Empirical Validation of Component-based Software Systems Generation and Evaluation Approaches

Víctor Sagredo, Carlos Becerra
Universidad de Valparaíso
Valparaíso, Chile
victor.sagredo@alumnos.decom-uv.cl, carlos.becerra@uv.cl

Gonzalo Valdes
Universidad Técnica Federico Santa María
Valparaíso, Chile
gvaldes@inf.utfsm.cl

Abstract
Component-based software development needs to formalize a process of generation, evaluation and selection of Composite COTS-based Software Systems (CCSS), enabling software architects to make early decisions; the Azimut approach and its associated software tool were proposed to tackle this problem. This article presents an experimental study conducted to compare Azimut approach with a Systematized Ad-Hoc approach, regarding generated solutions quality, cost and effort. Results suggest that: (1) Azimut generate better quality solutions at lower cost, but not statistically significant, and (2) there is strong evidence showing that the effort required is higher than for Systematized Ad-Hoc approach; re-sampling methods (Bootstrap and Jackknife) were applied to reinforce these conclusions. Also this study serves as a framework for validating approaches, process and tools for generating and evaluating component-based software systems.

Keywords: Software Architecture; Component-Based Software Development; Composite COTS-based Software Systems; Experimental Software Engineering.

1 Introduction
Component-based software development (CBD) [14] proposes building systems by using preexisting components, to reduce development time, costs and risks, and to improve product quality; achieving these goals requires an adequate selection of components to reuse. In this process the key is component evaluation and selection [5, 12, 1]; therefore based on a series of components with similar functionalities which fulfills different requirements, the goal is to provide a model that helps architects in generating different Composite COTS-based Software Systems (CCSS) candidates.

Since the number of potential CCSS increases with the availability of many components, one of the objectives is to automate the exploration of possible candidates which fulfill certain requirements of satisfaction and cost. Component-based approaches to elaborate software must deal with the fact that in real settings, components information may be incomplete, imprecise and uncertain, and requirements may be likewise. In order to evaluate candidate architectures regarding requirements satisfaction, architects need to use whatever information is available about components, however imperfect. The Azimut approach [10, 11] has been proposed to solve this problem, using architectural abstraction levels, quality models for the problem and
solution domains, and several algorithmic techniques, all of which allow working with imperfect information to generate and select CCSS.

This paper presents the design, execution and results of an experimental study conducted to evaluate the Azimut approach for generation, evaluation and selection of components systems, comparing it to a Systematized Ad-Hoc approach. The main goal of this experiment is to validate the efficiency and effectiveness of the Azimut approach and its associated automated tool, regarding (1) quality and cost of the solutions generated, and (2) the effort needed for the generation of candidate CCSS. The study was conducted in an academic environment and re-sampling methods (Bootstrap and Jackknife) were applied to reenforce the conclusions.

Since, to our knowledge, there are no previous reported experiences of experimental studies conducted to validate approaches, processes and automated tools for generation and evaluation of CCSS, this work also serves as a framework to tackle such initiatives.

The article is structured as follows: Section 2 describes the Azimut approach definitions; Section 3 presents related work in empirical validation for CBD; Section 4 defines the approaches to evaluate in this study, Azimut and Systematized Ad-Hoc; Section 5 describes the experimental study design and execution; Section 6 presents the study results; Section 7 analyzes these results and suggests some new questions; Section 8 describes the threats to validity; and Section 9 summarizes and concludes.

2 The Azimut Approach

The construction of software systems using components offers great promise of reducing development times and costs while increasing quality, but its realization requires that architects are able to choose among alternative solutions composed from available components. A straightforward strategy could be to identify all possible component combinations, perhaps incorporating some technical matching restrictions, followed by their evaluation and comparison; yet this brute force approach is unfeasible due to four main issues:

1. It is sometimes quite complex to relate components (and sets thereof) to specific requirements, and specially to NFRs (non-functional requirements) due to their systemic nature.
2. In real situations, architects have at hand incomplete, imprecise and uncertain component information.
3. Resulting search spaces may be quite hard to explore systematically by humans.
4. Generation and evaluation of CCSS could be very time consuming.

The Azimut approach [10, 11, 2] (see Figure 1 for an overall depiction) aims to support architects in generating component assemblies for a given set of requirements, by using a derivation chain, in which defined stepping stones have a fundamental role. To characterize such stepping stones, Azimut uses a vocabulary taken from the distributed systems community [15], duly adapted to the software architecture context, e.g., policies, mechanism, components.

The main concepts involved in Azimut are:

- **Architectural Policy**: The first reification from NFRs (Non-Functional Requirements) to architectural concepts. Architectural policies can be characterized through specific concern dimensions that allow describing NFRs with more details (see Figure 2), e.g., in the communication concern it is possible to distinguish a synchrony dimension where it is possible to find asynchronous and synchronous policies. It is relevant to mention that there are policies sets for every given Architectural Concern, for which is possible to distinguish Architectural Dimensions. Concerns and dimensions allow to refine policies characterization using widely known software architecture constructs (e.g., communication-type, synchrony, topology). It is also possible to consider a higher abstraction level, considering that a concern is related to an Architectural Quality Attribute, e.g., the access control concern is related
to the security quality attribute. Azimut provides Policy Catalogs which gather platform-independent architectural policies, and store dimensions for each concern and policy (which have different values for each dimension).

- **Architectural Mechanism**: 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 requirements, e.g., SMTP (the standard e-mail protocol) mechanism supports asynchronous communication policy.

- **Component**: A software artifact that is executable, multiple-use, non-context specific, composable with other components, encapsulated (i.e., only accessible through their interfaces), and a unit of independent deployment and visioning [14]. E.g., the SendMail component provides SMTP mechanism, which as well supports synchronous policy.

Azimut approach needs as input: a problem description, which may be in natural language; a set of architectural abstractions to map from the problem domain to the solution domain (policies, mechanism, components); a set of catalogs with information about these abstractions and their relationships (i.e policy-, mechanism- and component catalogs). And produces as output: candidate CCSS that may solve the problem.

An automated tool was used to support humans in using Azimut:
• Azimut/Prolog: A Prolog-based tool to do the mapping between problem and solution domains, thus allowing users to generate and evaluate CCSS.

3 Empirical Studies on CCSS Generation and Evaluation

Empirical software engineering has established itself as a research area within software engineering during the last two decades. Basili et al. [3] published a methodological paper on experimentation in software engineering. Since then the use of empirical strategies in software engineering has grown considerably, although much work still remains to be done. Nowadays, empirical evaluations and validations are often expected in research papers. Motivations for undertaking such studies are summarized by Wohlin et al. [18] as “to get objective and statistically significant evidence regarding the understanding, controlling, prediction and improvement of software development”.

In software engineering, the steps required to perform experiments have been documented in a books dedicated to help software engineers in performing experiments, e.g., according to [18], the steps constituting the process of experimentation include: definition; planning; operation; analysis and interpretation; presentation and package. The representation of the experimental process in these steps manifest that experimentation is a formal and controlled activity.

The literature report some studies on the component-based software development area. Torchiano and Morisio [16] presented a study where people involved in industrial projects had to name key features in Component of-the-shelf (COTS) based development and to tell what they thought constitutes a COTS product; they presented the interview results in the form of six theses, which contradict widely accepted (or simply undisputed) ideas and presented a definition of “COTS product” that captures the key features: “A COTS product is a commercially available or open source piece of software that other software projects can reuse and integrate into their own products”. Besides defining COTS products, they derived from the interviews theses related to COTS-based development, e.g., Open source software is often used as closed source, Architecture is more important than requirements for product selection.

Li et al. [8] presented an empirical study on why project decision-makers select COTS instead of Open-Source Software (OSS) components, or vice versa. The study was performed as an international survey in Norway, Italy and Germany. They gathered answers from 83 projects using only COTS components and 44 projects using only OSS components. Results of the study showed significant differences and commonalities of integrating OSS or COTS components: (1) Both COTS and OSS components are used in projects with different application domains and different non-functional requirements, (2) The main expectation of using either COTS or OSS components is to obtain shorter time-to-market and less development effort, and (3) It is more difficult for COTS component users to follow requirement changes than OSS users.

Li et al. [9] also described and executed an exploratory study of state-of-the-practice of COTS-based development processes. 16 software projects in Norwegian IT companies were studied by structured interviews. The results showed that COTS-specific activities can be successfully incorporated in most traditional development processes (such as waterfall or prototyping), given proper guidelines to reduce risks and provide specific assistance.

In other work, Kvale et al. [7] investigated whether Aspect-Oriented Programming (AOP) can help to build an easy-to-change COTS-based system. A case study was performed to compare changeability between an object-oriented application and its aspect-oriented version. Results from this study showed that integrating COTS component using AOP may help to increase the changeability of COTS component-based systems, if the cross-cutting concerns in the glue-code are homogenous (i.e., consistent application of the same or very similar policy in multiple places).

These studies are focused on improving the component-based software development process, but none of them address automation of CCSS generation, evaluation and selection process; they mainly use qualitative process notions, aiming to describe and improve the processes involved, but not to automate them. Therefore
the study presented in this paper constitutes a pioneer attempt to formalize the evaluation of such processes through empirical software engineering methods.

4 Defining the Approaches to Study

In order to compare efficiency and effectiveness, we have to formalize the definition of the Azimut and Systematized Ad-Hoc approaches to generate a candidate architecture.

4.1 The Azimut Approach

The Azimut approach [10, 11, 2] needs as input:

- A catalog of policies, a catalog of mechanisms, and a catalog of components.

As support Azimut has an Prolog-based tool to do the mapping between problem and solution domains, which allows users to generate the CCSS.

The information contained in these catalogs is gathered from different world-wide sources, and is context-dependent, i.e., it depends on the kind of problem to solve. The specific problem and catalogs were given to subjects.

The Azimut process steps, shown in Figure 3, are summarized as follows:

- First get a general vision of the problem by reading and understanding the problem definition document.
- Then, identify a set of policies that represent the requirements (which are implicit in the definition document) problem. These policies must be selected from the policy catalog provided (by hand).
- Then, input these policies into Azimut/Prolog, which returns a set of candidates architectures.
- Finally, subjects rank the candidate solutions, select one of them, and draw an architectural diagram of it, using the DIA\(^1\) tool.

4.2 The Systematized Ad-Hoc Approach

In order to compare Azimut with a coherent ad-hoc approach, we need to formalize it, because if we use a chaotic ad-hoc approach (i.e., each subject searching for candidates architectures with no process), the comparison results between both approaches would be explained by this factor, and not due to the approaches

\(^1\)http://projects.gnome.org/dia/
actual efficiency and effectiveness. The aim of the Systematized Ad-Hoc approach is to generate an architectural design which identifies the subsystems control and communication infrastructure of the CCSS. The output of this design process is a software architecture description.

The Systematized Ad-Hoc process steps are summarized as follows:

- First get a general vision of the problem by reading and understanding the problem definition document (same as Azimut).
- Then, design a conceptual model of the problem.
- Then, determine the design goals involved, i.e., the requirements associated with the problem.
- Finally, identify subsystems, define strategies to build the system (hardware, software, persistence, access, etc.), and produce a system model that includes the decomposition into subsystems. A subsystem is characterized by the set of services it provides to other subsystems. The system architectural design focuses on defining the subsystems services involved. Within this step subsystems are assigned to available COTS components, and an architectural diagram of it is drawn, using the DIA tool.

5 Experimental Design

The experiment planning includes: context selection, hypotheses formulation, variables selection, subjects selection, experiment design and instrumentation [18]. This section also present training and execution design.

5.1 Hypotheses Formulation

The main hypothesis of the experiment is:

- It is possible to semi-automate the generation, evaluation and selection of CCSS using quality models at different levels of abstraction.

To test this primary hypothesis, the following operational, null and alternative hypotheses are defined:

- $H_{A0}$: The level of requirements satisfaction of the generated CCSS using Azimut is the same as using Ad-Hoc.
- $H_{Aa}$: The level of requirements satisfaction of the generated CCSS is different using Azimut compared to Ad-Hoc.
- $H_{B0}$: At the same given level of requirements satisfaction, the cost of the solutions generated using Azimut and Ad-Hoc is the same.
- $H_{Ba}$: At the same given level of requirements satisfaction, the cost of solutions generated using Azimut is different than the cost of solutions generated using Ad-Hoc.
- $H_{C0}$: At the same given level of requirements satisfaction, the effort required to generate solutions with Azimut is the same as with Ad-Hoc.
- $H_{Ca}$: At the same given level of requirements satisfaction, the effort required to generate solutions with Azimut and Ad-Hoc is different.
5.2 Variables and Metrics

The independent variables that can affect quality and cost of the generated CSS, and the effort required to generate them are:

**Approach:** The approach used for the generation, evaluation and selection of CCSS. This is the factor under study, with two levels: “Azimut approach” and “Systematized Ad-Hoc approach”.

**Subjects Experience:** Participants had theoretical knowledge on CBD, and they did not have prior experience using the Azimut approach and its support tool. This is an undesirable influence factor, and the chosen experimental design aimed to minimize its impact; also training sessions were provided for balancing participants knowledge and for guaranteeing process conformance during the experimental study.

In order to test the hypotheses, the dependent variables quality, cost and effort are defined and measured, as follows:

**Quality:** The level at which a generated CCSS meets the requirements and restrictions imposed for the construction of the system, which is:

\[
Sat_{R,C} = \sum_{c,c \in CCSS_{sol}} \sum_{r,r \in R} \mu(r,c)I_r.
\]

*Sat* is the level of requirements satisfaction, *CCSS*\(_{sol}\) is the CCSS generated as a solution, *R* is the set of requirements that the CCSS must meet, \(\mu(r,c)\) is the level at which the components satisfy a requirement (ranging from 0.0 to 1.0), and \(I_r\) is the importance of requirement (ranging from 0.0 to 1.0). The set of requirements that must be met and their importance dependent on the problem to solve, and was determine by the subjects. Whether a component satisfies a requirement or not is computed using the mechanism and component catalogs provided by Azimut. The *Sat* formula yields a measurement of requirements satisfaction, expressed as a percentage.

**Cost:** The total cost of generated CCSS, in $USD. This cost is based on the cost defined for the components available to composite the CCSS, which is:

\[
Cost_{CCSS_{sol}} = \sum_{c,c \in CCSS_{sol}} cost(c).
\]

**Effort:** The amount of person-hours needed to generate, evaluate and select a candidate CCSS, which satisfies the requirements and architectural restrictions specified.

5.3 Subjects Selection

The participants were 28 computer science undergraduate students with backgrounds in software architecture drawn, from two Chilean universities. None of them had previous experience with the Azimut approach.

5.4 Experiment Design

The experiment design has one factor design with two levels: the factor is the Approach and the two levels are the Azimut approach and the Systematized Ad-Hoc approach.

The same number of subjects (14) was assigned to each treatment. Each group applied just one treatment. Two steps were taken to assign subjects to treatments:

- Measuring experience: A survey was conducted to measure subjects knowledge and experience about software architecture; according to the results obtained, the subjects were classified as experts or novices.

- Treatment assignment: The subjects of the experts group were assigned randomly to each treatment, taking care that half of them were assigned to Azimut and the other half to Ad-Hoc. The same assignment method was used with the subjects of the novices group. This balanced assignment was
applied to prevent the influence of the experience factor in the final study results (e.g., if all experts were assigned to the Azimut group, better quality, lower cost and less effort might be explained by their experience rather than by of the treatment they used).

Figure 4 shows the treatment assignment process.

![Figure 4: Treatment assignment process](image)

5.5 Instrumentation

The following instruments were developed to help the subjects in applying the treatments:

- **Azimut approach instruments.**
  
  - **Problem Definition**: A document containing a description of the problem to be solved. The problem was designed to be completely solved within the time box allocated for the experimental activity (two hours, time calibrated during a previous pilot study).
  
    The problem consists of designing an intercommunicated distributed system for a company that needs to implement a client and sales management system for all of their offices throughout the country. It presents a description from which requirements and architectural policies can be inferred (e.g., fault tolerance, uptime, etc.).

  - **Policies Catalog**: Contains platform-independent architectural policies, which are structured within various dimensions and concerns (e.g., communication-type, synchrony, asynchronous).

  - **Mechanisms Catalog**: Give support to the participant for mapping policies to mechanisms. This catalog indicates various confidence degrees to determine whether a mechanism satisfies a given policy. The confidence degrees are defined as: “supports (1)”, “probably support (0.6)”, “probably does not support (0.3)”, “does not support (0)” and “do not know (empty)”. For example, consider the following (mechanism, policy, confidence degree) triples:
    
    smtp,asynchronous,1,
    smtp,1:m,1,
    smtp,push,1,
    smtp,integrity-over-timeliness,1,

  - **Components Catalog**: Indicates the mechanisms that are implemented by each component. The components catalog define five confidence degrees for the mechanism-component relationship: “supports (1)”, “it probably supports (0.6)”, “probably does not support (0.3)”, “does not support (0)” and “do not know (empty)”. Also indicates the available platforms on which the component can be used (Windows, Linux, Solaris); market variables, such as: producer, market share, supply reliability (valuated from 1 to 5, with higher values for high reliability); cost; integration costs; and support costs. For example, consider the following (component, mechanism, confidence degree) triples:
    
    SurgeMail,smtp,0.4,
SurgeMail,smtp-auth,0.8,
SurgeMail, pop3,0.2,
SurgeMail, pop3-auth,0.4,

- **Prolog-based Azimut tool:** Allows to automatically generate CCSS solutions.
- **Azimut How-to:** A document that describes the steps that should be executed to apply the Azimut approach.

- **Ad-Hoc Approach Instruments.**
  - **Problem Definition:** The same problem used for the Azimut approach.
  - **Systematized Ad-Hoc How-to:** A document that describes the steps that should be executed to apply the Systematized Ad-Hoc approach.

Other instruments used in this study are:

- **DIA tool,** surveys for participants (pre- and post-execution), training slides,
- **Data Analysis:** we used MATLAB. With this tool we applied Kolmogorov-Smirnov normality tests, one- and two-sided Mann-Whitney tests of difference between means, and the Bootstrap and Jakknife re-sampling techniques.

### 5.6 Training

In the Azimut training the approach was totally unknown to the subjects who participated in the experiment, and therefore it was necessary to deliver the knowledge so they could properly use this approach and understand each of the underlaying concepts. The training included a theoretical session, examples with the associated tool and exercises.

This training was similar to the one for Azimut approach, but with the Ad-Hoc approach.

### 5.7 Experiment Execution

Figure 5 shows the experiment execution plan.

![Figure 5: Experimental Execution](image)

The Azimut and Ad-Hoc training were performed by the same instructors the same day. The next day subjects of group I applied the Azimut approach on the Problem, and in parallel group II applied the Systematized Ad-Hoc approach on the same problem.

Azimut produces a ranked CCSS candidates set, but Ad-Hoc produces only a single CCSS candidate. Hence, the results comparison was performed between the top ranked Azimut result and the (single) Ad-Hoc result.
6 Experimental Results and Analysis

Table 1 presents the mean and standard deviation values for each metric and treatment applied.

The Kolmogorov-Smirnov test [17] rejected the assumption of normality for every data set, and also due to the small number of original data points we decided to apply two re-sampling techniques - Bootstrap and Jackknife-, as used and proposed to study other areas of software engineering, e.g., [4, 13].

The number of samples is a factor that indicates the error of the conclusions obtained after the analysis. To reduce this error re-sampling techniques were considered to increase the amount of data (from 28 to 280 data points), we used Bootstrap and Jackknife for this purpose. Bootstrap replicate $n$ times any of the data point, and Jackknife select a subset from the original data points and repeat this subset $m$ times.

The original and re-sampled data sets failed to support the assumption of normality, so we used a non-parametric test to compare means from two independent samples in all the cases. Table 2 shows the resulting p-values of the one-sided (effort) and two-sided (quality and cost) Mann-Whitney tests of difference between means for quality, cost and effort using the Azimut and Ad-Hoc approaches.

The quality measurements (measured in percentage of requirements satisfaction) showed no significant differences (all p-values $> 0.05$) for both approaches - Azimut and Ad-Hoc - in the three samples (i.e., the original data sample, and the two re-sampled data with Bootstrap and Jackknife).

To test the cost (measured in $\$USD$) and effort differences we used the same data sets, because there were no significant differences in the level of requirements satisfaction.

The null hypotheses related to cost also cannot be rejected in any of the samples (all p-values $> 0.05$).

The effort variable (measured in minutes) delivered statistically significant results for the three samples (all p-values $< 0.05$) providing sufficient evidence to reject its associated null hypothesis. So one-sided Mann-Whitney tests were applied, showing significantly that Azimut required more effort than Ad-Hoc to be applied (these values are shown in the third column in Table 2).

A less formal analysis can be made looking at the means presented in Table 1. The three samples show that quality was better with the Azimut approach, and also in the three samples the cost of the CCSS generated using Azimut was lower. Unfortunately none of these conjectures was shown statistically significant.

<table>
<thead>
<tr>
<th>Original Data</th>
<th>Quality [%]</th>
<th>Cost [$$USD$]</th>
<th>Effort [Minutes]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Azimut</td>
<td>72.6</td>
<td>335</td>
<td>100.6</td>
</tr>
<tr>
<td>Ad-Hoc</td>
<td>69.4</td>
<td>355</td>
<td>81.4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bootstrap</th>
<th>Quality [%]</th>
<th>Cost [$$USD$]</th>
<th>Effort [Minutes]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Azimut</td>
<td>71.7</td>
<td>342</td>
<td>101.0</td>
</tr>
<tr>
<td>Ad-Hoc</td>
<td>69.2</td>
<td>367</td>
<td>81.4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Jackknife</th>
<th>Quality [%]</th>
<th>Cost [$$USD$]</th>
<th>Effort [Minutes]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Azimut</td>
<td>72.6</td>
<td>335</td>
<td>100.6</td>
</tr>
<tr>
<td>Ad-Hoc</td>
<td>69.4</td>
<td>355</td>
<td>81.4</td>
</tr>
</tbody>
</table>

Table 1: Mean and Standard Deviations for each Metric
Table 2: p-values for each Metric

<table>
<thead>
<tr>
<th>Metric</th>
<th>Quality</th>
<th>Cost</th>
<th>Effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Data</td>
<td>0.645</td>
<td>0.644</td>
<td>0.038</td>
</tr>
<tr>
<td>Bootstrap</td>
<td>0.212</td>
<td>0.067</td>
<td>0.007</td>
</tr>
<tr>
<td>Jackknife</td>
<td>0.103</td>
<td>0.088</td>
<td>0.001</td>
</tr>
</tbody>
</table>

7 Discussion

The experiment could not prove that Azimut was better than Ad-Hoc regarding quality and cost, but the mean for the three samples suggest a tendency in that sense, so if the approach and tool are refined they might get better result in the future.

In a survey delivered to subjects right after the experimental activity, they provided several suggestions to improve the automated tool user interface usability. According to their experience, if these opinions are considered, the tool should become more helpful and easy to use. Besides, the fact that the effort was higher for Azimut, is aligned with these survey results.

This experimental study clearly serves as a framework to gather quantitative and qualitative data, feedback and insight for improving the approach and tool in future versions. It can also be used to compare it with other approaches and tools, such as DesCOTS [6].

8 Threats to Validity

The main threats to validity are related to the following factors:

- **Subjects history**: It was necessary to avoid the occurrence of events that could generate changes in the history of the subjects (between the experiment training and execution) in order to prevent threats to internal validity. Therefore, the period of time between these two stages was set to 1 day; also, it was emphasized that individuals should not study or refine the skills learned in the training to carry out the experiment.

- **Ad-Hoc systematization**: In order to evaluate Azimut versus a comparable ad-Hoc approach, we needed to formalize it, because if we used a chaotic ad-Hoc approach (i.e., subjects generating candidates architectures with no process), the final comparison between both approaches would be explained by this fact, and not due to the approaches actual performance. So we defined a simple process, called Systematized Ad-Hoc, which reflects the minimum practices needed to generate a CCSS, unsupported by any automatic tool. Even with these precautions, the depicted adaptations are threats to construct validity.

- **Metrics and problem size**: The problem and metrics must have the potential to make evident any difference between both approaches (if there is any). Therefore, the problem elaborated for this study was designed to be complex enough, so the solutions generated would be potentially different depending on the approach applied, being these difference detectable through the chosen metrics of quality, cost and effort. This fact was tested in a pilot study.

- **Academic Environment**: Participants were undergraduate students, so they are not representative of professional architects. Although some of them (mainly from the expert group) had professional experience, they were not full time practitioners yet, and this is a threat, to the external validity of the study. To mitigate this threat an extensive training in Azimut, Systematize Ad-Hoc and software architecture was provided.
9 Conclusions and Future Work

The Azimut approach aims to support architects in generating component assemblies for a given set of requirements, by using a derivation chain, in which defined stepping stones have a fundamental role. It uses as input: 1) A problem description, which may be in natural language; 2) A set of architectural abstractions to map from the problem domain to the solution domain (policies, mechanism, components); and 3) A set of catalogs that have information about these abstractions and their relationships (policies, mechanism and components catalogs). It is supported by a software tool, and produces as output candidate architectures that may solve the problem.

An experimental study was conduced to compare the Azimut approach with a Systematized Ad-Hoc approach, regarding generated solutions quality and cost, and effort implied in applying each approach. Results suggest that Azimut generated better quality solutions at lower cost, although not statistically significant in the three samples, but also there is strong statistically significant evidence showing that the effort required is higher than for Ad-Hoc; re-sampling methods (Bootstrap and Jackknife) were applied to reinforce these conclusions and yielded the same results. Results concerning effort are aligned with a post-experiment survey answered by the participants, where they suggest various ideas for improving the usability of Azimut tool user interface.

Previous studies in the field of CCSS generation and evaluation were focused on improving the component-based software development process, but none of them focused on the automation approach. Therefore this work serves as a framework to tackle such initiatives, and currently we plan to refine it and use it to validate a new version of the Azimut tool, and to compare it with other tools an approaches.

References


