Multidimensional Catalogs for Systematic Exploration of Component-Based Design Spaces

Claudia López and Hernán Astudillo

Universidad Técnica Federico Santa María, Departamento de Informática
Avenida España 1680, Valparaíso, Chile
clopez @inf.utfsm.cl, hernan @inf.utfsm.cl

Abstract. Most component-based approaches to elaborate software require complete and consistent descriptions of components, but in practical settings components information is incomplete, imprecise and changing, and requirements may be likewise. More realistically deployable are approaches that combine exploration of candidate architectures with their evaluation vis-a-vis requirements, and deal with the fuzzyness of available component information. This article presents an approach to systematic generation, evaluation and re-generation of component assemblies, using potentially incomplete, imprecise, unreliable and changing descriptions of requirements and components. The key ideas are representation of NFRs using architectural policies, systematic reification of policies into mechanisms and components that implement them, multi-dimensional characterizations of these three levels, and catalogs of them. The Azimut framework embodies these ideas and enables traceability of architecture by supporting architecture-level reasoning, and allows architects to engage into systematic exploration of design spaces. A detailed illustrative example illustrates the approach.

1 Introduction

Component-based software development proposes building systems by using pre-existing components, to reduce development time, costs and risks and to improve product quality; achieving these goals requires an adequate selection of components to reuse. Current methods of component evaluation and selection are not geared to support human specialists in the systematic exploration of design spaces because they require complete and consistent descriptions of components behavior, connections and prerequisites. In the real-world software architects have at hand incomplete, imprecise and changing component information, and requirements may be likewise.

This article presents a process and tool to support software architects in the exploration of design spaces by enabling generation, evaluation and regeneration of component assemblies. The Azimut framework deals with the fuzzyness of component information using incomplete "characterizations" of available components and allowing the regeneration of assemblies when better information about components is obtained.

The reminder of this article is structured as follows: Section 2 provides a brief overview of related work; Section 3 introduces the process of generation, evaluation and regeneration of component assemblies, and the concepts of *architectural policies and mechanisms*; Section 4 describes the structure of the *multi-dimensional catalogs*, and illustrates the approach with an example; Section 5 describes the automation of derivation process and its implementation in a prototype; Sections 6 and 7 discuss ongoing work and conclusions.

2 Systematic Processes for Selecting Components

Component-Based Software Development (CBD) [19] suggests reusing existing components to build new systems, attending to benefits like shorter development times, lower costs and higher product quality. Thus, a key ingredient of CBD is component selection.

Some proposed techniques for component evaluation and selection [2–9] identify reuse candidates using criteria such as functionality, non-functional requirements (NFRs) or architectural restrictions that each component and/or the whole system must satisfy. Some of these proposals [5–8] give semi-automated support to the selection process using multi-criteria decision support techniques, such as AHP (Analytic Hierarchy Process) [12] or WSM(Weighted Scoring Method).

Most approaches [2–9] require complete and consistent descriptions of component behavior, connections and prerequisites, but in practice architects have at hand incomplete, imprecise and changing component information. Accepting this fuzzyness and dealing with it is a key step to supporting the actual COTS selection process.

Also, several proposals [2–7] only explore the space of available components without recourse to alternative designs at intermediate abstraction levels. These approaches force architects to deal with a big gap between the component and requirement spaces, and to describe exhaustively the relationships between them. Working with intermediate abstraction levels enables dealing with smaller gaps and searching smaller spaces. CRE [8] and CARE/SA [9] use the NFR Framework [10] to derive more specific requirements or design solutions when considering quality attributes or NFRs; unfortunately, the NFR Framework does not explicitly distinguish requirements more detailed than the design solutions that satisfy them, and the derivation process among them depends on the architect's knowledge of possible refinements, without recourse to a systematic and possible automated derivation support.

3 Exploration, Generation and Evaluation of Component Assemblies

Our larger research goal is supporting iterative exploration of design spaces by human architects, and keeping traceability of the resulting architectural deci-

sions. The Azimut project focuses on enabling architects to generate component assemblies [15] for some given requirements; evaluate and compare these assemblies regarding their requirements satisfaction and some higher-order criteria (e.g. economic, risk); and regenerate assemblies when new or better information is available.

The conceptual vocabulary underlying our approach is description of selection decisions using the concepts of architectural policies and architectural mechanisms.

3.1 Architectural Policies and Mechanisms

Architects may reason about the overall solution properties using architectural policies, and later refine them (perhaps from existing policy catalogs) into artifacts and concepts that serve as inputs to software designers and developers, such as component models, detailed code design, standards, protocols, or even code itself. Thus, architects define policies for specific architectural concerns and identify alternative mechanisms to implement such policies. For example, an availability concern may be addressed by fault-tolerance policies (such as master-slave replication or active replication) and a security concern may be addressed by access control policies (such as identification-, authorization- or authentication-based) [16].

Each *reification* yields ever more concrete artifacts; thus, architectural decisions drive a process of successive reifications of NFRs that end with implementations of mechanisms that do satisfy these NFRs.

To characterize such reifications, we use a vocabulary taken from the distributed systems community [14], duly adapted to the software architecture context:

Architectural Policies: The first reification from NFRs to architectural concepts. Architectural policies can be characterized through specific concern dimensions that allow describing NFRs with more details.

Architectural Mechanisms: The constructs that satisfy architectural policies. Different mechanisms can satisfy the same architectural policy, and the differences between mechanisms is the way in which they provide certain dimensions.

As a brief example (taken from [24]), consider inter-communication among applications. One architectural concern is the communication type, which might have the dimensions of sessions, topology, sender, and integrity v/s timeliness [18]; to this we add synchrony. Then, the requirement *send a private report to subscribers by Internet* might be mapped in some project (in architectural terms) as requiring communication 'asynchronous, with sessions, with 1:M topology, with a push initiator mechanism, and priorizing integrity over timeliness'. Based on these architectural requirements, an architect (or automated tool!) can search a catalog for any existing mechanisms or combination thereof that provides this specified policy; lacking additional restrictions and

using well-known software, a good first fit as mechanism is SMTP (the standard e-mail protocol), and thus any available component that provides it.

3.2 Systematic Generation of Component Assemblies

To illustrate how these concepts relate and are used in practice consider the following example (see Figure 1. The derivation process starts from quality attribute that may be associated to specific *architectural concerns* (e.g. `access control` for `security` requirements, `replication` for `availability`). Architectural concerns can be characterized through *dimensions*, which are discriminating factors among policies (e.g. `authentication type` [16] in access control, `update pro pagation type` [17] for replication). Each dimension can be satisfied by some *architectural policies* (e.g. `authentication based-on-something-that-the-user-knows`, `operations-based update propagation`). Each policy may be satisfied by several *architectural mechanisms* (e.g. `SMTP-AUTH` for authentication based-on-something-that-the-user-knows, `active replication` for replication with state-based update propagation). Finally, mechanisms may be provided by one or more available *components*, which in turn may implement several mechanisms (e.g.`SendMail` v8.1 and later for SMTP-AUTH; `LifeKeeper` for active SMTP server replication en Linux).

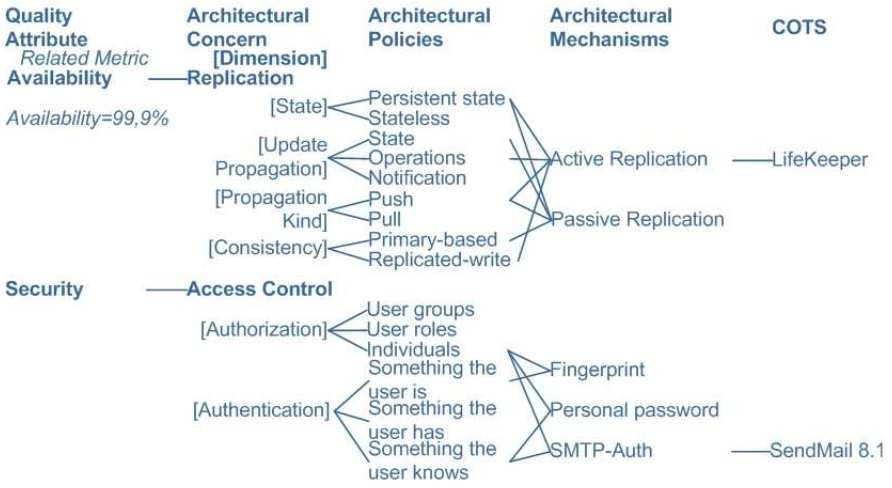


Fig. 1. Example of Systematic Generation of Component Assemblies

The selected components are organized in alternative assemblies that aim to satisfy all the systemic properties at once. Assemblies are later subject to evaluation choose among them using some system-wide criteria (e.g. cost, or smallest number of suppliers). This process is described in Figure 2.

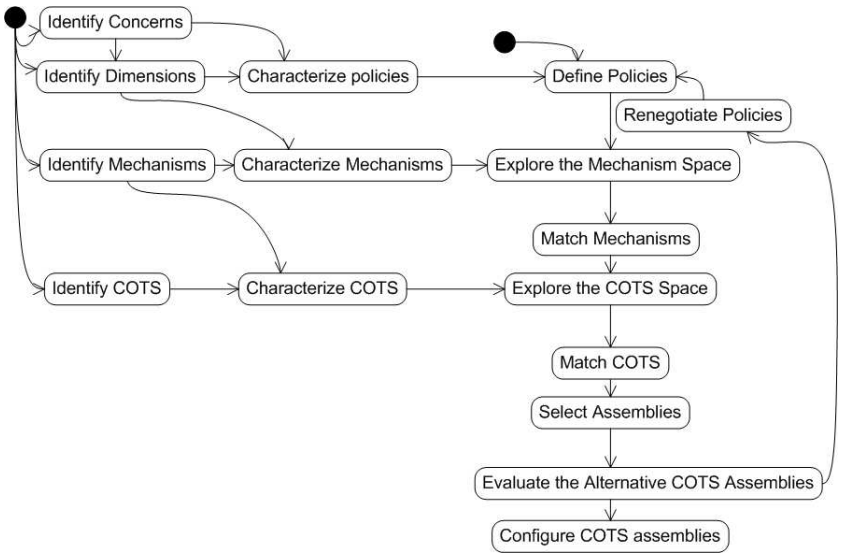


Fig. 2. Generation and Evaluation of Component Assemblies

3.3 Systematic Exploration of Design Spaces

Architects repeatedly perform derivations from systemic properties to possible solutions, identifying and evaluating those that each architectural mechanisms provides, as well as each selected component. These alternative solutions and reifications are the design space that the architect must explore, and which is currently done in an intuitive manner. As shown in Figure **??**, after identifying potential architectural mechanisms ('Match Mechanisms') comes an exploration of the components space to determine which ones implement them. The result is a set of alternative components ('Match COTS') from which the alternate component assemblies are generated to be evaluated. Notice that the solution space is generally quite large, highly changing and in constant growth, mainly due to the dynamic components market. In an open market of independent component developers, the set of possible combinations is not known to any of the involved parties [19]. The architect's knowledge of architectural mechanisms and available components (held a priori or acquired in the ongoing selection process) is the basis for reasoning that justifies selection decisions. Thus, keeping in catalogs information about which mechanisms satisfy which policies and which components implement which mechanisms allows sharing this valuable knowledge; and identifying derivation rules allows supporting, and perhaps even semi-automating, the exploration process performed by architects.

Alternative assemblies can be evaluated to select the one that best fits the specified requirements, matches the platform restrictions, and meets the non-technical selection criteria, such as minimal cost, minimal number of suppliers, and maximal suppliers' reliability.

When new information becomes available, or when requirements change, regeneration of component assemblies is called for. New assemblies may include other mechanisms and/or new components, or in fact drop some and consolidate others. To support these generation, evaluation and regeneration processes, and the consequent design space exploration, we deploy multi-dimensional catalogs to characterize policies, mechanisms and components, and systematic derivation rules among these levels, as shown in Figure 3 and explained in Section 4.

4 Multi-dimensional Catalogs

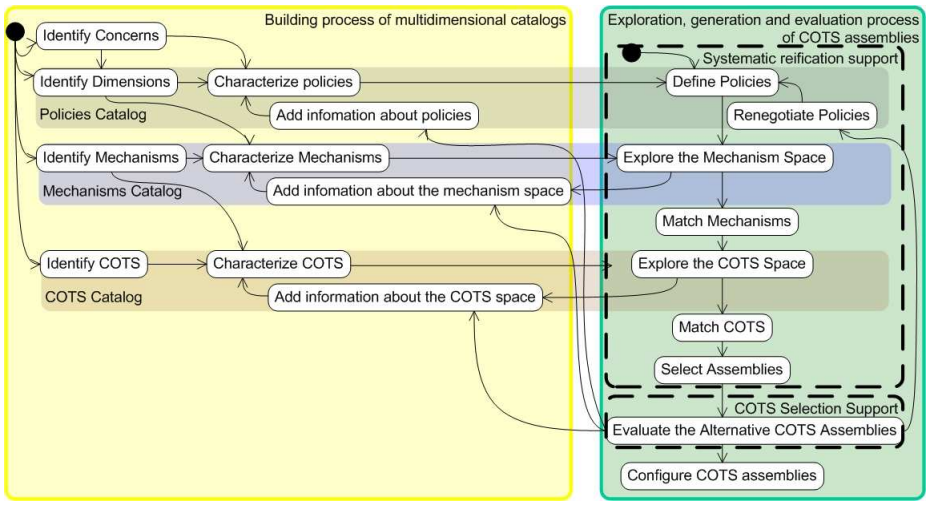


Fig. 3. Multidimensional Catalogs for Exploration, Generation and Evaluation of Component Assemblies

Catalogs store architects' knowledge about architectural policies, mechanisms and components, as well as the derivation rules among them. Thus, they are the key to reusing information about previous selection processes; improving knowledge quality about design spaces and components insofar as better descriptions are stored; and supporting architects in the exploration of these design spaces.

Figure 3 describes the two roles that catalogs fulfill: as repositories of information necessary to generate, evaluate and regenerate component assemblies; and as actively maintained descriptions of the components available in a given milieu. The parallelism and mutual feedback of these two processes allow to use catalog information and derivation rules for selection decisions, and to add information to the catalog when some ongoing selection process gathers additional data.

This section will illustrate the deployment and use of catalogs with a running example. Consider propagation of stock prices information, and the requirement *'the system shall send a report to each customer according to his stocks portfolio; this service must have 99.9% availability and provide access security.'*

4.1 Policy Catalogs

The *policy catalogs* gathers platform-independent architectural policies and stores dimensions for each concern and policies that have different values for each dimension. The catalog incorporates knowledge for each architectural concern, and the dimensions themselves are collected from authoritative sources of the relevant discipline (e.g. Tanenbaum [17] for replication, Britton [18] for middleware communication, and Firesmith [16] for security). Figure 4 shows a partial content of the policies catalog.

Choosing among the policies shown in Figure 4, we notice that the system requires `Asynchronous` *Communication Type, with* `1:M topology`*, with* `Push initiator`*, and communication must privilege* `Integrity over Timeliness`*. Security is focused on Access Control, and the usual policies are* `Individual Authorization` *and* `Authentication based on something the user knows` *[20]. Availability is represented by several architectural concerns, such as* `Replication`*,* `Recovery` *and* `Failure Monitoring`*; here, we'll use only Replication. To meet the availability requirement, we define replication policies with* `Persistent State` *and* `Replicated Write Consistency`*.*

Independently of the suggested use of catalogs as stepping stones in larger derivation chains, it should be noticed that even a stand-alone catalog of architectural policies (however incomplete) would be useful to help in representing (and thus negotiating and validating) quality attributes, as long as the relevant concerns, dimensions and policies are present.

4.2 Mechanism Catalogs

The *mechanism catalog* records known architectural mechanisms, which are implementation-independent design-level constructs that satisfy architectural policies. This catalog indicates which mechanisms satisfy which policies, and characterizes each mechanism with the values of each concern dimension that it can satisfy. A given mechanism may implement several policies for a same concern, or policies across several concerns; similarly, a given policy may be implemented by several mechanisms. Figure 5 shows partial content of the mechanisms catalog.

In real-world deployment situations, the catalog preparators might not know or not be certain whether a given mechanism supports a certain policy. To account for this uncertainty, the mechanisms catalog allows five degrees of certainty regarding support for a given policy: 'supports'(1), 'probably supports'(0,6),'probably does not support'(0,3), 'does not support'(0), and

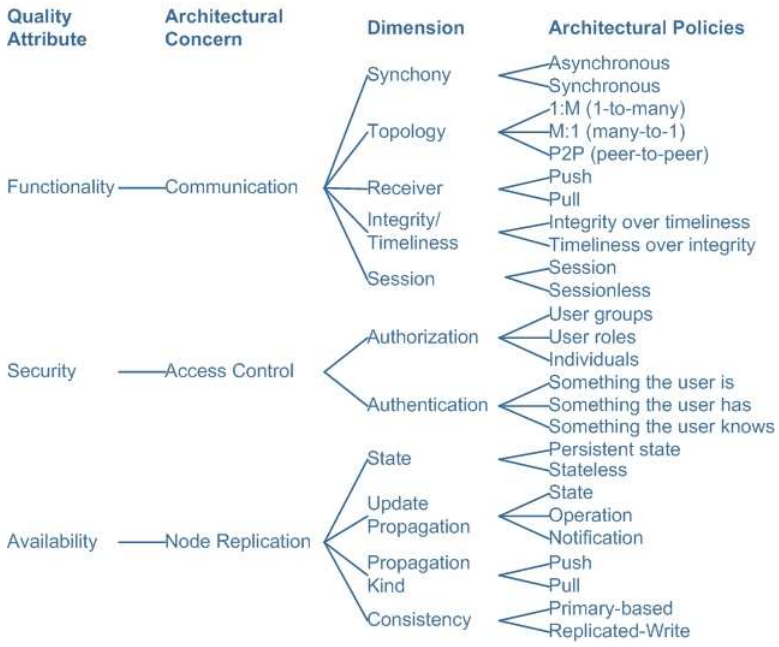


Fig. 4. Partial Content of Policies Catalog

'unknown'(empty) (since absence of knowledge differs from knowledge of absence). Incidentally, current work is using fuzzy optimization techniques on these uncertainty-rich descriptions to evaluate and regenerate component assemblies.

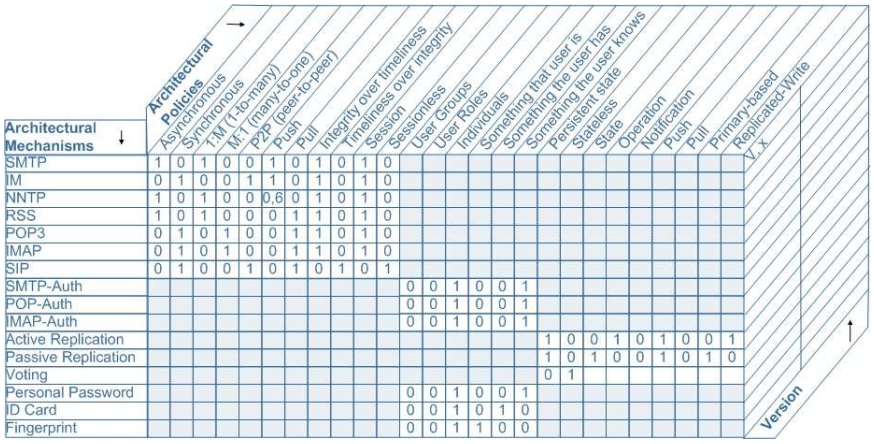


Fig. 5. Partial Content of Mechanisms Catalog

The catalog shown in Figure 5 sugggests that the architect has several options to satisfy the specified policies: Communication Type may me reified with the NNTP *protocol (used to post subscription-based "news") or* SMTP *(used to send e-mail). The Access Control policies might be satisfied with a* `personal password` *mechanism. The protocols SMTP-Auth, POP-Auth, IMAP-Auth and Personal Password do satisfy the requirements of Access Control. The replication policies can be satisfied with* `active replication`*.*

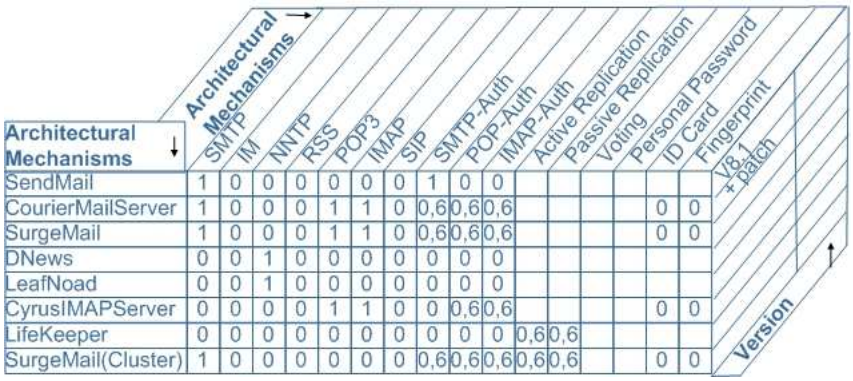
4.3 Components Catalog

The *component catalog* describes the space of components. A software component [19] is multiple-use, non-context-specific, composable with other components, encapsulated (i.e., non-investigable through its interfaces) and a unit of independent deployment and versioning. In Azimut, components are characterized according to the architectural mechanism(s) that implement. A given component may implement several mechanisms; similarly, a given mechanism may be implemented by several components. Besides mechanism support, the components catalog has four dimensions:

- **Uncertainty** Just Similarly to the mechanisms catalogs, the components catalog allows five degrees of certainty regarding support for a given mechanism: 'supports', 'probably supports', 'probably does not support', 'does not support', and 'unknown'.
- **Available platforms** Platform(s) under which the component can be deployed (e.g. Windows, Linux, Solaris).
- **Market issues** Component selection requires using non-technical criteria to distinguish among otherwise equivalent alternate components and assemblies. The catalog includes some key characteristics: Supplier; Market Share [11]; Supplied Reliability [6] (valued from 1 to 5, with higher values for higher reliability); Initial Cost [11]; Integration Cost [11]; and Support Cost.
- **Description credibility** An important criterion is the credibility degree [5] of the component description, which quantifies confidence regarding descriptions. We follow Philips and Polen [5] in assigning credibility values for descriptions: (1) user- or supplier-provided, or seen in third-party literature; (2) seen but not studied; (3) witnessed in personalized demos; and (4) verified hands-on "in-house".

Figure 6 shows partial content of a components catalog relevant to the running example.

Several mechanism configurations are possible, and in fact some components do implement each desired mechanism. Components that implement mechanisms that satisfy all required quality attributes are `LifeKeeper` *and* `SendMail` *(v8.1 and later; notice that earlier versions might also be recorded in the catalog); or* `SurgeMail (Cluster)`*. Choosing among them means having an additional goal function: if it is minimizing number of components (to reduce complexity), the optimal solution is* `SurgeMail (Cluster)`*, but if it is minimizing costs,*



Architectural Mechanisms ↓	SMTP	IM	NNTP	RSS	POP3	IMAP	SIP	SMTP-Auth	POP-Auth	IMAP-Auth	Active Replication	Passive Replication	Voting	Personal Password	ID Card	Fingerprint	V8.1 + patch
SendMail	1	0	0	0	0	0	0	1	0	0							
CourierMailServer	1	0	0	0	1	1	0	0,6	0,6	0,6					0	0	
SurgeMail	1	0	0	0	1	1	0	0,6	0,6	0,6					0	0	
DNews	0	0	1	0	0	0	0	0	0	0							
LeafNoad	0	0	1	0	0	0	0	0	0	0							
CyrusIMAPServer	0	0	0	0	1	1	0	0	0,6	0,6					0	0	
LifeKeeper	0	0	0	0	0	0	0	0	0	0	0,6	0,6					
SurgeMail(Cluster)	1	0	0	0	0	0	0	0,6	0,6	0,6	0,6	0,6			0	0	

Fig. 6. Partial Contents of Components Catalog

the other option is better. Other alternatives are looking for additional information (and enrich the catalogs); considering ad-hoc implementation of passive replication; or outsourcing the replication service and defining in the SLA an availability target of 99.9%. At this point, active exploration of design spaces by the architect should ensue.

Another difference between catalogs is the global and authoritative nature of the policies and mechanisms catalog versus the local nature of the component catalog in each organization. In fact, there might be sub-catalogue suppliers for a global component information repository.

4.4 Recording Feedback into Catalogs

A better evaluation could inject some new information to the selection process as well: new descriptions (characterizations) of components and mechanisms to increase the knowledge of solutions spaces, or new policies to better describe some requirements; or it might suggest renegotiation of requirements if impossible to find any assemblies that satisfy all given requirements (see Figure 3).

Thus, an additional advantage of these catalog-based process is that exploration of mechanisms and components feeds back into the catalog construction process (see Figure 3).

5 Automation of Derivation Process

Based on the several platform abstraction levels, we can identify derivation rules among them (the relationships 'provides' among mechanisms and policies, and 'implements' among components and mechanisms), as well as combination restrictions. Automating these derivation rules allows proposing components and assemblies dynamically to the architect. Currently, we are at work in two alternative approaches to achieving automation: one rule-based (herein shown),

and one based on combinatorial optimization algorithms [**?**]. Both approaches try to avoid the complexity of assigning weights to the influence of each solution element (mechanism, component) on each goal, unlike AHP (the multi-criteria decision technique used by several CBD methods [5–8]).

Combinatorial optimization techniques have allowed us to explore some very interesting problems, like treatment of fuzzy data (such as 'probably supports'), information variability at the level of both requirements and components, treatment of conflict among mechanisms or components as restrictions, and incompatible combinations.

Azimuy possibly uses incomplete, imprecise, unreliable and changing descriptions of architectural policies, mechanisms and components. As mentioned above, these characteristics allow using the catalogs even during early definition stages, to help with requirements definition and validation.

Later on, assemblies that are proposed in the absence of full knowledge (i.e. catalogs with several 'unknown' entries) may turn out to be sub-optimal regarding number of components or some other criterion, but new information will not necessarily invalidate it (unless it generates a conflict).

Fuzzy information is a normal situation in architecture development, since incomplete and imprecise information is what most architects actually have at hand. Accepting this fuzzyness and dealing with it is a key step to supporting actual architects elaborating actual software systems.

5.1 Rule-based Prototype

We have developed a prototype to validate the feasibility of this approach. Rules [27] describe the "characterizations" of policies, mechanisms, and components, and relationships among them and the other attributes. Using these rules, the system generates component assemblies that satisfy the required policies. The prototype deals with fuzzyness by showing first solutions based on 'supports' and 'implements', and later the fuzzy attributes, but currently it optimizes for simples non-technical attributes (e.g. minimum number of components, or total cost). Examples of these rules are shown in Table 1.

Figure 7 shows the output given by the prototype when you search assemblies satisfying the policies of our example.

6 Ongoing and Future Work

Work in progress includes expanding the kinds of recorded information in catalogs; identifying further derivation rules; implementing algorithms to treat fuzzy information [25]; and managing conflicts among mechanisms or components.

Also, some computationally hard problems are being studied jointly with combinatorial optimization researchers to analyze trade-offs among several selection criteria; what-if analysis to quantify the impact of requirements changes; and reverse questioning, i.e. determining satisfiable requirements given a set of components.

Table 1. Rules

satisfies(smtp,[asynchronous,synchrony,communication_type]).
satisfies(smtp,[1:m,topology,communication_type]).
satisfies(smtp,[push,receiver,communication_type]).
satisfies(smtp,[integrity_over_timeliness,integrity/timeliness,communication_type]).
satisfies(active_replication,[persistent_state,state,node_replication]).
satisfies(active_replication,[replicatedwrite,consistency,node_replication]).
satisfies(smtp_auth,[individuals,authorization,access_control]).
satisfies(smtp_auth,[something_the_user_knows,authentication,access_control]).
satisfies(rss,[asynchronous,synchrony,communication_type]).
satisfies(rss,[1:m,topology,communicatin_type]).
satisfies(rss,[pull,receiver,communication_type]).
probablySatisfies(nntp,[push,receiver,communication_type]).
satisfies(nntp,[integrity_over_timeliness,integrity/timeliness,communication_type]).
....
implements(sendMail,smtp). implements(sendMail_v8_1,smtp).
implements(sendMail_v8_1,smtp_auth). implements(surgeMailCluster,smtp).
implements(dNews,nntp). implements(leafNoad,nntp).
probablyImplements(lifeKeeper,active_replication).
probablyImplements(lifekeeper,passive_replication).
probablyImplements(surgeMailCluster,smtp_auth).
probablyImplements(surgeMailCluster,active_replication).
....
notRelatedCots(lifeKeeper,dNews). notRelatedCots(lifeKeeper,leafNoad).
notRelatedCots(lifeKeeper,surgeMailCluster).

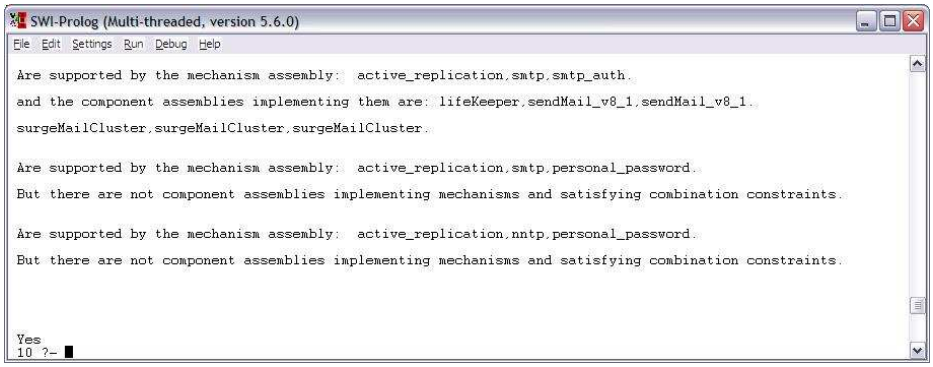


Fig. 7. Prototype: Output for the example

6.1 Application: MDA

Model-Driven Architecture (MDA) [21] aims to derive/generate software systems through systematic transformations from high-level models. Some projects, such as CoSMIC [22] and UniFrame [23], implement MDA to generate component-based systems, but use formal component specification languages to describe

the available components, and from these descriptions (consistent and precise) they automate the component selection and integration process.

However, in most systems without strong constraints like hard real-time, the cost of using formal specifications is difficult to justify; thus, we aim to integrate incomplete, imprecise, unreliable and changing descriptions into MDA techniques. Current systematic techniques to select components are hard to integrate with MDA due to the lack of explicit mappings among PIM-level concepts of analysis and design, and PSM-level constructs such as components.

We have deployed the described approach and techniques in the Azimut framework [24], which extends MDA to automate architectural decisions from NFRs through components. The prototype is described in [26].

7 Conclusions

The described process to generate, evaluate and regenerate component assemblies, combined with the multi-dimensional catalogs that support it, allows architects to engage in iterative exploration of design spaces. A key goal of this exploration is finding the "best" combination of components that not only satisfy the given requirements, but also fit some non-technical second-order criteria (such as minimal cost or maximal supplier reliability), but accepting the fuzzy nature of available component information.

The underlying concepts are representation of quality attributes using architectural policies, their systematic reification into architectural mechanisms, and reification of mechanisms into components that implement them. The main operational feature of the approach are *catalogs* for three abstraction levels (policies, mechanisms and components); these abstractions are "characterize" with possibly *incomplete, imprecise, unreliable and changing* data, and are *multi-dimensional* in including technical data but also higher-order information (e.g. cost, supplier). Thus, keeping in catalogs information about which mechanisms satisfy which policies and which components implement which mechanisms allows sharing this valuable knowledge; and identifying derivation rules allows supporting, and perhaps even semi-automating, the exploration process of design spaces performed by architects.

References

1. Sihem Ben Sassi, Lamia Labed Jilani, Henda Hajjami Ben Ghezala: "COTS Characterization Model in a COTS-Based Development Environment." APSEC 2003, p. 352.
2. Ncube, C., Maiden, N. "PORE: Procurement-Oriented Requirements Engineering Method for the CBSE Development Paradigm." International Workshop on CBSE, May 1999.

3. Alves, C., Finkelstein, A.: "Challenges in COTS-Making: a Goal-Driven Requirements Engineering Perspective." Proc. 14th Intl. Conf. on Software Engineering and Knowledge Engineering (SEKE'02), Italy (July 2002).
4. Ochs, M.: "A COTS Acquisition Process: Definition and Application Experience." 11th ESCOM Conference, Shaker, Maastricht, 2000.
5. Philips, B., Polen, S.: "Add Decision Analysis to Your COTS Selection Process." The Journal of Defense Software Engineering, Software Technology Support Center Crosstalk, April 2002.
6. Kontio, J.: "A case study in applying a systematic method for COTS selection." Proceedings ICSE 1996, p. 201-209.
7. Kunda, D., Brooks, L.: "Applying Social-Technical Approach to COTS Selection." Proceedings 4th UKAIS Conference, Abril 1999.
8. Alves, C., Castro, J.: "CRE: A Systematic Method for COTS Components Selection." 15th Brazilian Symposium on Software Engineering (SBES), Rio de Janeiro, Brazil (Oct 2001).
9. Chung, L., Cooper, K.: "COTS-Aware Requirements Engineering and Software Architecting." Proceedings IWSSA 2004
10. Chung, L.,Nixon, B., Yu, E., Mylopoulos, J.: *Non-Functional Requirements in Software Engineering.* Kluwer Academic Publisher, 2000.
11. Chin Yeoh, H., Miller, J.: "COTS Acquisition Process: Incorporating Business Factors in COTS Vendor Evaluation Taxonomy." METRICS 2004, pp. 84-95.
12. Saaty, T.: "The Analytic Hierarchy Process". New York: McGraw-Hill, 1990.
13. Albin, S.: *The Art of Software Architecture: Design Methods and Techniques.* Wiley, Mar 2003.
14. Policy and Mechanism Definitions. `http://wiki.cs.uiuc.edu/MFA/Policy+and+Mechanism`
15. Vitharana, P.,Fatemah "Mariam" Zahedi, Jain, H.: "Design, retrieval, and assembly in component-based software development." Commun. ACM (46)11, Nov 2003, p.97-102.
16. Firesmith, D.: "Specifying Reusable Security Requirements." Journal of Object Technology, 3(1), pp.61-75 (Jan-Feb 2004). `http://www.jot.fm/issues/issue_2004_01/column6`
17. Tannenbaum, A., van Steen, M.: *Distributed Systems Principles and Paradigms.* Prentice Hall (2002).
18. Britton, C., Bye, P.: *IT Architectures and Middleware: Strategies for Building Large, Integrated Systems (2nd Ed).* Addison-Wesley Professional (2004).
19. Szyperski, C.: *Component Software* (2nd Edition). Addison-Wesley Professional(2002).
20. Authentication Mechanisms. `http://sarwiki.informatik.hu-berlin.de/Authentication_Mechanisms`
21. *MDA Guide Version 1.0.1.* Object Management Group (June 2003). `http://www.omg.org/cgi-bin/doc?omg/03-06-01`
22. Gokhale, A., Balasubramanian, K., and Lu, T. "CoSMIC: Addressing Crosscutting Deployment and Configuration Concerns of Distributed Real-Time and Embedded Systems." OOPSLA 2004, ACM Press, p. 218-219.
23. Cao, F., Bryant, B., Raje, R., Auguston, M., Olson, A., Burt. C: "A Component Assembly Approach Based on Aspect-Oriented Generative Domain Modeling." ENTCS 2005, pp.119-136.

24. López, C., Astudillo, H.: "Explicit Architectural Policies to Satisfy NFRs using COTS." Workshop NfC 2005 in MoDELS'2005, Oct 2005. In: *Satellite Events at the MoDELS 2005 Conference*, Bruel, Jean-Michel (Ed.), LNCS 3844, pp. 227 - 236, Springer (Jan 2006).
25. Astudillo, H., Pereira, J., López, C.: "Evaluating Alternative COTS Assemblies from Unreliable Component Information." Technical Report DI-2006/05, Departamento de Informática, Universidad Técnica Federico Santa María, Valparaíso, Chile (2006).
26. Montenegro, A., Astudillo, H.: "Generation of hybrid code+COTS systems." Technical Report DI-2006/06, Departamento de Informática, Universidad Técnica Federico Santa María, Valparaíso, Chile (2006).
27. SWI Prolog Documentation. `http://www.swi-prolog.org/`