EINDHOVEN UNIVERSITY OF TECHNOLOGY

Software architecture analysis tool : software architecture metrics collection

Citation for published version (APA):
Muskens, J., Chaudron, M. R. V., & Westgeest, R. (2002). Software architecture analysis tool : software architecture metrics collection. In *Proceedings 3rd PROGRESS Workshop on Embedded Systems (Utrecht, The Netherlands, October 24, 2002)* (pp. 128-139). STW Technology Foundation.

Document status and date:
Published: 01/01/2002

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 05. Dec. 2021

Software Architecture Analysis Tool

Software Architecture Metrics Collection

Johan Muskens, Michel Chaudron and Rob Westgeest
Technische Universiteit Eindhoven and CMG Eindhoven
P.O. Box 513, 5600 MB Eindhoven, the Netherlands
Phone: +31 (0)40 2472993 Fax: +31 (0)40 2436685
E-mail: J.Muskens@tue.nl

Abstract— **The Software Engineering discipline lacks the ability to evaluate software architectures. Here we describe a tool for software architecture analysis that is based on metrics. Metrics can be used to detect possible problems and bottlenecks in software architectures. Even though metrics do not give a complete evaluation of software architectures it is a useful analysis method. The Software Architecture Analysis tool can be applied to XMI output generated by a UML modelling tool. We have done this using Rational Rose.**

Keywords— **software architecture, analysis, metric collection**

I. INTRODUCTION

The first step in making good software is making a good design. The design defines the architecture of the software to be built. The quality of the software highly depends on the architecture defined in the early stages of he development process.

The architecture can influence the functional requirements as well as the non-functional requirements. The impact of architectural design decisions in a software development process is very high.

At the present time there are a few methods to evaluate software architectures. SAAM [3] and ATAM [3] are by far the most well known. These methods are evaluation techniques of quality attributes of software architectures by a group of experts.

Experts are often not available, these methods are not repeatable, time consuming, and subjective.

We analysed software architectures based on the 4+1 View Model [14] and describe software architecture analysis in a way that is implemented by the software architecture analysis tool. This tool calculates metrics; these metrics can help architects evaluating software architectures.

Section 2 discusses the notion of software architectures that we use. *Section 3* discusses what can be analysed of software architectures and how software architectures can be analysed. *Section 4* presents the software architecture description model used as input for the software architecture analysis tool. *Section 5* presents a data model for the architecture description models described in section 4. *Section 6* discusses software metrics based on the data model described in section 5. *Section 7* concerns the interpretation of metrics. *Section 8* describes the environment of the tool. *Section 9* describes the design of the tool.

II. WHAT IS SOFTWARE ARCHITECTURE?

There are several definitions of software architectures, almost as many as there are software architects. Here are some examples.

(Bass, Clements and Kazman. Software Architecture in Practice, Addison-Wesley 1997)
The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.
By "externally visible" properties, we are referring to those assumptions other components can make of a component, such as its provided services, performance characteristics, fault handling, shared resource usage, and so on. The intent of this definition is that a software architecture must abstract away some information from the system (otherwise there is no point looking at the

© PROGRESS/STW 2002, ISBN 90-73461-34-0

architecture, we are simply viewing the entire system) and yet provide enough information to be a basis for analysis, decision making, and hence risk reduction.

(Booch, Rumbaugh, and Jacobson, 1999)
An architecture is the set of significant decisions about the organization of a software system, the selection of the structural elements and their interfaces by which the system is composed, together with their behaviour as specified in the collaborations among those elements, the composition of these structural and behavioural elements into progressively larger subsystems, and the architectural style that guides this organization --- these elements and their interfaces, their collaborations, and their composition.

For the analysis of software architectures the definition of software architectures is not the most important thing. The notation used to describe the architectures is more important. We use a notation based on "The '4+1' View Model of Software Architecture" [14] by Philippe Kruchten, which is described in section 4.

III. Software Architecture Analysis

Software architecture can be analysed on functional requirements as well as quality requirements. We consider the analysis of software architectures on non-functional quality attributes like extendibility, maintainability,

scalability, reusability, etc by means of an analysis tool.

Software architectures can be analysed on several aspects. These aspects can be divided in the following categories:

<u>What can we analyse:</u>

- Structural: The structural aspects of software architectures are things like the logical decomposition of the system in components and the distribution of services over the component.
- Behavioural: The behavioural aspects concern the ordering & multiplicity of actions and how the components work together.
- Semantical: Semantical aspects are usually not described in any kind of diagram, but concern the meaning / interpretation of the software architecture description.

For automated analysis there are several approaches.

<u>How can we analyse:</u>

- Metrics: Software metrics concern the calculation of scores for elements (for example components) in software architectures. Metrics can give useful information on architectures, however the interpretation of metrics can be quite complicated. Currently most software metrics are based on code [2,4,17].
- Conformance to design patterns or architectural styles: Patterns and styles are often used in software architecture designs. They represent a simple concept and have proven their value over time. Therefore it can be useful to check whether architectures conform to a specific style or pattern [5].
- Detection of Bottlenecks and design critiques: Bottlenecks and design critiques are typically things an architect wants to avoid in his design. A tool that points the bottlenecks / anti-patterns in a design can be useful [6].

This paper considers analysis of the structural, behavioural, and in some way also some semantical aspects of software architectures by means of metrics.

IV. "The '4+1' View Model of Software Architecture"

Software architecture deals with abstraction, with decomposition and composition, with style and esthetics. To describe software architectures we use "The '4+1' View Model of Software Architecture" by Philippe Kruchten [14]. This model is composed of multiple views or perspectives:

- *Logical view*
- *Process view*
- *Physical view*
- *Development view*

The description of architectures can be organised around these four views, and then illustrated by scenarios that become the fifth view.

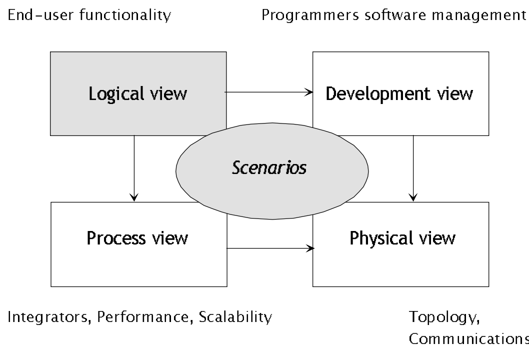

Figure 1 - The '4+1' View Model

We will now look at the views in turn. Only the "Logical view" and the "Scenarios" are analysed by the analysis tool.

A. Logical View

The logical architecture primarily supports the functional requirements – what the system should provide in terms of services to its users. The system is decomposed into a set of key abstractions, taken (mostly) from the problem domain, in the form of components. Besides the functional decomposition in components the logical view shows the logical dependencies of the components.

For the logical view we use a component diagram. The component diagram consists of the following elements:

- *Components*: A black box with an interface. This interface is a list of services which the component provides to the out-side world.
- *Relations*: A component can use one or more services of another component; the uses relation models this.
- *Interface descriptions*: Describe the provided services to the out-side world (name and parameter list)

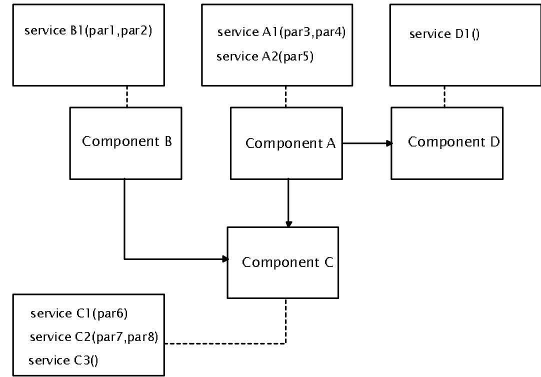

Figure 2 - Example of component diagram

In addition to the decomposition in components we describe the behaviour of the components in a state-transition diagram, which has the following elements:

- *State*: Show the different states of a component.
- *State-transition*: Show the allowed state changes of a component.
- *Begin state indication*: Indicates the initial state of a component

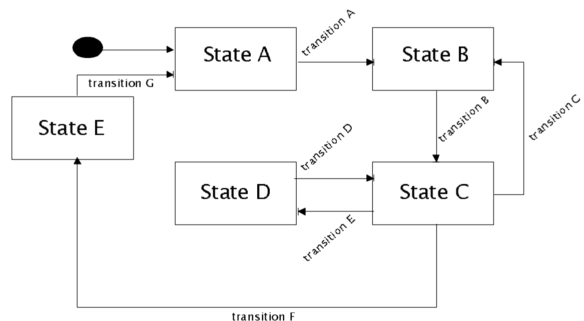

Figure 3 - Example of state-transition diagram

B. Process View

The process view takes some non-functional requirements into account, such as performance and availability. It addresses issues of concurrency and distribution, of system integrity, of fault-tolerance, and how the main abstractions from the logical view fit in the process architecture.

C. Development View

The development view focuses on the actual software module organisation in the software development environment. The software is packaged in small chunks –

program libraries, or subsystems – that can be developed by one or a small number of developers. The subsystems are organized in a hierarchy of layers, each layer providing a narrow and well-defined interface to the layers above.

D. Physical View

The physical view primarily takes into account the non-functional requirements of the system such as availability, reliability, performance and scalability. The software executes on a network of computers, or processing nodes. The various elements identified – networks, processes, tasks and objects – need to be mapped onto the various nodes. Several different physical configurations can be used: some for development and testing, others for the deployment of the system for various sites or for different customers. The mapping of the software to the nodes therefore needs to be highly flexible and have minimal impact on the source code itself.

E. Scenarios

The elements in the four views are shown to work together seamlessly by the use of a small set of important scenarios – instances of more general use cases. The scenarios are in some sense an abstraction of the most important requirements.

We describe a scenario by means of a message sequence diagram. These diagrams contain the following elements:

- *Components*
- *Service calls*

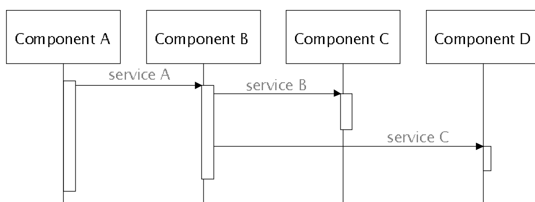

Figure 4 - Example of scenario

Normally there are several scenarios. Each scenario is an instance of a more generic use case. The use cases are shown in a use case diagram.

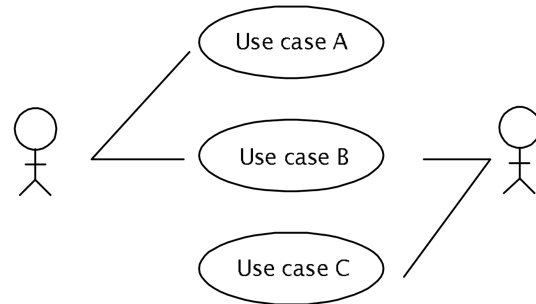

Figure 5 -Example of use case diagram

The relation between a scenario and a use case is not explicitly shown in a diagram.

V. Data Model

The data model used by the software architecture analysis tool is restricted to the following diagrams (logical view and scenarios).

- *Use case diagram*
- *Sequence diagrams*
- *Component diagram*
- *State-transition diagrams*

These diagrams are related. A use case diagram is linked to several scenarios, a sequence diagram contains components from the component diagram and the state-transition diagram shows the states and state-transitions of a component from the component diagram.

In order to be able to analyse these diagrams we distilled an abstract data model from the diagrams. It is obvious that the diagrams contain more information, but the distilled information is sufficient for the calculation of a lot of interesting metrics. It is likely that for the calculation of new metrics the following data selection needs to be extended.

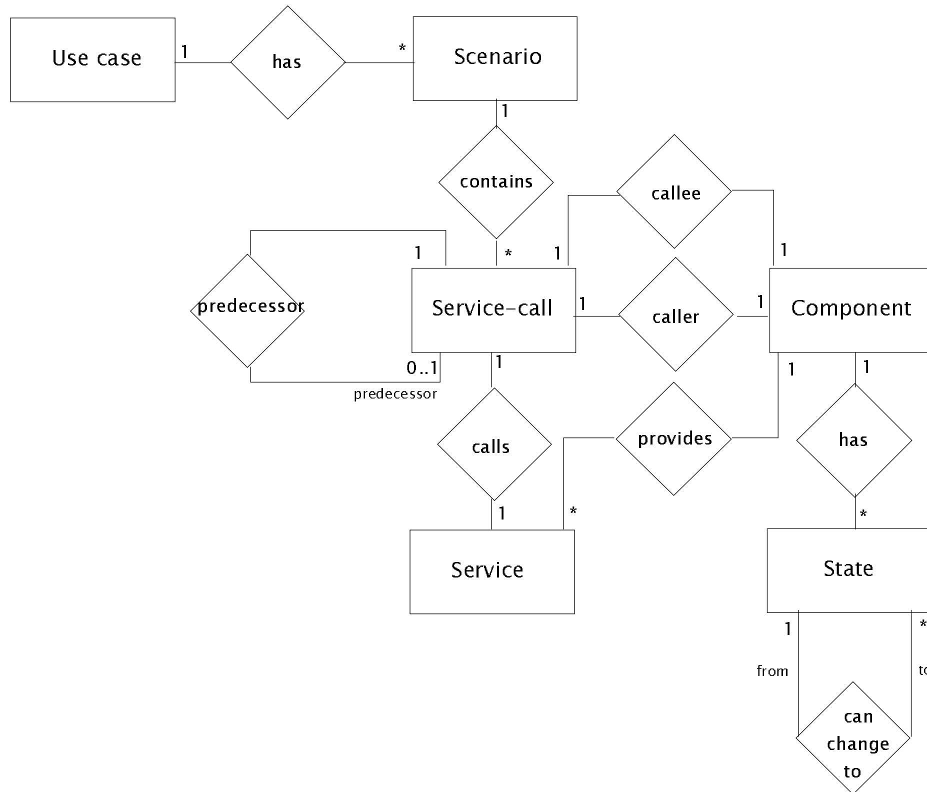

Figure 6 – ER diagram

- *Set of all use cases*
- *Set of all scenarios*
- *Set of all components*
- *Set of all services*
- *Set of all states*
- *Set of all service calls*

The elements of these sets are related. For example a use case has several scenarios, a scenario contains several service calls and a service call has a caller, callee, predecessor and a service that is called. This is modelled in the Entity-Relation diagram in figure 6.

The Entity-Relation diagram in figure 6 can be translated in to tables. This results in the following tables.

Table U contains the set of all use cases. For each use case the NAME and an ID are stored. The NAME is a unique string to identify the use case.

U	
ID	CHAR
NAME	CHAR

Table S contains the set of all scenarios. For each scenario the NAME and an ID are stored. The NAME is the name of the sequence diagram describing the scenario. The ID is a unique string to identify the scenario.

S	
ID	CHAR
NAME	CHAR

Table C contains the set of all components. For each component the NAME and an ID is stored. The NAME is then name the component has in the component diagram. The ID is a unique string to identify the component.

C	
ID	CHAR
NAME	CHAR

Table M contains the set of all services. For each service the NAME and an ID is stored. The NAME is the name of the service (as described in the component diagram). The ID is a unique string to identify the service.

M	
ID	CHAR
NAME	CHAR

Table T contains the set of all states. For each state the NAME and an ID is stored. The NAME is the name of the state (as described in one of the state-transition diagrams). The ID is a unique string to identify the state.

T	
ID	CHAR
NAME	CHAR

Table US is used to store the ("has") relation between a use case and a scenario. The table has two fields. ID_U identifies the use case and ID_S identifies the scenario.

US	
ID_U	CHAR
ID_S	CHAR

Table CM is used to store the ("provides") relation between a component and a service. The table has two fields. ID_C identifies the component and ID_M identifies the service.

CM	
ID_C	CHAR
ID_M	CHAR

Table SM contains the set of all service calls. For each service call an id, scenario, caller (component), callee (component) and a service are stored. In some cases a predecessor is stored. The table has six fields. ID is a unique string identifying the service call, ID_S identifies the context (scenario) in which the call is made, ID_C1 identifies the caller of the service, ID_C2 identifies the callee, ID_M2 identifies the called service and ID_PRED identifies the service call preceding this call in the specific scenario (if available).

SM	
ID	CHAR
ID_S	CHAR
ID_C1	CHAR
ID_C2	CHAR
ID_M2	CHAR
ID_PRED	CHAR

Table CT is used to store the ("has") relation between a component and a state. The table has two fields. ID_C identifies the component and ID_T identifies the state.

CT	
ID_C	CHAR
ID_T	CHAR

Table TT is used to store the ("can change to") relation between two states. The table has two fields. ID_T1 identifies the "from" state and ID_T2 identifies the "to" state.

TT	
ID_T1	CHAR
ID_T2	CHAR

Implementing the metric calculation by means of SQL statements has several benefits:

- *Extendibility*: Adding and removing metrics to the analysis tool comes down to executing an extra SQL statement or removing one.
- *Easy implementation*: Implementing the calculation of the metrics comes down to the execution of an SQL statement. Execution of SQL statements is possible in many database management systems. One of these database management systems can be used for the implementation of the metric calculation.

VI. METRICS

The software architecture analysis tool calculates several metrics. For software metrics there are several guidelines. Metrics must have the following properties:

- *Simple and computable* – the metrics should be easy to learn and use.
- *Empirically convincing* – they should satisfy the expectations of the engineer.
- *Consistent and objective* – they should produce unambiguous results.
- *Consistent in dimensionality* – they should be mathematically reasonable.
- *Language independent*
- *Facilitating feedback* – they should provide useful information for software improvement.

Metrics usually are sizes of selections or a simple function based on several selection sizes, which means that the metrics are queries and can be implemented in a database query language like SQL.

The next sub-sections discusses some example metrics.

A. Coupling

A component uses services of other components; this means the component is dependent on the other component.

This metric counts the number of components of which a service is called. The main thought behind this metric is that dependence on many different components id bad for reusability, extendibility and maintainability, because:

Reuse of a specific component then requires reuse of a large number of components.
Maintenance of a specific component then requires knowledge of a large number of other components.

Definition:

$$(\#\, c \in C :\ (\exists\, sm \in SM : sm.id_c1 = a.id \wedge sm.id_c2 = c.id))$$
$$\text{for all } a \in C$$

Query, using the abstract data model described in section 5:

```
SELECT C.NAME, COUNT(DISTINCT SM.ID_C2)
FROM C, SM
WHERE (C.ID = SM.ID_C1) GROUP BY C.ID;
```

B. Cohesion

When a component is used for the implementation of several use cases it is likely that the component implements requirements that are logically unrelated. It usually means that the cohesion between the different services of the component is low.

This metric counts the number of use cases that contain a scenario in which a service of a specific component is called or in which that component calls a service. The main thought behind this metric is that cohesion within a component should be high (number of use cases per component should be low). High cohesion within a component is good for maintainability, because this means the component implements logical dependent functionality.

Definition:

$$(\#\, u \in U :\ (\exists\, sm \in SM : (sm.id_c1 = a.id \vee sm.id_c2 = a.id) \wedge (\langle u.id, sm.id_s \rangle \in US)))$$
$$\text{for all } a \in C$$

Query, using the abstract data model described in section 5:

```
SELECT C.NAME, COUNT(DISTINCT US.ID_U)
FROM C, US, SM
WHERE ((C.ID = SM.ID_C1 OR C.ID = SM.ID_C2)
AND (SM.ID_S = US.ID_S)) GROUP BY C.ID;
```

C. Complexity of Services

This metric attempts to give an indication of the average complexity of the services of a component. A component has a number of states and transitions between these states. Service executions are responsible for state changes (transitions). Therefore we presume that the services of a component are more complex when they are responsible for a larger number of state changes.

This metric computes the average number of state transitions per service for a component. Main thought behind this metric is that complexity of components /

services is bad for maintainability and extendibility.

Definition:

(# tt ∈ TT : <a.id,tt.id_t1> ∈ CT) / (# cm ∈ CM: cm.id_c = a.id)
for all a ∈ C

Query, using the abstract data model described in section 5:

```
SELECT C.NAME, COUNT(DISTINCT TT.ID_T1,TT.ID_T2)
               / COUNT(DISTINCT CM.ID_M)
FROM C, CM, CT, TT
WHERE (C.ID = CM.ID_C AND C.ID = CT.ID_C
               AND CT.ID_T = TT.ID_T1)
               GROUP BY C.ID;
```

D. Number of Services of Component

A component provides services. It is wise to distribute functionality evenly over the design. Large differences in the number of services per component can five an indication that this is not the case.

This metric counts the number of provided services of a specific component. The main thought behind this metric is that a well-balanced distribution of functionality over the design is good for extendibility and maintainability, because excessively large components are difficult to understand.

Definition:

(# m ∈ M : (∃ cm ∈ CM : cm.id_m = m.id ∧ cm.id_c = a.id))
for all a ∈ C

Query, using the abstract data model described in section 5:

```
SELECT C.NAME, COUNT(DISTINCT CM.ID_M)
FROM C, SM
WHERE (C.ID = CM.ID_C) GROUP BY C.ID;
```

E. Fan in

Tasks should be distributed as equally as possible. When the number of called services of a component is high this can give an indication that the component is a possible bottleneck considering scalability. It also indicates that dependency of other components on the specific component is high.

This metric counts the number of called services of a specific component for all scenarios.

Definition:

(# sm ∈ SM : sm.id_c2 = a.id)
for all a ∈ C

Query, using the abstract data model described in section 5:

```
SELECT C.NAME, COUNT(*)
FROM C, SM
WHERE (C.ID = SM.ID_C2) GROUP BY C.ID;
```

F. Fan out

A component usually uses services of other components. This means that the component is dependent on the other component. The dependence on other components increases with the number of service calls of a component.

This metric counts the number of service calls of a specific component for all scenarios. Main thought behind this metric is that dependencies are bad or reusability and maintainability, because:

- Reuse of a specific component then requires reuse of a large number of components.
- Maintenance of a specific component then requires knowledge of a large number of other components.

Definition:

(# sm ∈ SM : sm.id_c1 = a.id)
for all a ∈ C

Query, using the abstract data model described in section 5:

```
SELECT C.NAME, COUNT(*)
FROM C, SM
WHERE (C.ID = SM.ID_C1) GROUP BY C.ID;
```

G. Depth of Scenario

Keep software architectures as simple as possible. Simplicity is good for maintainability and reusability. This metric gives an indication of the complexity of a scenario. It measures how deep the service calls are nested for a scenario. If the depth of a scenario is to high

this is bad for the understandability and therefore also bad for maintainability and adaptability. Note that this metric does not necessarily indicate the complexity of an architecture, it indicates the complexity of some of the diagrams used to describe the architecture.

Definition:

(# c ∈ C : (∃ sm ∈ SM : sm.id_c1 = c.id ∧ sm.id_s = a.id))
for all a ∈ S

Query, using the abstract data model described in section 5:

```
SELECT C.NAME, COUNT(DISTINCT SM.ID_C1)
FROM S, SM
WHERE (S.ID = SM.ID_S) GROUP BY S.ID;
```

VII. How Do We Interpret the Results?

The metrics described in the previous section give values for certain elements (use cases, scenarios or components). The result of the query is a table containing two columns. Each record has an element description (first column) and a value (second column). What can we do with this table?

We use the following approach. We do not use benchmark values telling whether a score of an element is good or bad, but we compare the score of an element with the score of the other elements within the design. We look for the elements that have outlying values, because we suspect these elements to be the problem elements. Outlying values are values that differ more than 2 times the standard deviation from the mean value. The standard deviation is calculated as follows:

First calculate the variance σ2.

$$\sigma^2 = \frac{\Sigma(X - \mu)^2}{N}$$

Were μ is the mean value of the distribution (all scores of the elements) and N is the number of scores. The standard deviation is the square root of the variance. It is the most common used measure of spread.

Consider the following example of output of a metric.

Component A	2
Component B	3
Component C	3
Component D	2
Component E	1
Component F	**15**
Component G	2
Component H	3

```
Average: 3.875
Standard deviation: 4.26

Conclusion: Component F has an outlying value and is
a possible problem element.
```

Conclusion: Component F has an outlying value and is a possible problem element.

Note that not all outlying scores are the result of a design error. It is up to the architect to judge whether an element is a real problem and redesign is necessary.

VIII. Environment of the Analysis Tool

The Software Architecture Analysis Tool analyses architectures created with Rational Rose (UML modelling tool). Rational Rose is used for the following tasks.

- Creation of the software architectures
- Export of the software architectures to an interchange file (XMI).

Input for the tool is the interchange file created by Rational Rose. The tool has one specific task: the creation of an analysis report. But before this report can be created the architecture has to be stored in a database and analysed.

The output of the tool is a report in HTML format. This report can be read with an ordinary HTML browser.

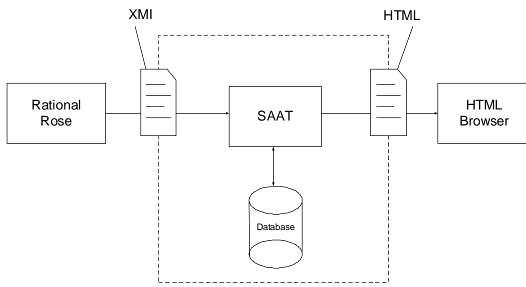

Figure 7 - Environment of Software Architecture Analasys tool

IX. DESIGN OF THE ANALYSIS TOOL

The tool consists of several components working together. Each component has its own responsibilities. The components are:

- *Parser*: This component extracts the relevant architecture information from the input file. The input file is an .xmi file generated by Rational Rose.
- *Database creator*: This component creates a new database with the database management tool (mySQL) and creates the empty tables in this database.
- *Database filler*: This component fills the database with the software architecture information extracted from the .xmi file.
- *Database checker*: This component checks the database for incomplete information.
- *Analyser*: This component executes the queries that are the actual architecture analysis.
- *Statistics calculator*: This component calculates some statistics on the results of the analysis.
- *Statistic filter*: This component filters the result based on the statistics calculated by the statistic calculator such that only the elements with the outlying values remain.
- *Saat*: This is a control component that is used to configure the Software Architecture Analysis Tool.

The component diagram in figure 8 illustrates this. The scenario in figure 9 illustrates how the components work together.

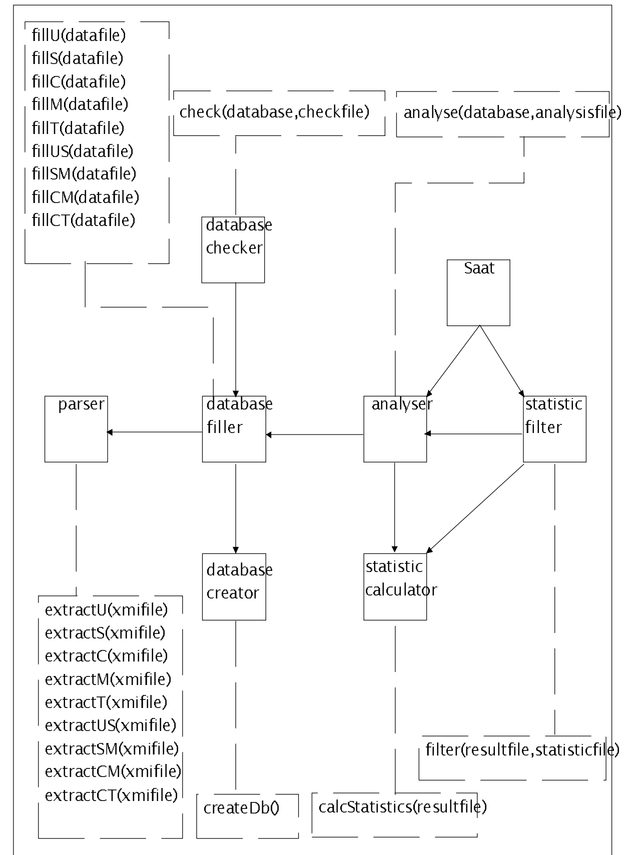

Figure 8 - Component diagram

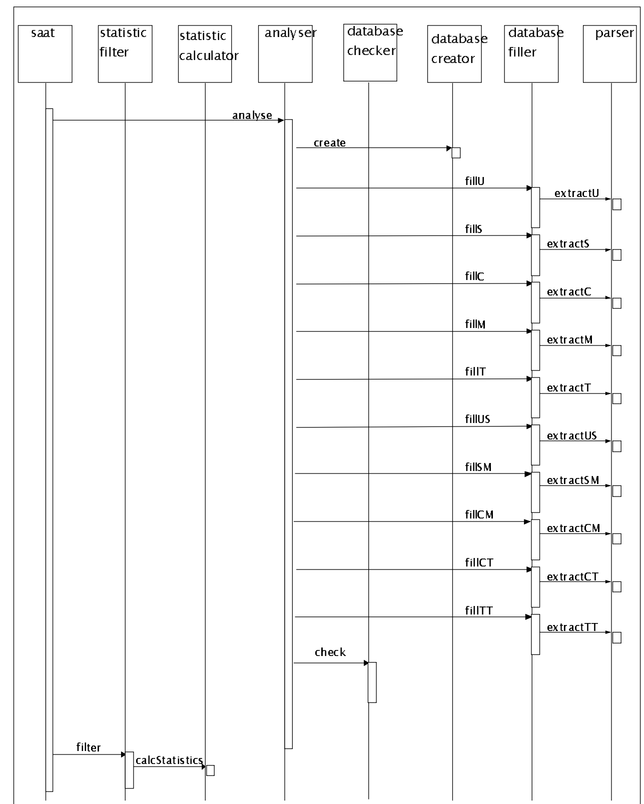

Figure 9 - Scenario

X. RELATED WORK

MAISA is a research and development project aiming at developing methods for the measurement of software quality at the design level. The metrics are computed from the system's architectural description, predicting the quality attributes of the system derived from it. Most notably, size and performance metrics are addressed. The performance analysis is refined by analysis at code level.

XI. CONCLUSION

Analysing software architectures is a complicated task. Several methods have been designed to evaluate software architectures. SAAM and ATAM are the most well known. The most important thing those methods have in common is that they use experienced architects for the evaluation of a design.

Metrics can be calculated by a tool, without the help of an architect. However the results should be interpreted by an architect and the architect can take advantage of the information given by the metrics.

The metrics do not tell whether architectures are good or bad, but it helps the architect in improving his design by indicating possible problem elements.

The Software Architecture Analysis Tool calculates an arbitrary selection of metrics. The use of SQL for the implementation of the metrics makes it easy to extend the tool with new metrics and tune the old ones.

XII. ACKNOWLEDGMENTS

The Software Architecture Analysis Tool is developed during a period of 9 months at CMG Eindhoven. During this period there were several people that have offered useful advice particularly I would like to mention Gert Florijn, Andre Postma, Bjorn Bon and Onno van Roosmalen.

This work profited from several test cases provided by Ronald Pulleman, Onno van Roosmalen and Philips ASA lab.

REFERENCES

[1] Object Oriented Design Heuristics, Arthur J. Riel ,ISBN 0-201-63385-X (1996)

[2] Object Oriented Software Metrics, Mark Lorenz and Jeff Kidd, ISBN 0-13-179292 (1994)

[3] Software Architecture in Practice, Len Bass Paul, Clements and Rick Kazman, ISBN 0-201-19930-0 (1998)

[4] Software Metrics (A Rigorous & Practical Approach), Norman E. Fenton and Shari Lawrence Pfleeger, ISBN 0-534-95600-9 (1996)

[5] Analysis patterns: reusable object model, Martin Fowler, ISBN: 0-201-89542-0(1997)

[6] AntiPatterns: refactoring software, architectures and projects in crisis, William J. Brown, Raphael C. Malveau and Hays W. McCormick III, ISBN 0-471-19713-0 (1998)

[7] Attribute-Based Architecture Styles; Mark H. Klein, Rick Kazman, Len Bass, Jeromy Carriere, Mario Barbacci and Howard Lipson (1999)

[8] Staan op schouders van reuzen, Bjorn Bon

[9] A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems; Mary Shaw and Paul Clements (1996)

[10] Metrification of a software architecture definition; F.W. Greuter

[11] An Architectural Connectivity Metric and Its Support for Incremental Re-architecting of Large Legace Systems; Reinder J. Bril and Andre Postma (2000)

[12] The Koala Component Model for Consumer Electronics Software; Rob van Ommering, Frank van der Linden, Jeff Kramer and Jeff Magee (2000)

[13] Software Architecture Documentation in Practice: Documenting Architectural Layers; Felix Bachmann, Len Bass, Jeromy Carriere, Paul Clements, David Garlan, James Ivers, Robert Nord and Reed Little (2000)

[14] Architectural Blueprints -- The 4+1 View Model of Software Architecture; Philippe Kruchten (1995)

[15] Principles for Evaluating the Quality Attributes of a Software Architecture; Mario R. Barbacci, Mark H. Klein and Charles B. Weinstock (1997)

[16] A Survey of Architecture Description Languages; Paul C. Clements (1996)

[17] Software Metrics: Roadmap; Norman E. Fenton and Martin Neil (2000)

[18] Scripting Coordination Styles; Franz Achermann, Stefan Kneubuehl and Oscar Nierstrasz (2000)

[19] Applying Relation Partition Algebra for Reverse Architecting; Andre Postma and Marc Stroucken (1999)

[20] A Two-phase Process for Software Architecture Improvement; Rene Krikhaar, Andre Postma, Alex Sellink, Marc Stroucken and Chris Verhoef (1999)

[21] Maisa, Jukka Paakki, Inkeri Verkamo, Juha Gustafsson, Lilli Nenonen (1999 – 2001) (http://www.cs.helsinki.fi/group/maisa/)

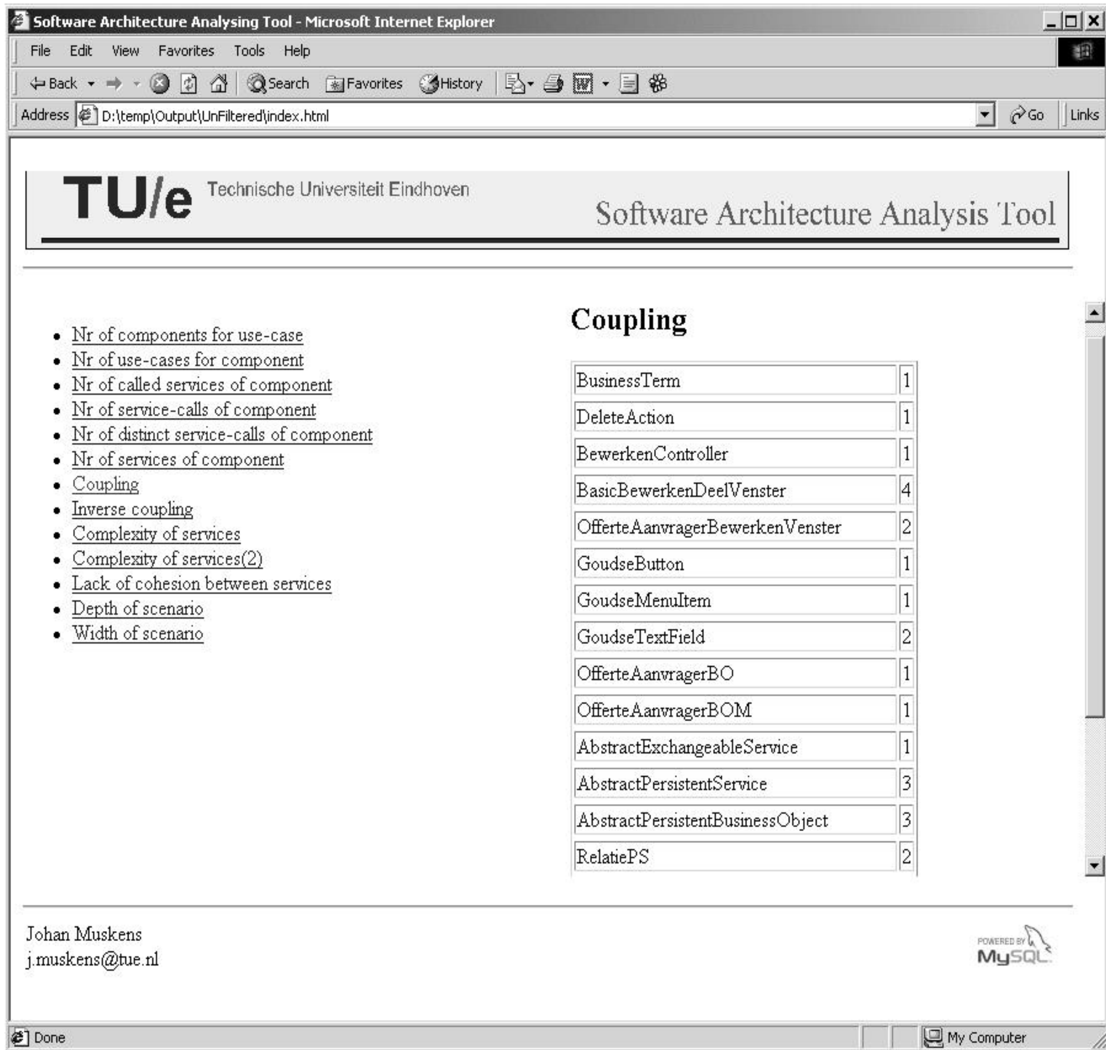

Figure 10 - Screenshot of Software Architecture Analysis Tool