

Towards an Agent-based Framework for Guiding Design Exploration

J. Andres Diaz-Pace Software Engineering Institute 4500 Fifth Avenue Pittsburgh, PA 15213 (USA) adiaz@sei.cmu.edu	Len Bass Software Engineering Institute 4500 Fifth Avenue Pittsburgh, PA 15213 (USA) ljb@sei.cmu.edu	Felix Bachmann Software Engineering Institute 4500 Fifth Avenue Pittsburgh, PA 15213 (USA) fb@sei.cmu.edu	Phil Bianco Software Engineering Institute 4500 Fifth Avenue Pittsburgh, PA 15213 (USA) pbianco@sei.cmu.edu
--	---	--	--

ABSTRACT

One of the premises of conceptual design is that the designer must evaluate a range of candidate solutions before selecting the final solution. Tool support is critical to aid designers in that exploration, because the design space is usually large and involves multiple constraints. A modality of assistance is that the tool criticizes the current design and provides the designer with recommendations for improving it. Traditional knowledge-based systems and optimization tools tend to be inappropriate when the designer is actively involved in the search loop, because the design proposals should match the designer's context. To address this challenge, this paper describes an agent-based framework for developing design recommendation tools that help designers to perform explorative search more effectively. The approach is exemplified with an experimental design assistant for software architecture design.

Categories and Subject Descriptors

D.2.2 [Software Engineering]: Design Tools and Techniques – computer-aided software engineering (CASE).

General Terms

Design

Keywords

Design assistance, agents, knowledge-based recommender, ArchE

1. INTRODUCTION

Conceptual design is an engineering phase that takes a behavioral specification for a product as input and synthesizes solutions for that product as output. Examples of products can be: electronic circuits, software systems, chemical compounds, operational plans, etc. From a constructive perspective, design is a mapping “from functionality to structure” under a set of constraints [8, 9].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

RSSE '08, November 10, Atlanta, Georgia, USA

Copyright 2008 Carnegie Mellon University.

Thus, design can be formulated as a *search problem*, in which the designer navigates the space of possible components and their configurations [4, 10]. This space is usually so complex that tool support is indispensable to assist designers in the search.

In large design spaces, the search generally has an *explorative purpose*, such as: understanding the space characteristics or identifying feasibility regions for solutions. The designer's primary goal is to “sample” the space and analyze how solutions behave regarding a set of quality criteria [10]. Furthermore, the designer gets actively involved in the exploration, manipulating the problem constraints as the search proceeds. Constraints can be seen here as *user's preferences* for the search. In this context, the role of tool assistance is to make design proposals (e.g., exemplar solutions, promising directions in the design space) that match the designer's context, rather than leading the designer linearly to an optimum. The development of such design assistants requires a knowledge-level approach (similarly to many AI approaches developed in the past, e.g. [4, 12, 13]). Furthermore, the approach must focus on generating design proposals perceived as “useful” by the designer, even with incomplete knowledge-based modeling and reasoning capabilities.

For several years, the authors have been involved in the development of knowledge-based tools for software design [2, 7]. The most recent tool is ArchE [6], an assistant for architecture design. Based on these experiences, we see three key challenges (in addition to search) for proactive design assistants:

1. **An explicit design theory.** It provides the building blocks for representing problems and solutions, and for reasoning about them. It should also include a problem-solving design strategy and criteria to assess progress during search.
2. **A user-oriented metaphor.** A design tool should be user-friendly and give support for “opportunistic design” [13]. This is so because people (and therefore, designers) have a limited ability to process information. Also, it is not possible to encode (and automate) all the knowledge sources needed for doing design. The metaphor for a proactive tool is that of a secretary or *personal assistant* to the architect [11].
3. **Support for tool and knowledge integration.** Currently, there are many specialized, but powerful, design tools. Consequently, in the development of a design assistant, we should enable the incorporation of third-party tools in an incremental way, as needed by the design domain.

Several knowledge-based tools and critiquing systems for design have been proposed within the AI community. However, few

design tools have taken some of the above challenges into account [8, 10, 13]. The majority of these tools has been targeted to specific domains or design tasks. *Agent-based assistance* (also called intelligent agent technology) is an integration of AI techniques with software agents [5, 11] that holds promise for explorative tools. This approach augments the designer’s capabilities when searching, and it is also flexible for handling constraints as preferences. The preferences may change over time, depending on the designer’s context. In this paper, we describe an agent-based recommendation framework for design exploration that is amenable for tool support. The framework has two parts: a design model and an agent architecture, which are a generalization of previous work on software design assistants. Specifically, the framework is able to generate design proposals as questions for the designer, so that she can analyze tradeoffs and select the proposal that best fits her needs.

The rest of the paper is organized into 4 sections. Section 2 describes the design model used by the framework. Section 3 describes the main characteristics of the agent-based architecture. Section 4 briefly comments on the ArchE design assistant, as an instantiation of the general framework. Section 5 presents the conclusions of the work and discusses issues for future research.

2. DESIGN MODEL

A central aspect of “design as a search” is about modeling the design space in which the designer makes her decisions. We conceptualize this space in terms of three sub-spaces: the functional, structural and quality spaces. This division is consistent with the function-behavior-structure approach [8].

Design solutions exist to satisfy some purpose or function. The functional space refers to symbolic functions that represent the intended behavior of the solution. Designers can abstractly work with functions without making commitments to a particular design structure. The structural space comprises basic components and arrangements of components into larger components. Normally, a library of “reusable” components and possible relationships between components is assumed. There is a “function-means” mapping between the functional and structural spaces: functions are decomposed into responsibilities that are embodied by different components. At last, the quality space defines constraints on the way functions should work together when mapped to components. Constraints serve to evaluate function-structure pairs through some quantifiable measure. That is, we can numerically determine whether a constraint is met. Related constraints can be grouped under a quality criterion for tradeoff analysis (e.g., performance, allowed component configuration, cost, etc.).

Let’s consider the design of a bicycle [12], whose expected functionality is to provide transport. In the functional space, the provide-transport function is decomposed into child functions such as: facilitate-movement, provide-energy, support-passenger, change-direction and provide-support. Let’s assume that each child function maps to a corresponding responsibility. Responsibilities can take part in relationships. For instance, provide-energy drives facilitate-movement, change-direction depends on provide-support, provide-energy depends on provide-support, etc. Here, drives and depends-on are examples of binary responsibility relationships. We have available a set of bike-related components, such as: saddle, wheel, pedal, handlebar, or frame. The saddle component realizes support-passenger, pedal realizes provide-energy, and so

forth. Responsibility relationships prescribe the design relationships that should exist between the components allocating the responsibilities. For instance, the drives relationship between facilitate-movement and provide-energy means that if these responsibilities are realized by wheel and pedal respectively, then there must be a design relationship between those components. Figure 1 shows a component assembly that realizes all the responsibilities, thus achieving the function provide-transport.

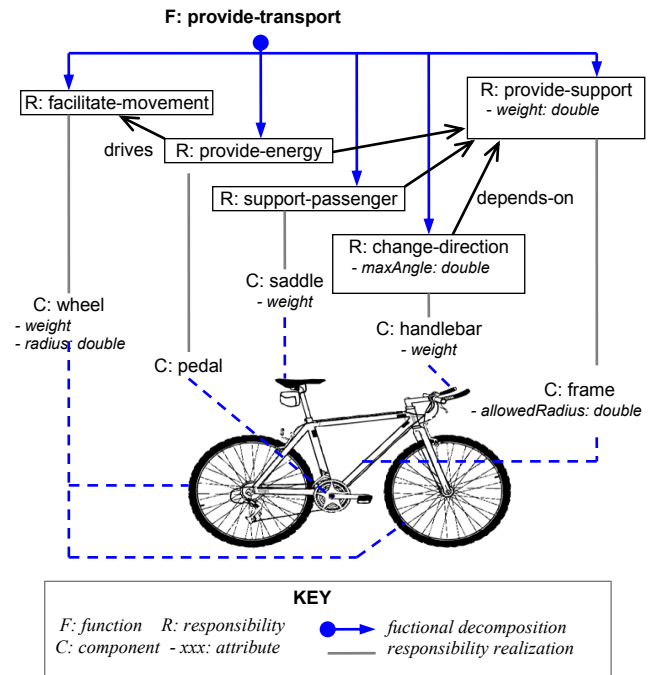


Figure 1. Function, structure and quality attributes, when designing a bicycle (adapted from [14])

Functions, responsibilities, relationships and components can be annotated with attributes. For instance, an attribute may be: the frame weight, the component color, the angle of allowed movements for change-direction, etc. On the basis of attributes, one can specify constraints on attribute values. Examples of constraints can be: “the weight supported by the bicycle should not exceed X kg”, or “the frame should only support wheels with diameters between A and B cm”. The concepts developed for this bicycle example are also applicable to other design domains.

2.1 Making Design Proposals

A design solution (in the structural space) may fail to provide the expected functionality or to fulfill some quality criteria. If so, a design assistant should provide recommendations to the designer. A recommendation can either make changes to the solution or to a functional subpart of the problem. In the design space above, there are five mechanisms for repairing a problem or solution:

- **Reduce functionality.** Functions that are not essential for a solution (or that could be added in the future) can be removed from the functional specification. Having fewer functions should lead to a different (smaller) component structure that satisfies the constraints.
- **Adjust attributes.** Changes in attribute values of functions, responsibilities, components or related relationships can indirectly affect the evaluation of constraints.

- **Re-arrange responsibilities.** Decomposing a function into different responsibilities changes the responsibility allocation to components, while ensuring that the function is still delivered. This new mapping may then meet the constraints.
- **Change design structure.** If functionality cannot be altered, re-structuring the components can help satisfying the constraints. The modification of the component structure is also known as a *design transformation*.
- **Relax constraints.** If constraints are too restrictive, they can be reformulated so that some of their parts are weakened.

In order to make recommendations, an assistant must carry out two tasks [4]: i) identify the causes of the “design failure”, and ii) select suitable repairing mechanisms. This is similar to critiquing systems [13], although critics may not always repair the failure. In general, the assistant will rely on design heuristics that connect predetermined structures and functions with predictable results in the quality space. For example, ArchE implements this connection by means of reasoning frameworks and tactics [2]. A reasoning framework is a reusable knowledge source that encapsulates a quality-attribute theory for analyzing software architectures. A tactic is a design transformation for the architecture that is driven by the quality-attribute analysis results provided by a reasoning framework. In fact, tactics are recommendations derivable from the mechanisms listed above, which are activated by heuristic rules associated to each reasoning framework.

An interesting aspect of a “quality-driven” recommendation approach to explore the design space is the delivery of useful design proposals, without requiring detailed domain knowledge or complex reasoning procedures. Furthermore, knowledge sources can be updated over time. For example, ArchE supports the integration of new reasoning frameworks and tactics for different qualities. Nonetheless, the designer must still know about the quality criteria, the semantics behind functions and structure, and must judge if the recommendations are relevant to her context.

3. AGENT ARCHITECTURE

The framework architecture is a blackboard model that achieves collaboration among knowledge sources [5]. Since the late 90’s, the blackboard model has been revisited from an agent-centric perspective, emphasizing engineering features such as autonomy, modularity, and decoupling of the system. The knowledge-based recommendation framework [3] involves two types of agents: interface agents and expert agents. Interface agents are responsible for user interaction issues and design proposals. These agents rely on expert agents that have expertise on particular quality-driven theories. A control component manages the work of the agents over the blackboard. Figure 2 shows the reference architecture that supports the design model of Section 2. The ArchE prototype derived from this architecture will be described in Section 4.

The Design Repository stores all the data items being processed during an exploration session, such as: functions, responsibilities, quality criteria, components, transformations, etc. Control data is also stored for coordination between the Seeker and the agents.

The Main GUI is the front-end that allows the designer to enter design information, and displays results and user questions. These questions are primarily intended for design proposals to move in the design space. Questions also show warnings about the design (e.g., incorrect parameters, inconsistent configurations).

The Main GUI comes with a visual metaphor called “traffic light”. This is a decision table that summarizes the available design proposals versus the evaluation of quality criteria. The payoffs of each alternative are shown using color-coded icons. The designer is at the driving seat: she analyzes the tradeoffs in the decision table and decides on the design transformations to be executed. In addition to looking at the quality criteria, the designer is likely to choose proposals based on “non-technical” considerations, e.g. solutions she is familiar with, solutions that reuse existing assets, or solutions aligned with organizational practices, among others. These considerations reflect the *utility* of the proposal from the designer’s viewpoint. To support this feature, the agents are able to evaluate a solution either in terms of quality or utility criteria.

The Interface Agent works embedded in the Main GUI. This agent monitors the designer’s actions when she is interacting with the design tool, in order to know the designer’s preferences (also called user’s profile). These preferences have two purposes. First, they serve to inform the search for design proposals, as implemented by the Seeker and the Expert Agents. Second, preferences are used by the Interface Agent to rank and filter questions for the designer. This agent can also take the user’s feedback for making better suggestions in the future.

The Seeker implements the search strategy that determines when the agent capabilities should be activated, and the order in which design proposals are generated and evaluated against the quality criteria. The Seeker has no semantic knowledge of the system being designed (e.g., representation via functions, responsibilities, components, or agent computations). That is, when the Seeker gets notified about user’s inputs, it delegates the function-to-structure mapping work to the Expert Agents, and then assembles their results for the Interface Agent. The agent communication takes place through a command-based publish-subscribe schema.

The Expert Agents can react to events of interest in the blackboard, or respond to commands from the Seeker. Each expert agent handles a specific quality criterion, and the agent can independently evaluate its goals and execute different actions accordingly. The actions include: critiquing the current solution, searching possible transformations for improving the current design, creating user questions, invoking specific tools to implement the previous actions, etc. An agent has knowledge to the extent necessary for dealing with its quality of interest.

The search strategy for exploring alternative designs relies on a “propose-critique-modify” approach [4]. The search is a cycle of four steps [6]. In step 1, if no initial solution exists, each expert agent proposes a solution according to its own quality criterion. In step 2, the solutions are checked by the rest of the agents and by the designer. If the solution and its quality tradeoffs are satisfactory for the designer, the search stops. If not, step 3 is to have the agents analyze the current solution for design flaws. The agents use their knowledge to seek for repairing mechanisms and make design proposals. If no proposals are found, the search is suspended. Here, the designer may go back to previous steps and start a different exploration. In case proposals are suggested, step 4 is about applying those proposals to the current solution. This way, the Seeker constructs a tree of candidate solutions, in which each branch corresponds to a design proposal and each node represents the solution derived from that proposal. After step 4, the search cycle moves to step 2 and every child solution is analyzed by all the agents.

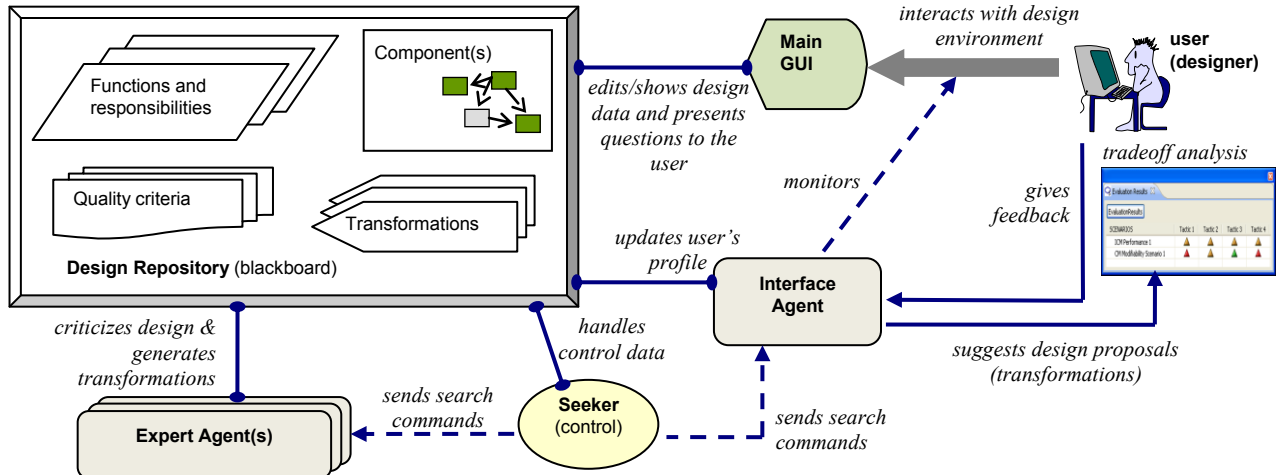


Figure 2. Agent-based blackboard approach to design assistants

The design tree is expanded a fixed number of levels, and a decision rule selects the most promising solution at each level (like in traditional game trees). If any change is made to the design (either by the user or by some agent), the Seeker starts over the searching cycle from the root node, so as to adapt the design proposals to the new situation. Details of the search algorithm are beyond the scope of this paper.

4. CASE-STUDY: ARCHÉ

ArchE¹ (Architecture Expert) is a design assistant developed by the Software Engineering Institute that helps architects to explore alternative architecture designs. Table 1 summarizes the relationships between the concepts of the agent architecture and the ArchE counterparts. Figure 3 shows ArchE at work. In terms of assistance, ArchE asks the reasoning frameworks to analyze quality-attribute scenarios over the current architecture and to recommend tactics for it. At the end of each search cycle, ArchE shows a prioritized list of tactics (architectural transformations) to the designer. An example of quality scenario is: "Adaptation of the software to different operating systems should be done within 5 person-days (modifiability)". A possible tactic (user question) for that is: "Dependencies of module X negatively impact scenario S. Do you want ArchE to insert an intermediary with cost C, in order to reduce the cost of the change from A1 to A2 person-days?". The more reasoning frameworks are available to ArchE, the broader its assistive capabilities will be. To accomplish this goal, reasoning frameworks are modeled as expert agents, and third-party researchers can easily configure reasoning frameworks as ArchE plug-ins [6]. We have created two plug-ins for the qualities of modifiability and performance.

As a recommendation system, ArchE "knows" about: (i) the links between quality-attribute analyses and architectural structures; and (ii) how to run a search by delegating on the right reasoning frameworks for each quality criterion. The designer (as a user of ArchE) provides: (i) the meaning of scenarios and responsibilities, (ii) the mapping of scenarios to responsibilities; and essentially, (iii) the design context. That is, the designer can trust the "quality-attribute reasoning" followed

by ArchE when proposing tactics, but she does not have to agree on applying the best tactics found for satisfying her scenarios. Since ArchE has limited searching capabilities, the tactics are often local optima. In other words, the designer uses the tactics as "leads" to alternatives rather than as definite architectural transformations to pursue. The most important part of a tactic is its quality-attribute justification, not the resulting architecture.

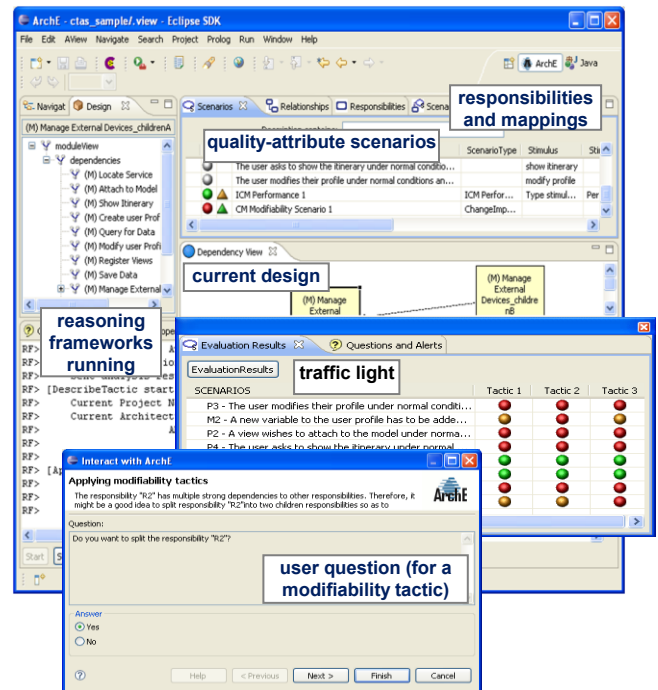


Figure 3. ArchE showing a user question to improve a modifiability scenario

If a reasoning framework does not have enough rules that tell it what tactics to recommend, the necessary knowledge should be provided by architects. A way of capturing that knowledge is via case-based reasoning (CBR) techniques [1]. As a proof of concept, we have recently implemented a CBR system for

¹ ArchE web: <http://www.sei.cmu.edu/architecture/arche.html>

modifiability tactics based on the IUCBRF toolkit². Although rules for suggesting tactics may exist already, the cases help to customize those rules to the architect's preferences. The reasoning frameworks invoke the CBR system and try to reuse tactics from past design problems to solve new problems. The resulting tactics are sent to ArchE as normal user questions.

Table 1. Instantiation of the framework by ArchE

Concept	ArchE Term	Short description
Function	idem	A description of a feature that the system should satisfy. Functions are translated into responsibilities.
Responsibility	idem	An activity undertaken by the software being designed. Responsibilities express functional requirements and are part of quality-attribute scenarios.
Attribute	Responsibility parameter	Responsibilities can have quality-specific properties. The reasoning frameworks use these properties to analyze whether scenarios are met.
Constraint	Quality-attribute scenario	A system-independent table for deriving quality-attribute requirements. Concrete scenarios are created by assigning values to the table entries.
Quality Criterion	Quality-attribute scenario	idem above
Component	Design element	Design elements are grouped into architectural views. Examples are: the module view, the process view, the component-and-connector view, etc
Interface Agent	ArchE GUI	An Eclipse-based GUI.
Seeker	idem	Search strategy based on a "maxmin" decision rule for the design tree.
Expert Agent	Reasoning framework	It encapsulates quality-specific design knowledge, so that this knowledge can be applied by non-experts.
Design proposal	Architectural tactic	A means to control the quality-attribute response of a scenario. Tactics are derived from general design principles, and lead to the application of specific architectural patterns.

5. DISCUSSION

Agent-based assistants constitute a promising engineering approach to support designers in making informed decisions during design explorations. We believe that agents provide good extensibility mechanisms for design tools that need to incorporate, and progressively update, their base of knowledge. In this paper, we have discussed a framework architecture for design search problems with the following characteristics:

- Design is casted to function-to-structure mappings with constraints. Solutions are evaluated via quantitative criteria.
- Recommendations are quality-driven proposals for transforming the functions and/or structure of a design.
- The designer is responsible for analyzing the tradeoffs among quality criteria. The search is guided by the interplay between constraints and design recommendations.

In ArchE, the reliance on expert agents to deal with reasoning framework favored integrability and modular reasoning about quality attributes. Nonetheless, ArchE does not intend to be a "complete" expert system, but rather a platform for plugging in knowledge sources and for making them interact. Along this

line, ArchE is able to search for alternatives, show quality tradeoffs, and adapt its analyses and recommendations to the designer's preferences. However, ArchE cannot make decisions for the designer, and is not concerned with the quality and amount of knowledge codified inside the reasoning frameworks.

Finally, as future work, we plan to enhance the advice offered by the framework, refining the design model into an ontology, and adding design rationale and automated tradeoff reasoning .

6. REFERENCES

- [1] Aamodt A., and Plaza, E. 1994. *Case-based reasoning: Foundational issues, methodological variations and system approaches*. AI Communications, Vol. 7, No 1. - 39-59.
- [2] Bachmann, F., Bass, L., Klein, M., & Shelton, C. 2005. *Designing software architectures to achieve quality attribute requirements*. Software IEE, Vol. 152 Issue 4 - pp. 153-165. UK
- [3] Burke, R. 2000. *Knowledge-based recommender systems*. Encyclopedia of Library and Information Systems, 69(32).
- [4] Chandrasekaran, B. 1990. *Design problem solving: a task analysis*. AI Magazine, Vol. 11, Issue 4 (Winter 1990) Pp. 59 – 71. Menlo Park, CA, USA
- [5] Corkill, D. 2003. *Collaborating software: blackboard and multi-agent systems & the future*. International Lisp Conference, New York, October 2003.
- [6] Diaz-Pace, A., Kim, H., Bass, L., Bianco, P., and Bachmann, F. 2008. *Integrating quality-attribute reasoning frameworks in the ArchE design assistant*. To appear in Proceedings QoSA 2008, University of Karlsruhe (TH), Germany, October 14-17, 2008.
- [7] Berdún, L., Díaz-Pace, A., Amandi, A., and Campo, M. 2008. *Assisting novice software designers by an expert designer agent*. In Expert Systems with Applications. Vol. 34(4): 2772-2782. ISSN 0957-4174, Elsevier.
- [8] Gero, JS and Kannengiesser, U. 2006. *A framework for situated design optimization*. In Innovations in Design Decision Support Systems in Architecture and Urban Planning, Springer, pp. 309-324.
- [9] Holland, A., O'Callaghan, B., and O'Sullivan, B. 2004. *Supporting constraint-aided conceptual design from first principles in Autodesk Inventor*. In Proceedings of IEA/AIE, Springer, LNCS, Canada.
- [10] Josephson, J., Chandrasekaran, B., Carroll, M., Iyer, N., Wasacz, B., Rizzoni, G., Li, Q., and Erb, D. 1998. *An architecture for exploring large design spaces*. Proceedings of the 10th Conference on AI. Madison, Wisconsin, United States. Pp: 143-150.
- [11] Maes, P. 1994. *Agents that reduce work and information overload*. Comm. of ACM 37 (7), Pp. 31-40. ACM Press
- [12] O'Sullivan, B. *Constraint-aided conceptual design*. 1999. PhD Thesis, Chapter 3, Department of Computer Science, University College Cork, Ireland.
- [13] Robbins, J., Hilbert, D., and Redmiles, D. 1996. *Extending design environments to software architecture design*. International Journal of Automated Software Engineering. Special issue: The Best of KBSE'96.

² IUCBRF web: <http://www.cs.indiana.edu/~sbogaert/CBR/index.html>