposed on a style determines the set of features that are fixed for the style, as well as those that may vary from system to system within the given style. Consequently, the use of architectural styles simplifies the description task by already inheriting the values for the features relevant at the style level. Those features not fixed for the style are initially set to unknown, and may potentially be refined as system-specific knowledge becomes available. AAA contains a set of predefined architectural style descriptions, but allows descriptions of other styles based on their feature set, covering both fixed and unconstrained values. The set of software connectors (interactions) supported by AAA are calls, spawns (or forks), data connectors (e.g. pipes), shared data variables, triggers and shared resources (e.g. a hardware node).

AAA handles partial descriptions of software components due to the fact that it can be used very early in the software development process, when information is scarce and not yet fully defined. AAA deals with incomplete information by making pessimistic assumptions while checking for architectural mismatches. Since component-mismatch analysis is carried out based on assumptions, the results obtained are not precise. However, these results are highly valuable for risk assessment, and provide some insight for later refinement of the previously unconstrained features.

4.2 Detecting component mismatches with AAA

In the context of the hospital system, AAA was given the information depicted in Fig. 3. This is a very high-level architectural model describing the proposed interactions of the existing legacy system (*Patient System*). COTS product (*Medication DB*), and the subsystems under development (*Hospital Core*, *User Interface*). The component *Hospital Core* is yet to be developed but its set of desirable component features is known. However, it is desired not to overconstrain the options at this point by already committing to an architectural style. Features for the *Hospital Core* have been set in AAA according to the description shown in Fig. 3. Note that some features, such as control unit and pre-emption, have been set as 'unknown'. The other components are defined similarly. Note that for brevity only the features of the most significant component are listed, excluding those of others such as User Interface.

AAA generates a list of potential mismatches by checking the descriptions of the components and the connectors used to compose them against a set of predefined mismatch rules. Mismatch rules are specified in terms of preconditions and potentially resulting problems that may result from them. A mismatch example would be: 'This component is sharing data with some component(s) that may later



Subsystem	Hospital core	Patient system	Medication DB
Predefined style	undefined	database-centric	database-centric
Background information	in-house	legacy	COTS
Backtracking	no	yes	yes
Concurrency	yes	yes	yes
Control unit	unknown	central	central
Distribution	single-mode	single-mode	single-mode
Encapsulation	yes	yes	unknown
Data/control layer	yes/yes	yes/unknown	unknown
Pre-emption	unknown	unknown	unknown
Reconfiguration	online	offline	offline
Re-entrance	no	yes	yes
Response time	unbounded	bounded	bounded
Shared-variable data connector	unknown yes	no unknown	no unknown

Fig. 3 *Hospital System architecture and component features (excerpt)*

backtrack', with the associated precondition that at least one of the subsystems sharing a given data set has backtracking, and the specific problem of concern being the fact that, while backtracking these may have undesired side-effects on the overall system state. The presence of such a precondition would be checked by iterating through every pair of components connected via shared data, and verifying whether at least one of the components has the backtracking feature.

As a result of the AAA analysis, an indication is obtained of potential risks which might be encountered at a later state when combining those components. In the present example, AAA is able to detect roughly 20 mismatches. Nonetheless, AAA comes up short in actually specifying the details of how the component interaction is enabled and, specifically, how the *Hospital Core* component is actually realised. The following lists an excerpt of mismatches detected by AAA while processing the Hospital System information on components and connectors as depicted in Fig. 3 (these mismatches are discussed in more detail below):

(i) A remote connector is extended into or out of a nondistributed subsystem. The originally nondistributed subsystem(s) cannot handle delays and/or errors occurring due to some distributed communication event (violating subsystems: *Hospital Core* and *Patient System*, *Hospital Core* and *Medication DB*);
(ii) Sharing data with some component(s) that may later backtrack. Backtracking may cause undesired side effects on the overall composed system state (violating subsystems: *Hospital Core* and *Patient System*);
(iii) Only part of the resulting system automatically reconfigures upon failure (violating subsystems: *Reconfiguration on-line*: *Hospital Core*, *Reconfiguration off-line*: *User Interface*, *Patient System*);
(iv) Call to a component that performs on-the-fly garbage collection; undesirable side effects on the overall predictable response times (violating subsystems: *Hospital Core* and *Patient System*)

The AAA approach is tool supported. The results obtained while analysing the information relevant to the Hospital System can be seen in Fig. 4. One of the great advantages of using the AAA tool is the minimal effort required for obtaining results, as well as experimenting on varying the features and/or connectors used.

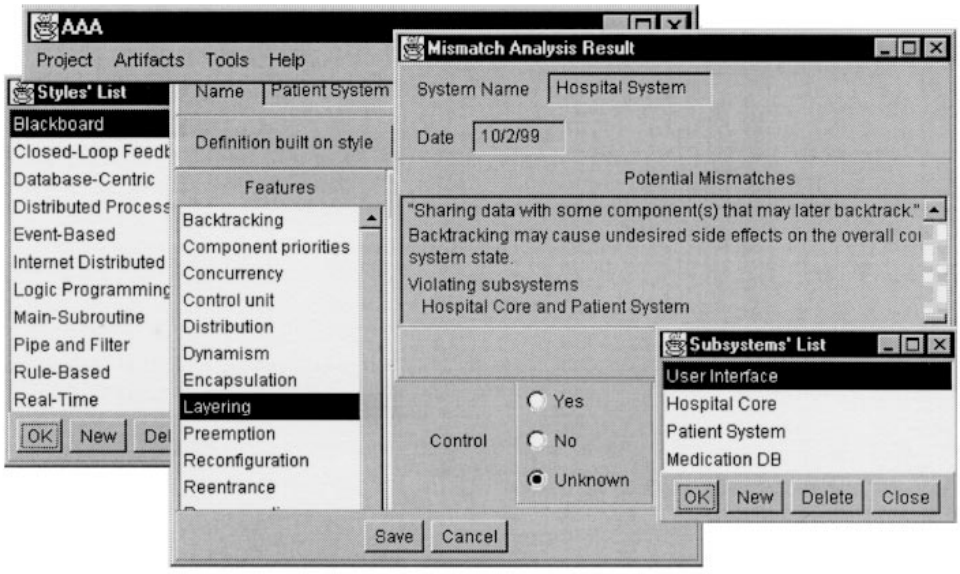


Fig. 4 *AAA screen snapshots*

4.3 Minimising component mismatches through architectural trade-off analysis

The results obtained by analysing architectural descriptions with AAA must be dealt with by domain and application experts to determine the mismatches that are a real threat and decide how to deal with them. Some of the approaches that may be used for mismatch-risk minimisation include but are not limited to, using a different (set of) component(s) with differing characteristics to support same required functionality, changing the means of components interaction (connectors), varying the features of the components to be built, and introducing wrappers or instrumented connectors [9]. All but the last of these approaches is of an exploratory character, and hence easily supported by AAA, requiring only a very limited set of actions. The main result of this risk-minimisation activity is a high-level architectural description reflecting the component configuration that has the greatest chances of success.

5 Architecture modelling to integrate components

Section 4 presented an approach (AAA) whereby a preliminary, coarse-grain model of a system based on its major components is used to identify one or more viable architectural options for the system and highlight any unresolved risks associated with each option. These issues are dealt with early in a system's lifecycle, thus potentially minimising the costs of selecting suboptimal options or failing to mitigate the major risks. However, the AAA approach by itself does not provide the mechanisms needed to perform in-depth comparisons of identified options or to resolve the risks. Instead, AAA must be accompanied by additional approaches that will, for each architectural option:

(i) Find an architectural style that supports the system's current components, their conceptual features, and their interactions (recall the discussion in Section 4). Certain styles may be better suited than others to handle particular types of components, features, and interactions. Some styles may even resolve certain risks carried over from the evaluation of architectural options. Additionally, an architect must ensure that styles do not clash with system requirements and exacerbate other known risks. Finally, it may be difficult to assess the impact of a style and the architectural model on a given issue, in which case any decision must be deferred pending additional modelling and analysis (e.g. as proposed in Section 6);

(ii) Assess the ability of the chosen style to help minimise the mismatches between those components, features, and interactions;

(iii) Carefully model the critical aspects of the system in the chosen style, e.g. by using an architecture description language (ADL); [10] and

(iv) Analyse the architectural model for model mismatches. Note that these mismatches will typically differ from those identified by AAA since the architectural model will be more detailed and complete.

5.1 Finding an architectural style to fit components

Architectural options, as found through AAA in Section 4, come up short in specifying how components have to be built (or wrapped in the case of OTS) and how they must interact. To refine a given architectural option into an eventual implementation, the components, connectors and their interconnections (guided by the rules of a given style) have to be defined in a more rigorous manner. All software systems adhere to some architectural style. Common styles include layered, client-server, pipe and filter, event-based and main–subroutine [5]. An architectural style defines the relationships of components and connectors that are shared across systems. Choosing an (in)appropriate architectural style(s) has considerable impact on the feasibility of a project. The method described here provides help in this regard. For clarity, the method is illustrated in the context of the Hospital System example.

Table 1 couples the notion of component features (Section 4) and styles, showing their relationship (a more complete version of the table is given in [8]). The table can be used as a selection mechanism to infer candidate styles out of known components, connectors and their configurations. For example, although the *Hospital Core* component in the Hospital System currently has no style explicitly associated with it, its conceptual features exclude many

Table 1: Mapping component features to architectural styles

Features	Distributed process	Event-based	Main–subroutine	Pipe and filter	Database centric	Real time	C2SADEL
Backtracking	no	no	no	no	yes	no	no
Concurrency	multi-threaded	multi-threaded	single-threaded	multi-threaded	multi-threaded	multi-threaded	multi-threaded
Control unit	none		none	none	central	central	
Distribution	multiple nodes		single nodes				
Encapsulation							
Data/control layer							
Pre-emption			no	no		yes	
Reconfiguration							
Re-entrance			no	no	yes	no	
Response time	unbounded	unbounded	unbounded	unbounded	bounded	unbounded	unbounded
Shared variable			yes		no		
Data connector	yes			yes			yes

potential styles. From Fig. 3, we know that *Hospital Core* must support concurrency. We learn from Table 1 that all of the styles considered, except for main-subroutine, support concurrency. The main-subroutine style should therefore not be used as an architectural style in this case. This exclusion method can be applied in the context of all other features.

Further analysis establishes that, of the four major components in the present case study, two follow the database-centric style (*Hospital Repository* and *Medication DB*), one follows the event-based style (*User Interface*), and the last one is unspecified (*Hospital Core*). In choosing an architectural style for the overall system, an architectural style that fits all components and their proposed forms of interactions must also be chosen. (Note that it is indeed possible to select multiple architectural styles for different parts of a given application. The present approach allows this, in the same manner as discussed above, assuming that the styles are inherently compatible. (Architectural style incompatibilities [11] are beyond the scope of this work). In this case all components are currently defined as nondistributed, i.e. each individual component resides in a single address space. By itself, this property does not place any constraints on the configuration of the entire system, meaning that all styles remain in the candidate pool. Because of the need for concurrent components, the overall system must also be concurrent (excluding the main-subroutine style again). Since some components require re-entrance and others do not, it is necessary to use a style that supports both (excluding pipe-and-filter and real-time). Finally, since some components require a central control unit, all styles not supporting such a unit may be excluded as well (e.g. distributed processes).

With the remaining three styles (event-based, database-centric, and C2) we encounter a conflict. On the one hand, system response time needs to be unbounded, due to the *Hospital Core* component. On the other hand, backtracking capabilities are needed to support the *Patient System* component. The former favours the event-based or C2 styles, whereas the latter favours the database-centric style. To resolve this conflict, it is necessary either to investigate additional styles or to use additional information:

(i) To choose between event-based and C2, may choose the one with the greatest feature coverage: in this case, the example favours the C2 style since it supports explicit data connectors as required by the *Hospital Core* component (C2 is an event-based style with exactly that extension).
(ii) To choose between C2 and database-centric, we may choose the one with the highest potential for resolving known architectural mismatches (or risks) as they were identified in Section 4. To address the 'remote-connector mismatch', C2-style connectors limit the effects of individual components on overall system structure and evolution. C2 components exhibit very low coupling: a component is only aware of a single connector above and/or a single connector below it. That means that components can be added, removed, replaced or reconnected in a C2-style architecture, even at runtime, without their neighbouring components ever needing to know about those changes [12]. One the other hand, the database-centric style is not well suited to this case. Conversely, the database-centric style is well suited to address the 'sharing-data-and-component-backtracking mismatch', for which C2 is not as well suited. The 'automatic reconfiguration mismatch' is addressable by C2: the message-based interaction of C2 components via connectors presents an ideal foundation for system adaptability, both offline and at system runtime [12]. It is important to note that the fourth mismatch discovered by the AAA tool (response time due to on-the-fly garbage collection) is well supported neither by C2 nor by the database-centric style.

As will often be the case, none of the three styles is the perfect fit to this problem. In such a case, the selection of a style is based on an evaluation of which style can most easily be amended to address its deficiencies. If the selected architectural options simply cannot be implemented by any of the candidate styles, another option should be chosen. If no architectural option can be satisfied, then this may indicate a need for a change in the requirements. Specifically, for the hospital application C2 can be amended to deal with the backtracking issue. The topological rules and message-based communication of C2 support easy update of components. In particular, if the *Patient System* component from Fig. 3 backtracks, it can send a message to that effect to the *Hospital Repository*; in turn, the *Repository* will issue a notification message, which will be relayed by the appropriate connector(s) to the *Hospital Core* component. Note that *Hospital Repository* is a COTS component and, as such, is unlikely to adhere to C2's interaction rules. However, C2 provides a set of simple mechanisms and resulting tools for incorporating heterogeneous OTS components [13]. In particular, a lightweight wrapper can be constructed for *Hospital Repository*, as discussed below.

Section 5.2 will therefore model in more detail the chosen architectural option from Section 4 using the C2 style and its accompanying suite of modelling, analysis and implementation technologies. The goal of the further analysis is to investigate in more detail whether the above shortcoming can be mended, and whether there are other currently unknown risks. The potential role of other architectural technologies as candidates for effecting additional architectural styles will also be discussed briefly (e.g. layered, event-based, database-centric).

5.2 Modelling the Hospital System in C2SADEL

In this section the approach is discussed to leveraging an architectural style (C2) and using an ADL (C2SADEL) to further refine (or define) the component and connector features that were not elucidated as part of the process of evaluating architectural options (Section 4). An explicit architectural model allows mismatches to be detected at the level of component interfaces, behaviours and interaction protocols. Explicit models are also amenable to analysis tools that support rapid evaluations of architectural descriptions and highlight key problem areas in a given system's model.

The C2-style breakdown of the hospital-system architecture is shown in Fig. 5. The shaded portion in the centre of the architecture represents a refinement of the *Hospital Core* component from Fig. 3. C2 allows *Hospital Core* to be modelled as a single component or, hierarchically, to contain an internal architecture of its own.

Recall from Section 4 that the re-engineered hospital system will include components that supplement the architecture functionality already provided by one or more of the legacy components. Thus, for example, the *Patients* component in *Hospital Core* will supplement the legacy *Patient System* component. The *Patient Handling* component will issue patient handling messages. In turn, those messages will be routed by the appropriate connector(s) to

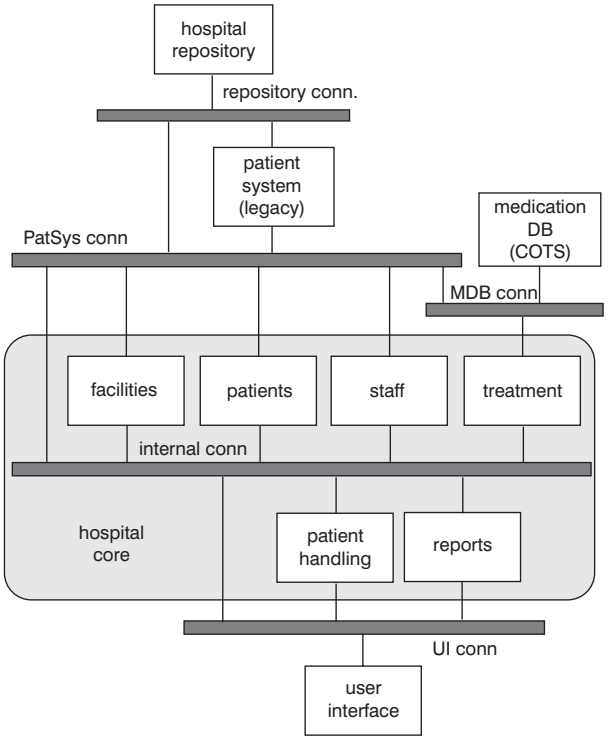


Fig. 5 *Hospital System architecture modelled in C2*

one of the two patient processing components (e.g. based on their interfaces). Occasionally, *Patient Handling* will issue patient treatment related messages to the *Treatment* component, which will, in turn, consult the COTS *Medication DB* component to determine, for example, any conflicts in prescribed medications. Similarly, the *Facilities* and *Staff* components will handle their respective responsibilities, while the *Reports* component will invoke any of the above components which it needs to produce a report the user demands via the *User Interface* component. The *Hospital Core* and *Patient System* components may store some information internally, or they may occasionally request information from a central, COTS *Hospital Repository*.

Fig. 5 gives us a good understanding of the 'big picture', i.e. the Hospital System's overall architectural breakdown. However, to ensure that the architecture exhibits the desired properties and that the different components can interact in envisioned ways, a more detailed and formal specification of the architecture is needed. For that purpose, C2SADEL, a language for describing and evolving C2-style architectures [14], is employed.

A C2SADEL architecture is specified in three parts: component types, connector types and topology. The topology, in turn, defines component and connector instances for a given system and their interconnections. A partial description of the architecture shown in Fig. 5 is

```
HospitalSystem is {
  component_types {
      component TreatmentComp is extern {TreatmentComp.c2;}
      component MedicationDBComp is virtual {}
      ...
  }
  connector_types {
      connector FiltConn is {filter msg_filter;}
      connector RegConn is {filter no_filter;}
  }
  architectural_topology {
      component_instances {
        Treat : TreatmentComp;
        MedDB : MedicationDBComp;
        ...
      }
      connector_instances {
        TreatConn : FiltConn;
        PatConn : FiltConn;
        ...
      }
      connections {
          connector TreatConn {
              top MedDB, PatConn;
              bottom Treat;
          }
          connector PatConn {
              top PatSys, RepConn;
              bottom TreatConn, InternConn, Fac, Pat, Staff;
          }
          ...
}}}
```

Fig. 6 *Partial specification of the Hospital System*

given in Fig. 6. The *Treatment* component type is specified externally, i.e. in a different file (Treatment.c2). The *Medication DB* component type is specified as a virtual type; it can be used in the definition and analysis of the topology, but it does not have a specification and does not affect analysis of component conformance. The concept of virtual types is useful for components for which implementations are known to already exist, but which are not specified in C2SADEL (e.g. COTS). Note that *Medication DB* is modelled as a virtual component in this case only to demonstrate this feature of C2SADEL. A typical approach to modelling COTS components has been to extract from their available documentation, e.g. AP1, the desired services and model them in C2SADEL, thus permitting more meaningful component conformance analysis).

Individual components, such as *Treatment* shown in Fig. 7, are specified to contain a set of component state variables, a component invariant and a set of services. Each service consists of an interface and an operation; operations are modelled via pre- and post-conditions in first-order logic. A service can be provided by the modelled component or required of some other component in an architecture. Finally, interfaces and operations are separated, so that it is possible for an operation to export multiple interfaces.

5.3 Minimising component mismatches through architectural analysis

The specification of component invariants and services in C2SADEL allows components that are composed in an architecture to be analysed for conformance. For example, the required *medConf* service of the *Treatment* component must be matched by a provided service of one of the components to which *Treatment* is attached in the Hospital System architecture. More specifically, owing to C2's style rules, this service must be matched by one of the components above *Treatment* in the architecture, namely *Medication DB*, *Patient System*, or *Hospital Repository.* A required service P matches a provided service Q if the following conditions hold [15]:

(*a*) P and Q have identical interface and interface parameter names;
(*b*) Either both P and Q have a return value or neither does;
(*c*) The types of P's interface parameters are subtypes of the types of Q's interface parameters;
(*d*) The type of P's return value is a supertype of the type of Q's return value;
(*e*) P's precondition implies Q's precondition; and
(*f*) Q's postcondition implies P's postcondition.

At a minimum, these rules ensure that, if, for example, procedure calls are used to enable the interactions of the implementations of the components containing P and Q, those interactions will be allowed by the underlying programming language. Additionally, the analysis of preconditions and postconditions permits establishment of the behavioural conformance of the two components. Since manual conformance checking would be a time-consuming and error-prone task, the SAAGE environment is used to perform automatic model checking of an architecture. Further, SAAGE supports automated generation of the architecture's prototype implementation [14]. The prototype allows observation and rapid evaluation of critical dynamic system properties.

Fig. 8 shows a screenshot of SAAGE and the results of its analysis of the hospital system specified in C2SADEL. Fig. 8 also shows the resulting UML model automatically generated in Rational Rose. Note that SAAGE reported three mismatches in the 'Architectural Type Mismatch' pane. The first mismatch states that the *Treatment* compo-

```
component Treatment is {
      state{
          cur_pats : \set Pat ID;
          pat_meds : \set Med;//implement as PatID− > \set Med;
      }
      invariant {cur_pats\eqgreater 0;}
      interface{
          prov ip' : presoriceMed (p : Pat ID; m : Med);
          req ir' : medConf (new_m : Med; old_m :\set Med); Bool;
      }
      operations{
          prov op' : {
              let pid : Pat ID; m : Med;
              pre (m\not_in pat_meds);
              post (pid\in - cur_cats)\and (m\in - pat_meds);
          }
          req or' : {
              let m : Med; ms :\set Med;
              post\result = (m\in ms);
          }
      }
      map {
          ip'− > op' : (p− > pid, m− > m);
          ir'− > or' : (new_m− > m, old_m− > ms);
}}
```

Fig. 7 *Partial specification of the Treatment component in C2SADEL*

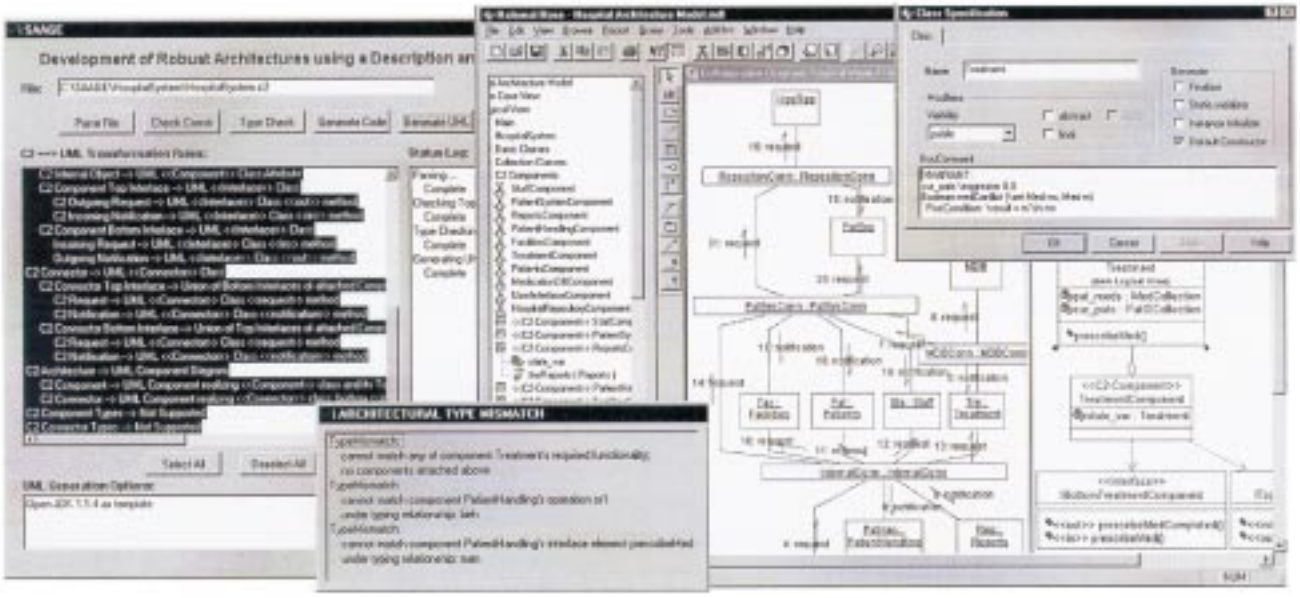


Fig. 8 *Screenshot of SAAGE*
The automatically generated UML model in Rational Rose is on the right

nent's required service *medConf* cannot be matched because no components are attached above it. In this case, this simply means that all components attached above *Treatment* in the hospital-system architecture, such as *Medication DB*, are virtual and SAAGE was unable to use them to establish component conformance.

The second mismatch is more interesting: it indicates that SAAGE was able successfully to match the interface of *Patient Handling* component's required service *prescribeMed* to *Treatment*'s corresponding provided service. However, the required behaviour does not match the provided behaviour. The preconditions and postconditions of the two operations are specified in Table 2. To satisfy behavioural conformance rules, the following two relationships must hold:

(*a*) req precondition implies prov precondition, i.e.

```
(p \in pats) => (m \not _in pat_meds)
```

(b) prov postcondition implies req postcondition, i.e.

```
(p \in ~cur_pats) \and (m \in ~pat_meds)
    => (m \in ~meds)
```

The first relationship clearly cannot be established: the left-hand side (LHS) of the implication deals with patients and the right-hand side (RHS) with medications. The second relationship is of the form $A \text{ and } B => A$, which is always true. To permit the match of the postconditions, SAAGE had to unify the RHS variable *meds*, which is of the generic type STATE_VARIABLE [14, 15], with the LHS variable *pat_meds*, which is defined as part of the *Treatment* component's state (recall Fig. 7).

Table 2: Preconditions and postconditions of two components

Patient handling *req* service	*Treatment* *prov* service
pre (p \in pats)	**pre** (m \not _in pat _meds)
post (m \in - meds)	**post** (p \in ~cur_pats) \and (m \in ~pat_meds)

Since the necessary relationship between the preconditions could not be established, SAAGE also attempts to match *Patient Handling* component's required service *prescribeMed* to the provided services of the other components attached above it in the Hospital System architecture. None of the other components provide a corresponding service, resulting in the third architectural type mismatch shown in Fig. 8. Finally, note that the architect's decision to generate the UML model in spite of the mismatches indicates the architect's belief that those mismatches are either not critical or would be remedied during the system's design phase.

5.4 Applicability to other styles and ADLs

The above discussion demonstrated the coupling of AAA and C2, two techniques independently developed by the authors. However the overall approach to exploiting architectures to aid component-mismatch detection is in no way predicated on the use of C2, quite the contrary. The same general process can be used when other styles and ADLs are coupled to aid in software modelling and analysis [10]. For example, the event-based style may be used effectively in conjunction with the Rapide ADL [16]; similarly, the layered style may be used with the GenVoca ADL [17]; and the distributed style may be coupled with Darwin [18]. Finally, certain ADLs, such as Aesop [19] and Wright [20], allow an architectural style to be modelled formally using the same environment and language that is, in turn, used to model application architectures within that style. In each such situation, the details of the models and analyses depend on the characteristics of the style and features and tool support of the chosen ADL.

6 Design modelling and refinement into code

The process of choosing an architectural option, selecting its architectural style and modelling the resulting architecture in detail yields an architectural model. At this stage, we have available a fairly well validated architectural model of components, connectors, and their combined configurations of which it is known that they exhibit properties that satisfy the given requirements. What

remains to be done is to refine that architectural model into a software system. That refinement involves (i) the implementation of components that are not off-the-shelf; (ii) the implementation of glue code (e.g. wrappers) around components that are pre-existing (e.g. COTS or legacy); and (iii) the implementation of connectors that enable components to interact (unless they are COTS themselves as in CORBA [21]). It is out of the scope of this paper to investigate these issues in detail: however, in the following a brief overview and references on how to realise such a transition are provided in the context of the present framework.

The traditional way of implementing software models of any kind is the manual process of reading and interpreting available documentation, followed by programming it. With the availability of architectural models, the task of refinement is simplified in that major components are specified: those components provide the major partitions of a system, and programming those partitions is less complex in that they can be implemented separately. Nevertheless, the downside of a manual refinement approach is that there are no guarantees that the model descriptions are implemented correctly. Refinement from architectures into code must, however, ensure consistency [22, 23]. Continuing on the path outlined in Fig. 1, two major refinement options are available:

(*a*) *Refining the architecture into code*: The synthesis process for refining architectures varies depending on the different architectural styles available (e.g. event-based, main–subroutine, C2, etc.). Depending on the 'richness' of the chosen architectural styles, high degrees of code generation are possible.

(*b*) *Synthesising a design from an architecture*: Using generic-purpose notations such as the Unified Modelling Language [3] for refinement has four major advantages:

(i) Refining into design models instead of code is less of a gap to bridge and thus easier;
(ii) Design languages are generally more easibly understandable which makes subsequent non-automatable coding simpler;

Table 3: Impact of component-based development approach

	Modelling architectural options	Modelling architecture	Modelling design and coding
Modelling constructs	• Components • Interactions • Conceptual features	• Architectural configuration • Architectural styles • Roles and responsibilities	• Classes • Methods • Variables
Questions	• What are the components required to do? • Can we exclude potentially unfeasible architectural options? • What are the component characteristics?	• How can architectural options be modelled and analysed without implementing it? • Does the architectural solution satisfy requirements?	• How can architectural models be implemented? • How are configurations, styles, roles, and responsibilities transformed into programming constructs?
Time frame	• Can frequently be done in hours for complex systems	• Can frequently be done in days or weeks for complex systems	• Frequently takes many months or even years for complex systems
Given	• Requirements • Domain knowledge • COTS and legacy components	• One or more viable architectural options based on components • Known risks from previous stage that were not resolved	• One or more viable architectural models that satisfy requirements • Known risks from previous stages that were not resolved
Goals	• Find components that could satisfy requirements (with or without overlapping functionalities) • Minimise mismatches between those components	• Find an architectural style that supports components, their conceptual features, and their interactions • Minimise the mismatches that could occur between those components, features and interaction with the respective chosen style (note: some styles may be more suitable than others for handling known mismatches)	• Consistently refine architectural model into code • Use design languages as intermediate models if necessary (e.g. to enable design pattern reuse)
Approaches	• Vary components to minimise potential risks (e.g. replace event-based component with blackboard-style component) • Vary component interactions (connectors) to minimise potential risks • Toggle variable component characteristics (e.g. of to-be-built subsystems)	• Evaluate different architectural styles to investigate their potential impact • Refine (define) undeclared component and connector features to tailor integration between styles, components, and connectors	• Refine known style, component and connectors information into code • Refine known style, component and connectors information into design information
Tools	• Trade-off analysis tool that supports rapid evaluation of components incompatibilities among them	• Selection mechanism that infers styles out of known component and connector characteristics • Trade-off-analysis tool that supports rapid evaluations between styles, components and connectors	• Defined refinement mechanism for all types of styles and ADLs

(iii) Design specifications provide additional, more program-language-specific constructs that might increase the degree of (automated) code generation; and finally;
(iv) Design modelling can make use of design patterns, such as wrapper technologies, which in turn increases the degree of reuse.

Based on the present example in C2SADEL, the SAAGE environment is capable of supporting both refinement activities above: it can generate a partial implementation of the architecture [14] and it can generate a UML design [24, 25]. An in depth study has also been conducted of the feasibility of mapping other ADLs to UML [25] covering Rapide [16] and Wright. Furthermore, related works of other researchers have done similar things for other types of ADLs and architectural styles (e.g. domain-specific layered style [26], real-time styles, object-oriented and main-subroutine styles [27], proof-carrying architectural styles [28]). Those works demonstrate that other types of architectural style can be refined into code, either directly or via intermediate design models.

7 Discussion

The proposed approach 'interfaces' with requirements engineering on one side and coding on the other. Requirements modelling needs feedback to evaluate the feasibility of suggested options. If it takes weeks or months to give that feedback, system requirements modelling may never stabilise or artificial cut off dates must be set with negative side effects for flexibility (e.g. waterfall model). If, on the other hand, the feedback can be obtained in days, requirements modelling intertwined with architectural modelling becomes feasible. The approach described provides feedback about unfeasible options in mere hours; in turn, promising options can be selected and refined into architectures for more thorough risk analysis in days or weeks.

It is believed to be very beneficial to combine model-based and component-based development, since model descriptions provide the context for how to combine components whereas components address more specific system needs. Furthermore, components are helpful in partitioning architecture and design models: since components represent independent subsystems, any resulting models tend to become more independent themselves. The proposed two-tiered development approach combines modelling constructs and processes that were previously separated.

The power of compositional-modelling techniques provided through AAA and SAAGE lies in the fact that major development concerns can be modelled, analysed, and simulated early on, resulting in a very cost-effective way of dealing with development risks. However, those models add little value to the final software product if the product-related information stored in them cannot be transitioned into the final product. For instance, the C2 architectural style was chosen because the AAA analysis suggested that there are potential risks that C2 could remedy (e.g. remote connector). To ensure that the final product still resolves those risks, it is necessary also to ensure that the final product inherited all relevant characteristics from its C2 architecture. In other words, it is necessary to ensure consistent refinement.

Since the example chosen for this paper is not representative for all software products, a discussion has been presented of how other types of situations could be addressed via the present approach. To summarise the 'big picture' of the proposed development approach, consider Table 3. Table 3 shows the three major elements of the proposed development approach (columns) and their properties. It briefly summarises the constructs available at those respective stages, the questions we would like to answer, the information initially available, the goals that need to be achieved, the approaches one has available, and the tools that support it. Table 3 has been extrapolated from the discussion provided in the previous sections. It represents a quick reference for assessing the issues, progress and results in a component-based development effort.

8 Conclusions

This paper discusses two modelling and analysis approaches dealing with the 'upstream' activities in the software engineering life cycle: architectural options, architectures and designs. An example is used to illustrate their respective advantages in identifying and resolving mismatches. It is shown how (mismatch) feedback from one could impact others and drive development decisions.

The technique for evaluating architectural options may yield useful feedback (e.g. risk assessments) within hours and is highly useful for eliminating unfeasible options early on (e.g. even during requirements negotiations), however, this is at the expense of precision. The technique for architectural modelling is complementary since it is capable of performing more detailed and precise component-mismatch analyses. This is at the expense of the effort required. Both are needed to develop component-based software rapidly, and thus, both have to be connected. To that extent, mechanisms are defined for information sharing among the chosen techniques. Additionally, techniques for refining architectures systematically (with their incorporated component definitions) into more detailed architectures, designs and code were found.

The authors believe that mismatch detection should happen as early on as possible. One software engineering truism is that the longer a mismatch remains undetected the more harm it might cause. The authors also believe that automated analysis techniques are invaluable in getting easy and fast initial feedback on potential problems and challenges, regardless of the current development stage. It has been shown that the techniques are useful as early as the initial system analysis stage when the lack of information drives development. It has also been shown that the techniques can be used as late as coding and maintenance when information abundance constitutes the key complexity.

Working continues on tool support in three primary directions:

(i) more extended coverage of other development models such as architectural description languages and design models; (ii) more complete data, control and process integration between the tools; and (iii) more comprehensive coverage of the software lifecycle (e.g. product families, requirements engineering, reuse).

9 Acknowledgments

The authors wish to thank the anonymous reviewers for their feedback. This effort was sponsored by the Defense Advanced Research Projects Agency, and Air Force Research Laboratory, Air Force Material Command, USAF, under agreements F30602-94-C-0195. F30602-99-

1-0524, F30602-99-C-0174 and F30602-00-2-0615. The effort was also sponsored in part by the National Science Foundation under grant CCR-9985441. The US Government is authorised to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Air Force Research Laboratory or the US Government.

10 References

1 BOEHM, B., and SCHERLIS, W.L.: 'Megaprogramming'. Proceedings of the *DARPA Software technology conference*, April 1992 (available via USC Center for Software EngineeringLos Angeles, CA 90089-0781)
2 GARLAN, D., ALLEN, R., and OCKERBLOOM, J.: 'Architectural mismatch or why it's hard to build systems out of existing parts', *IEEE Softw.*, 1995, pp. 17–26
3 RUMBAUGH, J., JACOBSON, I., and BOOCH, G.: 'The unified modeling language reference manual' (Addison Wesley, 1999)
4 EGYED, A., and GACEK, C.: 'Automatically detecting mismatches during component-based and model-based development'. Proceedings of the 14th IEEE international conference on *Automated software engineering*, Oct. 1999, pp. 191–198
5 SHAW, M., and GARLAN, D.: 'Software architecture: perspectives on an emerging discipline' (Prentice Hall, 1996)
6 PERRY, D.E., and WOLF, A.L.: 'Foundations for the study of software architectures', *ACM SIGSOFT Softw. Eng. Notes*, 1992, pp. 40–52
7 BOEHM, B.W.: 'Anchoring the software process', *IEEE Softw.*, 1996, 73–82
8 GACEK, C.: 'Detecting architectural mismatches during system composition'. PhD dissertation, 1998, Center for Software Engineering, University of Southern California, Los Angeles, CA, USA 90089-0781
9 BALZER, R.: 'An architectural infrastructure for product families'. Proceedings of the 2nd international workshop on *Development and evolution of software architectures for product families*, Lecture Notes in Computer Science 1429, 1998, pp. 158–160
10 MEDVIDOVIC, N., and TAYLOR, R.N.: 'A classification and comparison framework for software architecture description languages', *IEEE Trans. Softw. Eng.*, 2000, **26**, (1), pp. 70–93
11 SHAW, M., and CLEMENTS, P.: 'A field guide to boxology: preliminary classification of architectural styles for software systems'. Proceedings of the 21st international *Computer software and applications conference (COMPSAC)*, Aug. 1997
12 OREIZY, P., MEDVIDOVIC, N., and TAYLOR, R.: 'Architecture-based runtime software evolution'. Proceedings of the 20th international conference on *Software engineering (ICSE)*, 1998
13 MEDVIDOVIC, N., OREIZY, P., and TAYLOR, R.N.: 'Reuse of off-the-shelf components in C2-style architectures'. Proceedings of the 1997 symposium on *Software reuseability (SRR'97)*, pp. 190–198
14 MEDVIDOVIC, N., ROSENBLUM, D.S., and TAYLOR, R.N.: 'A language and environment for architecture-based software development and evolution'. Proceedings of the 21st international conference on *Software engineering (ICSE'99)*, May1999, pp. 44–53
15 ROBBINS, J.E., MEDVIDOVIC, N., REDMILES, D.F., and ROSENBLUM, D.S.: 'Integrating architecture description languages with a standard design method'. Proceedings of the 20th international conference on *Software engineering (ICSE'98)*, April 1998, Kyoto, Japan, pp. 209–218
16 LUCKHAM, D.C., and VERA, J.: 'An event-based architecture definition language', *IEEE Trans. Softw. Eng.*, 1995, **21**, pp. 717–734
17 BATORY, D., and O'MALLEY, S.: 'The design and implementation of hierarchical software systems with reusable components', *ACM Trans. Softw. Eng. Methodol.*, 1992, **1**, pp. 355–398
18 MAGEE, J., and KRAMER, J.: 'Dynamic structure in software architectures'. Proceedings of the 4th *ACM SIGSOFT symposium on the foundations of software engineering*, San Francisco, USA, Oct. 1996, pp. 3–14
19 GARLAN, D., ALLEN, R., and OCKERBLOOM, J.: 'Exploiting style in architectural design environments'. Proceedings of *SIGSOFT foundations of software engineering*, New Orleans, USA, pp. 175–188
20 ALLEN, R., and GARLAN, D.: 'A formal basis for architectural connection', *ACM Trans. Softw. Eng. Methodol.*, 1997, pp. 213–249
21 VINOSKI, S.: 'CORBA: integrating diverse applications within distributed heterogeneous environments', *IEEE Commun. Mag.*, 1997, pp. 46–53
22 EGYED, A.: 'Heterogeneous view integration and its automation'. PhD dissertation, 2000, University of Southern California, Los Angeles, CA
23 EGYED, A., and MEDVIDOVIC, N.: 'A formal approach to heterogeneous software modeling'. Proceedings of 3rd *Foundational aspects of software engineering (FASE)*, Berlin, Germany, March 2000
24 ABI-ANTOUN, M., and MEDVIDOVIC, N.: 'Enabling the refinement of a software architecture into a design'. Proceedings of the 2nd international conference on the *Unified modeling language (UML)*, Oct. 1999, , Fort Collins, USA
25 MEDVIDOVIC, N., and ROSENBLUM, D.S.: 'Assessing the suitability of a standard design method for modeling software architectures'. Proceedings of the first working *IFIP conference on software architecture (WICSA1)*, Feb. 1999, pp. 161–182
26 HOFMEISTER, C., NORD, R., and SONI, D.: 'Applied software architecture' (Addison Wesley, 2000)
27 SIEGFRIED, S.: 'Understanding object-oriented software engineering' (IEEE Press, 1996)
28 RIEMENSCHNEIDER, R.A.: 'Checking the correctness of architectural transformation steps via proof-carrying architectures'. Proceedings of the first working *IFIP Conference on Software architecture (WICSA1)*, Feb. 1999, pp. 65–81