



**Universidade de Brasília**

Instituto de Ciências Exatas  
Departamento de Ciência da Computação

# **An Architecture Conformance Process for Software Ecosystems with Heterogeneous Languages**

Sigfredo Farias Rocha

Dissertação apresentada como requisito parcial para conclusão do  
Mestrado Profissional em Computação Aplicada

Orientadora  
Profa. Dra. Genáina Nunes Rodrigues

Brasília  
2017

Ficha catalográfica elaborada automaticamente,  
com os dados fornecidos pelo(a) autor(a)

Ra Rocha, Sigfredo Farias  
An Architecture Conformance Process for Software  
Ecosystems with Heterogeneous Languages / Sigfredo Farias  
Rocha; orientador Genáina Nunes Rodrigues. -- Brasília, 2017.  
61 p.

Dissertação (Mestrado - Mestrado Profissional em  
Computação Aplicada) -- Universidade de Brasília, 2017.

1. Conformidade Arquitetural. 2. Modelos de Reflexão. I.  
Rodrigues, Genáina Nunes, orient. II. Título.



**Universidade de Brasília**

Instituto de Ciências Exatas  
Departamento de Ciência da Computação

# **An Architecture Conformance Process for Software Ecosystems with Heterogeneous Languages**

Sigfredo Farias Rocha

Dissertação apresentada como requisito parcial para conclusão do  
Mestrado Profissional em Computação Aplicada

Profa. Dra. Genáina Nunes Rodrigues (Orientadora)  
CIC/UnB

Prof. Dr. Ricardo Terra Nunes Bueno Villela    Prof. Dr. Rodrigo Bonifácio de Almeida  
DCC/UFLA    CIC/UnB

Prof. Dr. Marcelo Ladeira  
Coordenador do Programa de Pós-graduação em Computação Aplicada

Brasília, 26 de setembro de 2017

# Dedicatória

Dedico esse trabalho aos meus pais, à minha amada esposa Jeissi e aos meus cachorros Scott e Lucy.

# Agradecimentos

À UnB pela oportunidade.

Aos meus superiores do Centro de Informática (antigos e atuais): Jacir Bordim, Marcelo Ladeira, Jorge Fernandes, Consuelo Martins e Riane Torres por todo apoio dado.

À minha orientadora Genáina Rodrigues que guiou com paciência e incentivo o meu crescimento nessa jornada.

Aos professores Rodrigo Bonifácio e Ricardo Terra cujas críticas possibilitaram a melhoria do meu trabalho.

Ao colega Renato Edésio pela contribuição em vários momentos. Pelo apoio, dicas e feedback.

Aos meus colegas do PPCA que se alegraram e sofreram junto comigo nessa jornada: Renan, Fábio, Alan, Alysson e Reinaldo.

# Resumo

Os custos de manutenção de software são fortemente influenciados pela sua conformidade com a arquitetura idealizada e boas práticas de desenvolvimento. A falta de conformidade gera gastos desnecessários. Ambientes heterogêneos com diferentes plataformas de desenvolvimento são ainda mais difíceis de manter conformidade devido à necessidade de lidar com diferentes técnicas e ferramentas. Este trabalho procura aliviar esse problema propondo um processo de conformidade arquitetural independente de plataforma. Técnicas de conformidade arquitetural são comparadas e uma avaliação é feita nos sistemas da Universidade de Brasília (UnB). Seis software foram avaliados, três deles implementados em Java e os outros três implementados em Visual Basic. O processo foi capaz de identificar com sucesso, violações arquiteturais em todos os diferentes sistemas usando a mesma técnica e ferramenta.

**Palavras-chave:** Conformidade arquitetural, DCL, Modelos de Reflexão, Conformidade de software com Alloy

# Abstract

The software maintenance costs are strongly influenced for its conformance with the conceptual architecture and the good development practices. The lack of such conformance generates unnecessary expenses. Heterogeneous environments with different development platforms are even more difficult to keep the conformance due to the need of dealing with different techniques and tools. This work aims to overcome this problem by proposing a platform independent software conformance process. Conformance checking techniques are compared and an evaluation was carried out in the Data Center at the University of Brasilia (UnB) systems. Six software systems were evaluated where 3 were implemented in Java and other 3 implemented in Visual Basic. The process was able to successfully identify architectural constraints violations on all different systems using the same technique and tool.

**Keywords:** Architectural conformance, DCL, Reflexion models, Alloy software conformance

# Sumário

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Objectives . . . . .	2
1.3	Contributions . . . . .	2
1.4	Structure of the dissertation . . . . .	3
<b>2</b>	<b>Background</b>	<b>4</b>
2.1	Software Architecture . . . . .	4
2.2	Architecture Conformance . . . . .	5
2.3	Reflexion Models . . . . .	6
2.4	DSM . . . . .	7
2.5	Query Languages . . . . .	9
2.6	DCL . . . . .	9
2.7	Alloy . . . . .	11
2.8	Related Work . . . . .	12
<b>3</b>	<b>Proposed Solution</b>	<b>16</b>
3.1	Architecture Specification . . . . .	17
3.1.1	Architecture specification with Alloy . . . . .	18
3.1.2	Architecture Constraints on Alloy . . . . .	20
3.1.3	Architecture Specification with DCL . . . . .	21
3.1.4	Architecture Constraints on DCL . . . . .	21
3.2	Dependencies Extraction . . . . .	22
3.2.1	The Generic Language . . . . .	23
3.3	Conformance Analysis . . . . .	24
3.3.1	Alloy Conformance checking . . . . .	25
3.3.2	DCL Conformance Checking . . . . .	26
3.4	Result Output . . . . .	26
3.4.1	Alloy Output . . . . .	27



3.4.2 DCL Output . . . . .	29
3.5 Final Remarks . . . . .	29
<b>4 Evaluation</b>	<b>30</b>
4.1 Experiment Setup . . . . .	31
4.2 Results . . . . .	33
4.2.1 Violations . . . . .	35
4.3 Research Questions Analysis . . . . .	38
4.4 Threats to Validity . . . . .	39
4.5 Critical Analysis . . . . .	39
<b>5 Conclusion</b>	<b>40</b>
5.1 Future Work . . . . .	41
<b>Referências</b>	<b>42</b>
<b>Appendix</b>	<b>45</b>
<b>A Alloy Syntax</b>	<b>46</b>
A.1 Alloy Grammar . . . . .	46
<b>B CPD Architectural Documentation</b>	<b>50</b>
<b>C Dependencies Extraction Output</b>	<b>57</b>
C.1 Javadepextractor . . . . .	57
C.2 Vbdepextractor . . . . .	58
<b>D Conceptual Architecture Files</b>	<b>60</b>
D.1 Siex Architecture DCL File . . . . .	60
D.2 Siex Architecture Alloy File . . . . .	61

# List of Figures

2.1	Conceptual and concrete architecture and its calls. . . . .	5
2.2	Reflexion model process [38]. . . . .	7
2.3	SAVE tool conformance checking results [14]. . . . .	8
2.4	Sigra system's DSM obtained through the VBDepend Tool. . . . .	8
2.5	DCL syntax summary [3]. . . . .	10
2.6	Alloy 4.2 Tool. . . . .	11
2.7	Counterexample found shown on a graphic view. . . . .	12
3.1	Platform independent software architecture conformance process. . . . .	17
3.2	Architecture conformance analysis result example. . . . .	27
3.3	Console checking result of the Alloy model. . . . .	28
3.4	First Alloy counterexample found on the architectural conformance test of the Alloy model. . . . .	28
3.5	Second Alloy counterexample found on the architectural conformance test of the Alloy model. . . . .	28
4.1	CPD's Java System Architecture Layer View. . . . .	31
4.2	CPD/UnB Visual Basic Systems Modular View. . . . .	32
D.1	CPD's Java System Architecture Layer View. . . . .	61

# List of Tables

2.1 Architectural conformance analysis approach comparative . . . . .	14
4.1 Goals for the experiment in GQM format . . . . .	30
4.2 CPD/UNB's Layer Identifiers Nomenclature for Java Systems . . . . .	31
4.3 CPD's systems used on the experiment . . . . .	33
4.4 Single constraint conformance checking tests running time . . . . .	33
4.5 CPD systems conformance checking result . . . . .	34

# Chapter 1

## Introduction

Software Architecture is an abstract entity formed by elements and its relations. Each decision taken into its elaboration will affect the final product concerned to its dependability. Well planned documented, informed and managed architecture enables a better software quality because it allows a better action planning about its construction or evolution. When these aspects are ignored, architectural erosions arise and as its consequence, technical debt from the perspective of architecture since the software will be modified without any type of control [43] [17]. The control over the corrective or evolutive code interventions on a software is not a trivial task. Not only it is hard to keep watching all the incoming modifications, but it is also difficult to read all the code in a non automated way. A good awareness about the conceptual architecture and its supervision are quite sensitive to the developer team education and training. As result, forms of architectural control, capable of identifying architectural problems and constraints violation are highly required and may facilitate these tasks as well as reduce drastically the costs of maintaining a software running.

### 1.1 Motivation

The Informatics Center (*Centro de Informática - CPD*) of University of Brasília (UnB) is responsible for implementation and management of the software that take over the academic tasks at UnB. Through the years, CPD has developed several systems and today it has a vast catalog (over 30 software systems). However, CPD has suffered some drawbacks common on government institutions: high turnover, lack of training, high amount of demands with stringent deadlines, technology and software aging. In particular dealing with software aging is a big challenge because it imposes increasing costs over software evolution and maintenance [40].

Moreover CPD still has to deal with different software each one developed either in Visual Basic, Java, VB.NET, C#, PHP, JavaScript and Erlang. In that context, it is very hard to implement monitoring polices to supervise the software evolution and guarantee its delivered quality. In particular with respect to architecture quality enforcement, a defined conceptual architecture built over good software practices still lacks. Keeping the software architecture conformance in such heterogeneous software development ecosystem [26] is crucial to ensure its dependability. At the beginning of this dissertation, as far as we are concerned, there was no implemented tool capable of doing software conformance analysis regardless of the programming language used. To achieve architecture conformance on the CPD systems, several tools and techniques should be used.

## 1.2 Objectives

This work proposes a solution capable of overcoming the necessity of using several tools and techniques to achieve the conformance on software ecosystems with heterogeneous languages. The objective is to create a software architecture conformance process able to use the same technique to identify the software architectural constraints violations.

## 1.3 Contributions

The software architecture conformance process is innovative by proposing and evaluating a technique capable of identifying architectural violations in different systems written in different programming languages. The dissertation contributions comprises:

- A technique capable of making conformance analysis of legacy systems with different languages.
- A process that permits the use of any conformance technique, where the most suitable one might be used on each context. The process is composed by the following activities:
  - Specification of the conceptual software architecture.
  - Declaration of the architectural constraints.
  - Specification the concrete architecture abstracted from the source code.
  - Detection and report of the source code violations.
- An empiric evaluation of the platform independent conformance process using Alloy and DCL as well as its comparison.

## 1.4 Structure of the dissertation

The remaining chapters of this dissertation are organized as follows:

- **Chapter 2** reviews the key concepts of the background of the dissertation: software architecture, architecture erosion and architectural conformance,
- **Chapter 3** reviews the related work: Alloy language, DCL, DSM, Reflexion Models and SCQL.
- **Chapter 4** proposes a platform independent architecture conformance process to support the heterogeneous environments management.
- **Chapter 5** evaluates the conformance process in our implemented framework on 6 systems implemented at CPD.
- **Chapter 6** presents the final remarks and future work.

# Chapter 2

## Background

### 2.1 Software Architecture

The software architecture is an abstract entity. It is formed by the software elements and its relations. Two of the biggest factors to increase the software costs are its evolution and customization [41]. Those factor are inevitable since the software modification is part of its life cycle. That is why the shape of this entity should not be underestimated, the requirements, architectural elements, design, security issues, algorithms and data types must be taken into account when making the software architecture design decisions to control its characteristics. Some already known and tested architecture styles like Layered System, Pipes and Filters, Event-based and Implicit Invocation [20] might be used as is or to build a customized new architecture. The software architecture is not a created object, every software has one, good or bad, it exists. A better term to talk about the decisions taking on its modifications is *modeling*. A good model of transmitting architectural knowledge, which is a difficult task since the software is usually a big and complex set of structures and relations, is the concept of *views*. A view is a vision of part of the software architecture when divided into relevant elements being evaluated [31] [11]. The concept of views is important because it groups only a part of the system's elements and relations: The ones of interest of a specific concern. It makes easier to understand each specialized part. The combination of all architecture views, forms the full architecture. When modeling a software architecture, all decisions must be deliberate chosen, if not, there is a risk of imperceptible defects or performance issues arise. A well documented and informed architecture will keep a higher conceptual integrity, enable reuse more efficiently and better control [35].

## 2.2 Architecture Conformance

Architecture conformance is the name given to the synchronized association between the intended architecture for a software and its real architecture [29]. The lack of conformance happens for several factors like time pressure, the lack of technical knowledge and conflicting requirements. Some times these factors make the developer ignore the architectural rules and good software practices creating or increasing as consequence, architectural erosion [41]. In that way, the conceptual architecture documented and the concrete architecture existing as the source code structure does not relate anymore or has deviations between them. As an example Figure 2.1 shows a conflict between the software conceptual and concrete architecture. On the conceptual architecture the relation constrains are:

- A must call B
- A must call C
- C must call D

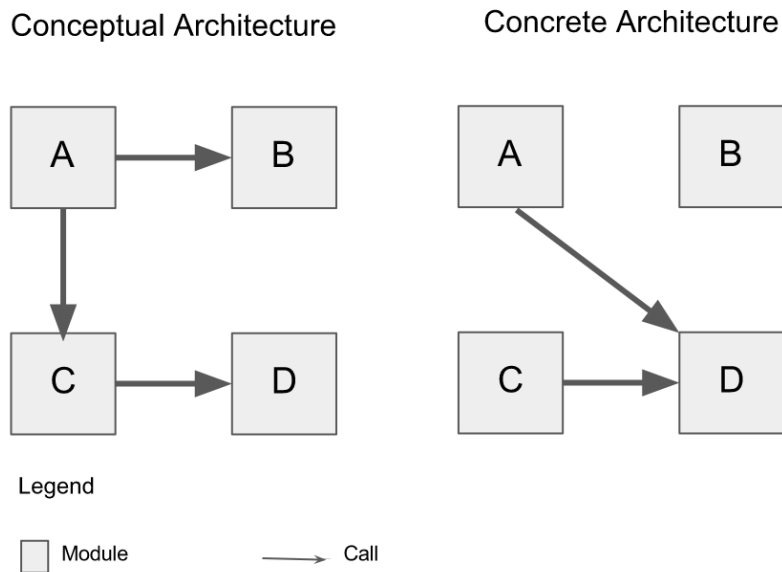


Figure 2.1: Conceptual and concrete architecture and its calls.

But as seen on the concrete architecture, module A might have needed some of the module D features and decided to call it directly creating a software erosion without even knowing. The conformance analysis will compare both models and identify the constraints defined by the conceptual model that were not satisfied by the concrete model. On a constraint checking over the example given, result is the following:



- A must call B (absent)
- A must call C (absent)
- C must call D (satisfied)
- A call D (Not intended)

The hypothetical software has at this stage a software with a source code that will probably be harder to read and costly to evolve, it might also have worse performance.

There are basically two ways of checking for architecture conformance: statically and dynamically. Static approaches are non invasive, they are able to read the source code through a syntax searching for character sequences of interest or searching in the abstract syntax tree (AST) for specific elements. On the dynamic approach, the object of interest is the software instantiated objects and its relations whose behavior is lesser predictable and non measurable through source code analysis.

## 2.3 Reflexion Models

The reflexion model technique seeks to help the understanding and conformance analysis efforts in an iterative and empiric way [38]. A view of the process is shown on Figure 2.2, the technique uses a high level architecture model, a source code abstraction and a mapping between these models. In possession of these artifacts, it is possible to compute a resulting reflexion model that exposes the absences, divergences and convergences found [37]. The technique might be applied through the steps below:

1. High level model definition: All the available information is used to create the architectural documentation with one or more architectural views.
2. Source code model extraction: Here, source code model extracting tools are used to build a lower level model.
3. Models Mapping: The architect declares the corresponding structures and relations between the high and low level models.
4. Reflexion Model computing: The reflexion model is then computed bringing as result the convergences, divergences and absences found in the low level model when compared to the high level model.
5. Refinement and investigation: The result is analyzed to verify if possible errors on the high or low level models definition had happened. After that, the source code problems are informed to the developer team.

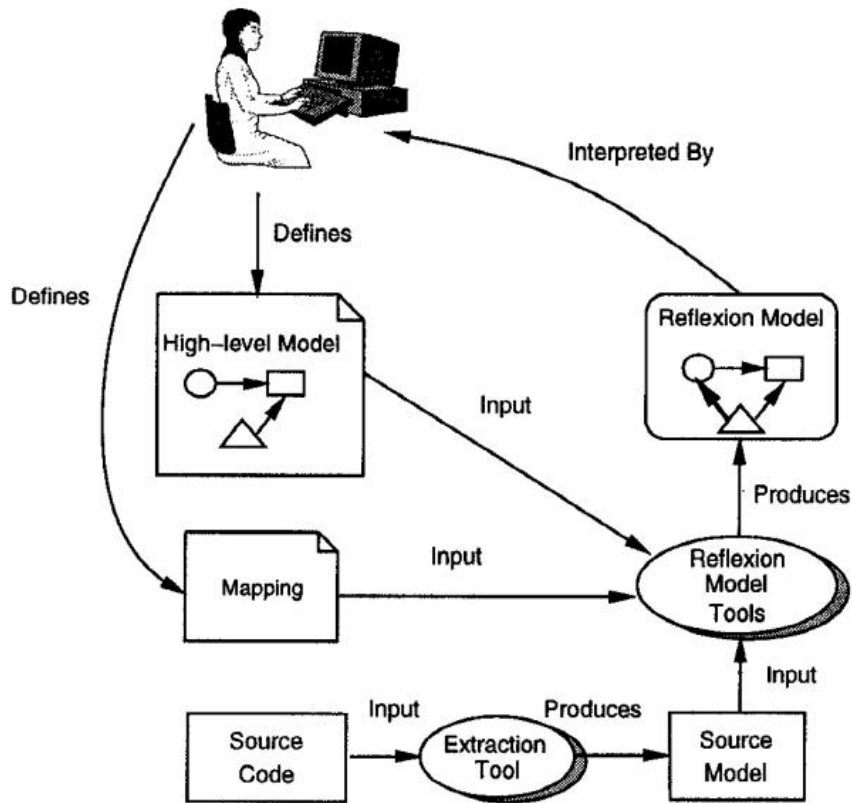


Figure 2.2: Reflexion model process [38].

Some tools have been used to evaluate the technique [14] [37]. Figure 2.3 shows an example of conformance checking made by the SAVE tool [28]. The *convergences* are marked as a check mark in green background, the *absences* are marked as a "X" red mark, the *divergences* are marked as a black exclamation point on an yellow background and finally *unexpected relations* are marked as a black question mark in a blue background.

## 2.4 DSM

The Dependency Structure Matrix (DSM) is a technique that helps analyze complex structures through a Matrix containing elements and its relations [9] [45]. It was first intended to be used on engineering design problems but it was found to be capable of helping to resolve several different problems, as it is the software analysis [43] [32] [44]. It has a simple structure composed by a matrix where the elements are disposed in a vertical line with a perpendicular horizontal line where the same elements are replicated. The Figure 2.4 shows a DSM taken from the Siga system through the VBDepend Tool<sup>1</sup>.

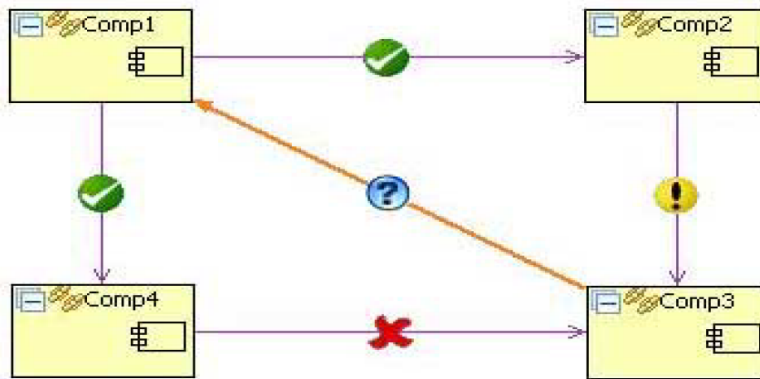


Figure 2.3: SAVE tool conformance checking results [14].

The intersection cells are highlighted and numbered to indicate a dependency and its dependency weight (how many calls were made).

The DSM Tools have the capacity of defining structural constraints which are indentified in the congruent cells of the involved classes of a violation. The kinds of conformance check that the DSM is capable of doing are restricted to *can-use* and *cannot-use* types.

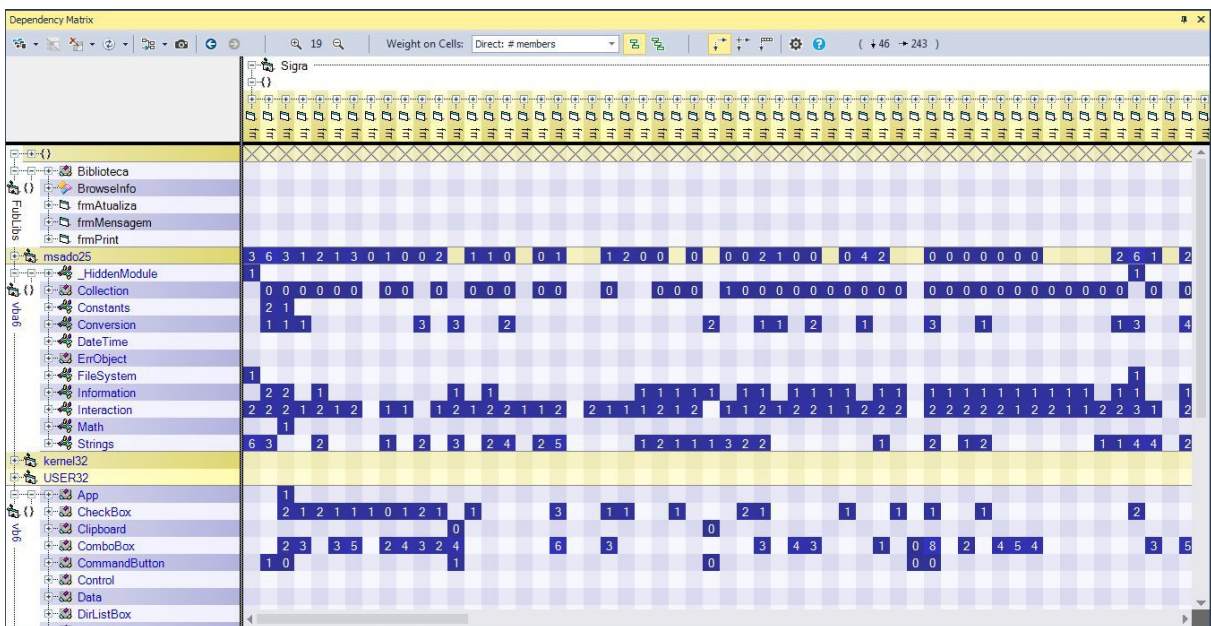


Figure 2.4: Sigra system's DSM obtained through the VBDepend Tool.

## 2.5 Query Languages

The Query Language (QL) is a language based on SQL and Database Theory [18] [13]. It is able to make search queries over source code. It is intentionally similar to the SQL because it is a simple syntax and also on the effort to facilitate its learning since SQL is a well known language. An example is shown below where a query searches the entire project for the methods and then counts its lines of codes. The CQLinq [2] language was used.

```
1 JustMyCode.Methods
2     .Max(m => m.NbLinesOfCode)
3     .ToEnumerable().Sum(loc => loc)
```

The next example searches for a constraint violation (Fields name should start with lower case):

```
1 warnif count > 0 (from f in Fields where
2     !f.NameLike (@"[a-z] ")
3     && !f.IsEnumValue && !f.IsThirdParty
4 select new { f }).Take(10)
```

SCQL [21] is one of its implementation, it is a domain specific language focused on using the relational queries to get version history information. The query languages were also used to query graphs that model hypertext [8].

## 2.6 DCL

The DCL (Dependency Constraint Language) [46] [47] [48], is a specific domain language that supports the system's module definition in a declarative way. It is able to analyze architectural constraints through static analysis on the source code. It is inspired on the ideas of reflexion models [47]. The constraints are defined by a software architect and the conformance checking is made by the *dclcheck* tool [46] which is able to identify and exhibit the violations as *absences* and *divergences* in the source code. The violations are searched by comparing the source code model to the previously defined conceptual architecture with its software modules and constraints definitions.

The first thing on declaring the software's architecture is its modules. In the DCL language this is done by the syntax:

```
module module_name: class_name1, class_name2, ... , class_nameN
```

The declaration supports several classes on one module, also the \* character denotes that all classes on the given package belongs to the module being declared. These are some examples:

```

1 module vision : br.unb.web.siex.vision.*
2 module action : br.unb.web.siex.vision.Action, br.unb.web.siex.business
  .ActionBusiness

```

The next step is the constraints declaration, the Figure 2.5 shows a DCL's syntax summary. The relation between two modules is declared with some types and modifiers. The "only" keyword is optional.

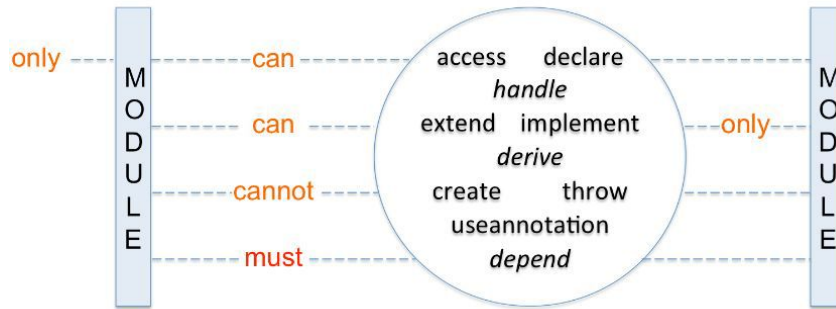


Figure 2.5: DCL syntax summary [3].

An possible example is:

```

1 only Vision can access Business

```

This declaration restricts the access to module "*Business*" exclusively from module "*Vision*". In that way, for example if the "*Persistence*" or "*Pojo*" modules have a dependency call of the kind "*access*", the technique will classify it as a divergence. The "*can*" keyword indicates that module "*Vision*" can access "*Business*" but is not obliged to. If that was the case, then the "*must*" keyword should be used. The language is very simple and easy understandable, some other examples are given below:

```

1 Visao must-extend BaseVisao
2 Negocio must-implement NegocioI
3 Visao cannot-access Persistencia
4 Persistencia cannot-declare Visao
5 only Visao, Negocio, Persistencia can-access Vo

```

The technique's evaluation was done by case study on the Brazil's Federal Data processing Service Organization (SERPRO) [47] where the personal management system were used to check for the approach validity. The authors evaluated 3 versions of the system and were able to successfully identify several architectural violations.

## 2.7 Alloy

Alloy is a declarative language based on first order logic capable of modeling and analyzing structure abstractions [24] [23] [1]. It is essentially a *model finder* because it uses the structure abstractions through declaration of formulas to build a model with the elements and relations satisfying the given formulas.

Alloy works as a simulator where it generates a sequence of states or transitions to satisfy a formula or as a checker where a counterexample that invalidates the formula being checked is searched. It is based and strongly influenced by the Z language and the Tiny Kernel language [24]. The Figure 2.6 shows the graphical mode of the tool being executed. The graphical view of the instance models found to invalidate a predicate or assertion is shown in Figure 2.7.

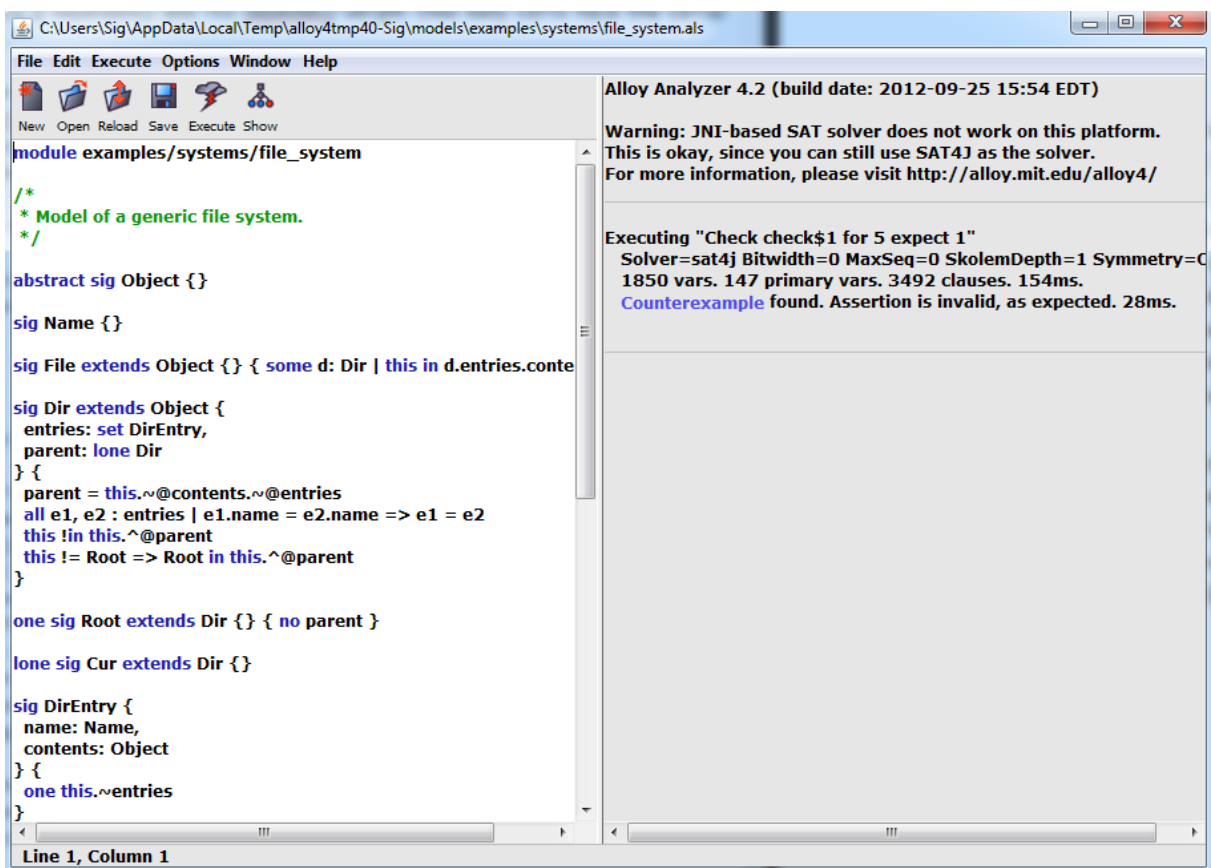


Figure 2.6: Alloy 4.2 Tool.

Alloy models are analyzable, the language has a formal semantics which makes it possible to be automatically analyzed. Its declarative syntax also helps to easily build complex specifications. The Alloy analyzer cannot guarantee a sound and complete analysis, it works through the generation of possible instances within a scope. The numbers

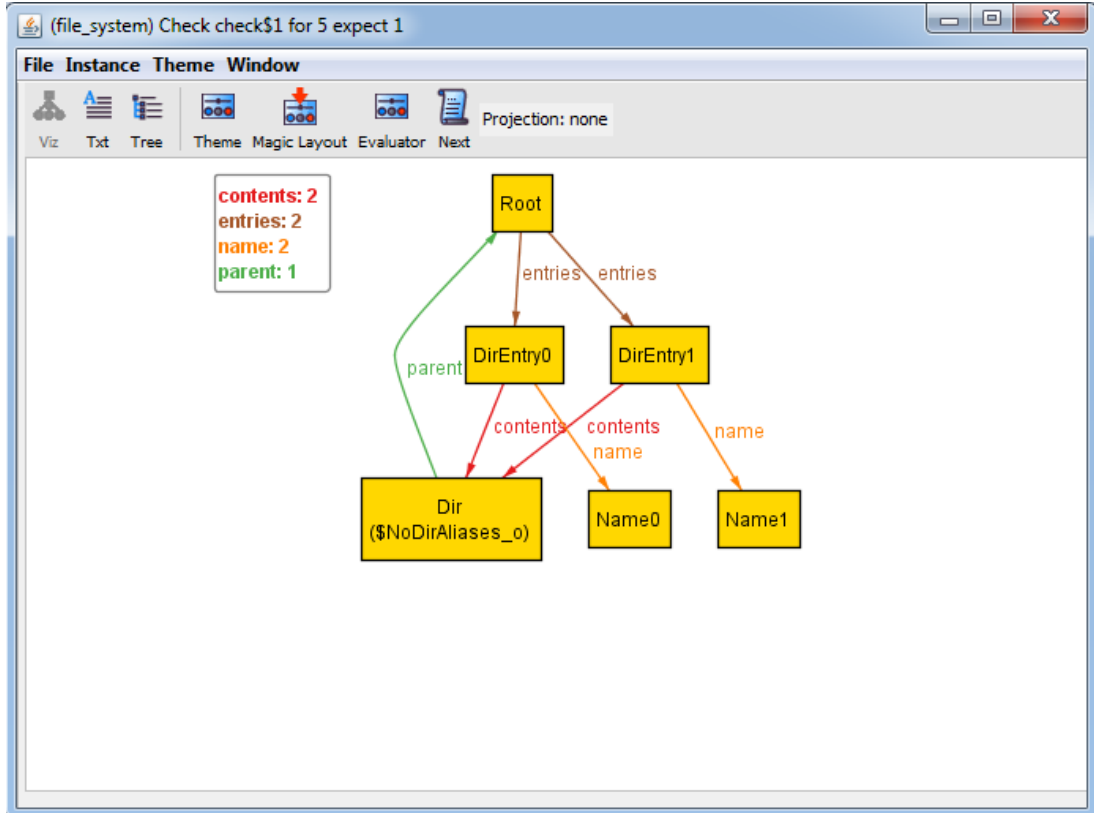


Figure 2.7: Counterexample found shown on a graphic view.

of cases to consider has an exponential growth related to the scope size, so a scope limitation is necessary because if there is not one, the counterexample searching would be infinite. A relation in a scope of  $k$  has  $2^{k*k}$  possible values [25]. The Alloy language mainly uses signatures declarations, facts, predicates and assertions to define and analyze models or simulate a model transitioning states. It reduces those elements and formulas into a boolean satisfiability problem (SAT) which is solved by the built-in SAT solver. The Alloy syntax is shown in Appendix A.

## 2.8 Related Work

Architecture description languages (ADLs) have been used as a form of specifying the software structures to support the reasoning about it. ACME [19] is an ADL capable of defining elements of component-based architectures, its efforts goes on the objective to integrate architecture analysis tools. Darwin [34] is used to specify hierarchically structured architectures, it is possible to use primitive components to composite component types. Darwin deals with the components with what services they provide and what services they require. Related to our problem, the ADLs are not very suited because their goal is the

specifications of architectures to support the reasoning about its structures. Checkstyle has been used to recover and maintain architectural conformance, a software architecture supervision technique through custom checks on commit time was proposed by [36]. The approach was tested on the *Tribunal de Contas da União* (TCU). It uses static source code analysis, visitor pattern and *Checkstyle*<sup>2</sup> tool custom checks to prevent architectural violations on their software. This approach is not suited to our context because it uses a specific tool to a specific programming language, it is a good option to homogeneous environments.

CQLinq [2] is a query language based technique that is used to obtain software metrics and conformance rules checking through customized queries. To our context, the CQLinq is not very suited since, there is no architecture abstraction and each constraint definition is more complex than other approaches.

The SAVE tool [28] is a reflexion model technique capable of comparing high level models to identify deviation between them. It has been used to the conformance analysis, the tool is also used on a proposal of continuous monitoring [29]. The tool might have difficulty on working with several languages simultaneously, it is also less expressive than other approaches such as DCL.

A comparison between three conformance analysis approaches (Reflexion Model, Relation Conformance Rules and Component Access Rules) are done by [30]. They use an implementation of the SAVE tool to evaluate its characteristics. The downside of this work is the strong bonding to the SAVE tool, there is too low possibilities of customization.

ISO/IEC 42010 [15] is a normative guidance that standardize the notion of architecture framework. It define requirements like viewpoint models, relation expressing and correspondence between models to be used by architecture frameworks. This normative is a good basis to influence software conformance checking frameworks, but it is too generic, further exploration is needed to access its importance.

The Alloy language has been used to several conformance analysis, the two more related to the problem studied are the works by Garlan and Kim where they use the Alloy to analyze architectural styles properties [27] and Crane and Dingel where they make dynamic runtime conformance checking of objects [12]. Alloy has also been used to analyze UML runtime models [49] and system access control testing [22]. As far as we are concerned, Alloy has never been used to check software conformance on a static fashion like DCL, SCQL and DSM approaches.

The architecture enforcement with the Checkstyle API is used on a slightly different context, it is used strictly on software written in Java and it is a complex technique when compared to the other ones being analyzed. The ArchLint is a tool capable of

---

<sup>2</sup>Checkstyle: <http://checkstyle.sourceforge.net/>



extracting a software’s architectural knowledge through the static analysis of source code history in the software repository. It is able to understand deeply the architecture but it is complex and lacks abstraction. The SCQL also lacks abstraction. It is less complex to use than ArchLint for example but compared to the other approaches like DCL it is more complex. Its strength is the capability of searching structures on the source code through queries which are highly customizable. DSM is language-independent, some tools already use them to abstract software architecture [5] [6]. It is very simple to use and understand. Its downside is the low expressiveness. The DCL language uses Reflexion Models concepts, that is why both has several common characteristics, their concepts are easily portable, they are capable of modeling structures on a high level abstraction and checking architectural constraints on the source code. The difference is that DCL language is more expressive, it allows declaration of more types of relations like for example implement, create, throw and extend. Alloy is a model checker that is able to define and analyze structure and its relations. It has never been used to make software conformance analysis through static source code analysis so an investigation about this possibility is done through the evaluation. We summarized the result of our study on Table 2.1

Table 2.1: Architectural conformance analysis approach comparative

	Multi Language Implementation	Dependency	Abstraction	Simple
SAVE [28]	No	Yes	Yes	Yes
CQLinq [2], SCQL [21]	No	Yes	No	Yes
DSM [43]	No	Yes	Yes	Yes
Checkstyle [10]	No	Yes	No	No
ArchLint [33]	No	Yes	No	No
DCL Suite[46] [48]	No	Yes	Yes	Yes
Alloy Analyzer[23]	No	Possibly	Possibly	No

The *Multi Language Implementation* column answers if the technique has a concrete implementation able to work on different developing platforms with its different programming languages. Most techniques are able to make the conformance analysis on different languages but using different implementations. The column is related to if there is an implemented version capable of dealing with any language, the answer "no" is given to the technique which have no tools or scientific work proving the concrete application of the technique implementation on different platforms. In the *Dependency* column it is analyzed if the technique is able to check dependency constraints between modules. The "Possibly" answer is given to the technique potentially able but not used for this objective yet. *Abstraction* Is related to the technique’s ability to promote higher levels of abstraction of the architecture components. The abstraction allows lower the architectural conformance process effort by facilitating the understanding of the structures and relationships

of the software. The last column (*Simple*) is related to if it is easy is to understand and use the technique or not.

# Chapter 3

## Proposed Solution

In this chapter we present the core of our work towards an architecture conformance process for software ecosystems with heterogeneous development languages. The proposed process is capable of identifying deviations of the software source code from the specified conceptual architecture. It uses the same technique to analyze the architectural conformance of software with different programming languages. The process relies on source code documentation patterns as well as architecture specification, constraints defined by the architect and the source code extracted architecture. The violations revealed by our reflexion based process are informed to the practitioner.

The reflexion process proposed starts by the specification of the architectural elements, then acquiring the source code elements, making the conformance analysis and finally showing the results. It comprises 5 steps:

- **Architecture Specification:** The architect must define the conceptual architecture rules, its modules, constraints and requirements.
- **Dependence Extraction:** The source code dependency extraction provides the concrete architecture view. The dependency file must be written on a single language independent of the programming language used to build the software.
- **Conformance Checking:** The translated model is checked over the conceptual architecture defined. All deviations between the models are captured.
- **Result Output:** The architecture conformance result must inform through the textual or visual exhibition, the deviations found and what are its kind (absence or divergence).

Figure 3.1 shows the process view, each process step is further explained in the next subsections and evaluated in Chapter 4.

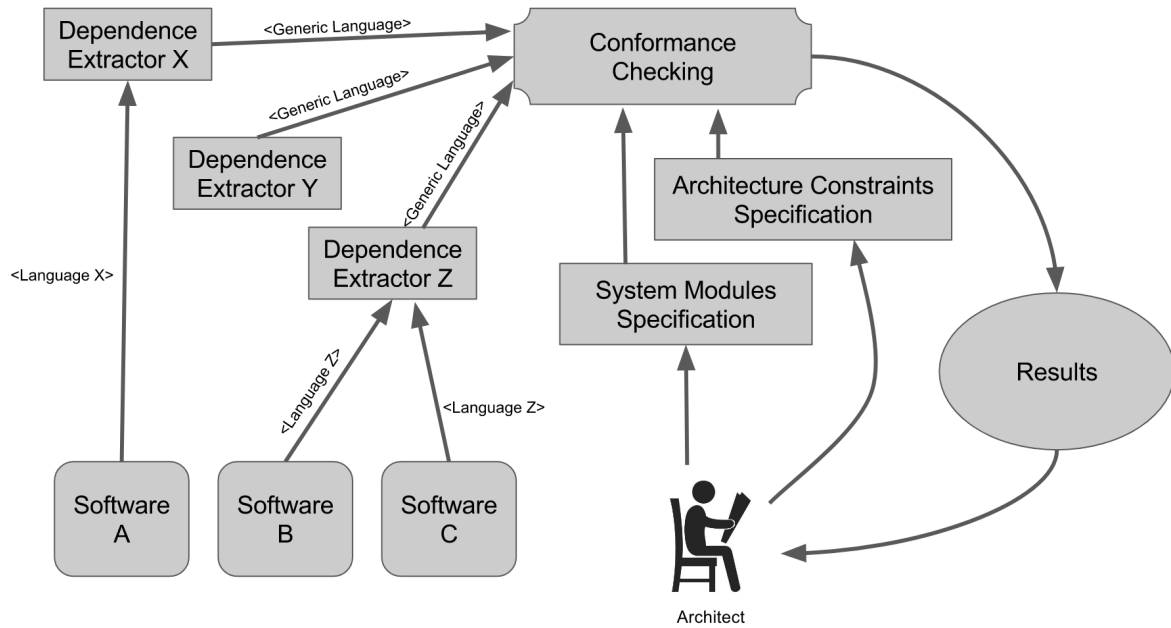


Figure 3.1: Platform independent software architecture conformance process.

### 3.1 Architecture Specification

This is the first step of the process. It is responsible for the "System Modules Specification" and "Architecture Constraints Specification" boxes of Figure 3.1. It takes as input the definition of an architect and all the documentation and knowledge existent. The architect is important because he is the responsible for defining the conceptual architecture documentation. If there is no architect, an experienced developer might assume the role. An architecture documentation is composed by several different views so, the architect must be able to define what kind of view will have its constraints checked. For example the layering view has the definition of the layers and how they communicate to each other, which ones are allowed or forbidden to make a dependence call to a specific layer. This modeling of an architectural view is important to allow a higher level of abstraction about the software, it is easier to understand and verify the architectural aspects that way. To fit on the proposed process, the conceptual architecture of all software present on the ecosystem must be specified on a singular language. Any language might be used as long as it is able to specify the architectural elements definition and their mapping to the source code artifacts. Also all architecture constraints must be specified.

With the software documentation in hands, the architect must specify on the chosen specification language, the software division (modules or layers) and its relation constraints. Those information compose the output of this step, the specified conceptual

architecture.

### 3.1.1 Architecture specification with Alloy

The Alloy model must be capable of defining the modules, the relations between them and its constraints. A Java project<sup>1</sup> was created to test several Alloy models with different formats. The most efficient modeling was the one structured in the following format:

```
1 module project_name
2
3 sig Class{}
4 sig Layer_name_A, Layer_name_B {classes: set Class}
5 sig Class_name_A, Class_name_B extends Class{}
6 sig Relation_type_1, Relation_type_2 {r: set Class -> Class}
7
8 fact {Layer_name_A.classes = Class_name_A + Class_name_X}
9 fact {Layer_name_B.classes = Class_name_B + Class_name_Y}
10
11 fact {Relation_type_1.r = Class_name_A -> Class_name_B}
12 fact {Relation_type_2.r = Class_name_X -> Class_name_Y}
```

The first line has the Alloy *module* definition, the syntax might be confusing when the dependencies between modules are being analyzed. This kind of module is a higher Alloy project division, like a package in the Java syntax. On the tests made, since it is used a simple layering architecture, the lower level modules are called "*layer*". The module declaring has the following syntax:

```
1 module system_name
```

Example:

```
1 module sisru
```

Or

```
1 module siex
```

On the line 3, the Class model type is declared as a signature, it was named "Class", it will be inherited by all the Classes. It might be seen as a super Class of all the Classes declared on the model.

On line 4, all the modules (the layers in this particular case) are declared, each one has a set of classes inside it. They will later be populated with the relevant classes. The layer declaration is done like the example below:

```
1 sig vision, business, persistence, pojo {classes: set Class}
```

---

<sup>1</sup>GitHub project of the Alloy testing tool: <https://github.com/Sigfredo/AlloyTesting>

The classes are declared on line 5 on a straightforward way: the class qualified name followed by the higher set (the super class) extension:

```
1 br_unb_web_siex_visao_ManterPropostaListagemVisao extends Class{}
```

The full syntax is seen on the example below:

```
1 sig br_unb_web_siex_visao_ManterPropostaListagemVisao ,
   br_unb_web_siex_negocio_AlocaMembroExtensaoNegocio extends Class{}
```

On Line 6 the relations types are declared, each one of them has a set of relations between classes. This is how the dependency type will be identified. Example:

```
1 sig Access, Declare, Create {r: set Class -> Class}
```

The layers are then populated on lines 8 and 9 through Alloy facts. The nomenclature pattern and packages used on the university's systems help to identify from where each class comes from. The pattern is shown in Table 4.2 of Section 4.1. This knowledge helps the testing tool to obtain automatically the classes layer. Binary relations are used on the form "A: B" or "A: B + C + D" when declaring several relations. Concrete examples on the layer populating are:

```
1 fact vison{ Layer.vison = InscricoesConfirmadasVisao +
2   ReplicarPropostaVisao}
```

or

```
1 fact business{ Layer.business = ManterProgramaTopicoNegocio +
2   ManterEditalNegocioImpl }
```

Lines 11 and 12 show the dependencies declaration form. The relations are declared through ternary relations on the form "A: B -> C". Where A is the variable holding all dependencies and B and C are respectively the classes making a method call and receiving the method call. A variable was declared before to identify each kind of relation. The appropriate one is used on this step. Below, an example is given:

```
1 fact {Access.r = br_unb_web_siex_visao_ManterPropostaListagemVisao ->
   br_unb_web_siex_negocio_AlocaMembroExtensaoNegocio +
   br_unb_web_siex_persistencia_CertificadoDAOImpl}
2
3 fact {Implement.r = br_unb_web_siex_persistencia_ManterAvaliacaoDAOImpl
   -> br_unb_web_siex_persistencia_ManterAvaliacaoDAO +
   br_unb_web_siex_pojo_Parecere -> java_io_Serializable}
```

The Alloy architecture specification, must be manually defined by the architect. The Alloy architecture model is very extensive and verbose, for this reason, the modules and classes information were inserted on the Alloy testing tool so it could extract some of the information automatically. The file generated is in fact very extensive (35 lines and

187333 columns of characters), part of it may be seen on Appendix D. The declarations types used are: access, declare, create, extend, implement, throw and useannotation.

### 3.1.2 Architecture Constraints on Alloy

An Alloy fact is an affirmation that always holds a true value. The Alloy assertive is different, it holds an affirmation that will be checked. The architectural constraints fits well in this definition, the architect must be capable of transforming the documentation's information into Alloy assertive. An Alloy assertive has the following syntax:

```
1 assert Assert_name{
2 all x: class_A, y: class_B | y in x.variables}
```

Where assert is the Alloy identifier for the assertive and inside the brackets there is a formula to be checked by the analyzer. For example, the Java layering view (Appendix B) specifies that no vision class can access directly a persistence class. One possible example is: "No class from vision can create a class from persistence". On Alloy syntax we have:

```
1 no x: Layer.vision, y: Layer.persistence | x->y in Create.relations
```

Being "*relations*" the variable holding all the dependency calls between the software classes of that type of relation.

Another example: "no class from persistence can implement a class from business":

```
1 no p: Layer.persistence, b: Layer.business | p->b in Implement.relations
```

The full assertion declaration uses the assertion syntax with the constraint formula inside it. It will be later checked by the *check* command. The concrete example is given below:

```
1 assert negocio_visao{
2   no x: Layer.vision, y: Layer.persistence | x->y in Create.relations
3 }
```

To effect of the evaluation, the constraint "No vision class may access the persistence layer classes" was used. This assertive was included on the Alloy architecture file and it was translated to the following syntax:

```
1 assert no_visao_persistencia{
2   no x: visao.classes, y: persistencia.classes | x->y in Access.r
3 }
```

Part of the file is disposed on the Appendix D, the full file is generated by the class "*GenericToAlloy.java*" available on the Alloy testing tool.

### 3.1.3 Architecture Specification with DCL

The DCL Architecture file is a *.dcl* extension file containing the architecture written in the dcl language informing the software's modules and its constraints. It is important to note that the modules cited here are different from Alloy modules. DCL modules are parts of the software grouped by affinity, like layers as the examples seen so far. The DCL architecture file is shaped in the way showed below:

```
1 module Module_A_name : Class_A_name
2 module Module_B_name: Class_B_name
```

Line 1 and 2 contain the module declaration. Each class name belonging to a module or the whole package must be declared as components of a module. The declaration is very straightforward. An example containing both cases is show below:

```
1 module vision: br.unb.web.siex.visao.InscricoesConfirmadasVisao
2 module business: br.unb.web.siex.negocio.*
```

It is also possible to use regular expressions to get the right classes through its nomenclature pattern, like this:

```
1 module visao: "br.unb.web.siex.visao.[a-zA-Z0-9/.]*Visao"
```

To exemplify the software architecture specification on dcl, the created documentation of the Java systems (using Siex as the test case) was used. The file is transcribed next <sup>2</sup>:

```
1 module $sql: java.sql
2 module visao: "br.unb.web.siex.visao.[a-zA-Z0-9/.]*Visao"
3 module basevisao: br.unb.fast.core.camada.visao.BaseVisao
4 module negocio: "br.unb.web.siex.negocio.[a-zA-Z0-9/.]*NegocioImpl"
5 module basenegocio: "br.unb.web.siex.negocio.[a-zA-Z0-9/.]*Negocio"
6 module icrudnegocio: br.unb.fast.core.camada.negocio.ICrudNegocio
7 module basepersistencia: "br.unb.web.siex.persistencia.[a-zA-Z0-9/.]*DAO
  "
8 module persistencia: "br.unb.web.siex.persistencia.[a-zA-Z0-9/.]*DAOImpl
  "
9 module daofactory: br.unb.web.siex.persistencia.DAOFactory
10 module pojo: br.unb.web.siex.pojo.*
11 module vo: br.unb.web.siex.vo.*
```

### 3.1.4 Architecture Constraints on DCL

The constraint declaration follows the model shown below:

```
1 (modifier) Module_A_name (verb)(relation_type) Module_B_name
```

---

<sup>2</sup>The transcription has only the modules definition, the constraints were taken out to be more clear



Or:

```
1 Class_B_name cannot-access Class_A_name
2 only Class_A_name can-create Class_B_name
```

Where the modifier is optional, the verb is one of the DCL's operators verbs like can, cannot and must and the relation\_type is the kind of relation between the classes, for example access, throw, extend and depend. Some examples are:

```
1 Vision can access Business
2 Vision cannot access Persistency
3 Vision must extend BaseVision
4 Pojo cannot depend Vo
5 only Business can throw BusinessException
```

The Siex constraints specified for the evaluation is shown next on a transcription of the DCL architecture file <sup>3</sup>:

```
1 visao must-extend basevisao
2 negocio must-implement basenegocio
3 only negocio, daofactory can-create persistencia
4 basenegocio must-extend icrudnegocio
5 only persistencia can-access $sql
6 negocio cannot-access visao
7 only negocio, daofactory can-access persistencia
8 only negocio, daofactory can-access basepersistencia
9 persistencia must-implement basepersistencia
10 persistencia cannot-access visao, negocio
11 pojo cannot-access visao, negocio, persistencia, vo
12 vo cannot-access visao, negocio, persistencia
```

## 3.2 Dependencies Extraction

The source code dependency extraction provides the view of how the concrete software architecture is structured. This step is related to the "Dependence Extractor" boxes in Figure 3.1, its inputs are the software source codes. Each software programming language has its own structuring characteristics and operation, anyhow a single language must be used in a sense that the extraction output have the same syntax independent of the software's language. The information of interest to the evaluation are the software dependencies which will be a view of the concrete architecture of the source code. The dependence file must contain the information about the class calling, the class called and

---

<sup>3</sup>The transcription has only the constraints, the part with the modules were already shown in subsection 3.1.3

what kind of dependence is that. It might be access, implementation or creation for example.

To extract the software dependencies, the project's source code must be walked checking each line if there is a dependence call to another class. All the dependencies found must be included on the output file. To convergence of several languages on a single one, a simple generic language is used. It is further explained on the next section.

### 3.2.1 The Generic Language

To use the same kind of conformance analysis, it is necessary a language conversion step. It might happen either inside the analysis tool or on the dependence extracting task. To converge inside the analyzer, it is necessary to deal and translate a huge number of possible tools output. We develop an extractor able to export the dependencies extraction on the desired format for each different programming language. The output proposed for the dependency extraction step is the one given by a simple generic language. This language has the most simple and objective information required:

- The class making the call
- The class receiving the call
- The type of the call

The required information is separated by comma and the dependencies types used on this dissertation are: *Declare*, *Access*, *Create*, *Extend*, *Implement*, *Throw* and *Useannotation*. The language has the following format:

```
1 Module_A,dependency_type,Module_B
```

A practical example is given by the following *AbrirCaixaVisao*<sup>4</sup> class source code:

```
1 @Named
2 @SessionScoped
3
4 public class AbrirCaixaVisao extends BaseVisao {
5
6     @EJB private CaixaNegocio caixaNegocio;
7     public String getDisplayAbrirCaixa(){
8         Caixa ultimoCaixaAberto =
9         caixaNegocio.buscarCaixaAindaAberto(
10             codigoPessoaLogado);
11     }
12 }
```

---

<sup>4</sup>To a better exhibition, the class was edited to keep only the necessary elements. It was taken from the University of Brasília's restaurant system (Sisru)

The class segment above holds the following dependency on the generic language syntax:

```
1 AbrirCaixaVisao , access , CaixaNegocio
```

The complete version of the dependency must contain the qualified class name informing its package hierarchy, this is important to identify and divide the software modules. The qualified version is:

```
1 br.unb.web.sisru.visao.AbrirCaixaVisao , access , br.unb.web .  
2 sisru.negocio.CaixaNegocio
```

The conformance analysis tool must be able to deal with this language to compare the concrete architecture with the conceptual architecture specified on its the tool's description language.

CPD has mainly Java and Visual Basic systems. As far as we are concerned there was no Visual Basic dependency extracting tool able to output the dependencies on a textual format close to the one used by the generic language. A Visual Basic dependency extractor was then implemented<sup>5</sup> specifically to evaluate this work. The extractor was build using the university systems characteristics to get the dependencies and export them to a textual file written on the generic language. To better understanding the output and its format, the first page of the file generated by the *vbdepextractor* when getting the University of Brasília Academic system (*Sigra*) is shown in Appendix C. To extract the Java software dependencies, a third party tool<sup>6</sup> was already implemented which gives the result output on the generic language. *javadepextractor* was used to acquire the dependencies of Java source code. The tool has as input the project directory path, the output is given in a text file (*.txt*) containing the dependencies written in the generic language. The Appendix C contains the first page of a file generated by the tool when it extracted the dependencies of the University's academic extension system (*SieX*). On its evaluation, no problem was identified in its output.

On both tools, the output is given on a text file called *dependencies.txt* which is written on the generic language syntax. The Appendix C has more information about the files extracted. All the original files may also be found on the Git repository<sup>7</sup>.

### 3.3 Conformance Analysis

On this stage the conceptual architecture model with the modules definitions and its restrictions are used to guide the searching for the source code violations. The "Confor-

---

<sup>5</sup><https://github.com/Sigfredo/vbdepextractor>

<sup>6</sup><https://github.com/rterrahb/javadepextractor>

<sup>7</sup><https://github.com/Sigfredo/Unbconformancefiles>

mance Checking" box on Figure 3.1 is related to this step. As was presented, the inputs to this step are the outputs of the previous steps: The file containing the system modules and architecture constraints specification as well as the file containing the source code dependencies on the generic language. Several conformance techniques exist using different operations and each context might demands a different one. Aspects like the capability of analyzing dependencies between modules, if it is platform independent, the level of source code modification, easiness of use and results intelligibility are aspects that might be taken into account when choosing a technique. The way the techniques check for constraints violation vary, for instance DCLcheck uses static code analysis to search for string patterns that are compared to previous defined modules. Alloy Analyzer transform the specified constraint formulas into a SAT problem, it is ran by a built-in SAT Solver to generate models using the module signatures and facts together with the constraint assertive. It searches for a counterexample model, if it finds, the constraint being analyzed is invalidated.

The process does not imposes a restriction related to what technique will be used but might be necessary on some cases, adapting the chosen one to deal with the dependencies file written on the generic language syntax. There is no need of adapting the conceptual architecture definition since when choosing a technique, the conceptual architecture specification will be already defined on its syntax. An Alloy testing tool<sup>8</sup> was constructed to test the Alloy models and conformance checking. It is also capable of transforming the generic language into Alloy syntax. The *pi-dclcheck* is a DCL tool that already deals with the generic language so, there is no need of adaptation when using it.

The output given by this step is the result of the conformance checking, which constraints were violated and by what dependencies defined on the source code extraction file.

### 3.3.1 Alloy Conformance checking

The proposed process uses the Reflexion Model concepts to verify the software architecture conformance, identifying its constraints violations. Since only the divergences and absences are of interest, an Alloy assertive must be declared as a restriction or absence check returning true or false to each assertive declared, it is important to note that the Alloy Analyzer is not capable of checking multiple assertive with one command. The *check* command analyze a single assertive given reducing it and all the signature declaration and facts to a SAT problem. It returns true or false depending on if it was able to find a counterexample or not in the scope given. The case being tested has 2 variables being related to each other so, a scope of 2 is used. The checking command, after executed will bring the result in two ways:

```
1 Counterexample found. Assertion is invalid.
```

Meaning that the assertive is certainly invalid, or the affirmation phrase given is false.

```
1 No counterexample found. Assertion may be valid.
```

As no counterexample was found, the assertive probably is valid. It is not possible to claim that the assertion is always true. But it is true inside the limit given (the scope). That is way Alloy is a scope complete analyzer. In the case given, the scope 2 was chosen so, it is possible to claim that the assertive is certainly true in the scope of 2.

The constraint checking command below was used to verify the conformance:

```
1 check no_visao_persistencia for 2
```

### 3.3.2 DCL Conformance Checking

The DCL's conformance checking uses 2 files, one containing the module declaration and one containing the dependency calls. The tool *pi-dclcheck*<sup>9</sup> has the feature of reading the files written in the generic language syntax so that is the one used on this work.

The *pi-dclcheck* tool populate the modules and dependencies. After that, it checks for the constraints rules in the dependencies set. Its output is given in a textual file called *violations.txt* created in the folder path given as input.

With the generated files on the previous process steps ready (*architecture.dcl* and *dependencies.txt*), the tool's execution command was used:

```
1 java -jar pi-dclcheck architecture.dcl dependencies.txt
```

After the execution a text file is generated containing the information of the violations found.

## 3.4 Result Output

The result output is given with the information about what deviations were found on the concrete architecture when compared to the conceptual architecture. The relations of interest are the ones given by any reflexion model inspired approach: the convergences, divergences and absences found in the extracted architecture from the source code when compared to the conceptual architecture defined by the architect. Figure 3.2 shows the information of interest. Although all these information are important to understand the software, on our context, the convergences where not relevant so they are ignored by the results.

---

<sup>9</sup>pi-dclcheck GitHub project: <https://github.com/rterrabh/pi-dclcheck>

There are basically two forms of exhibiting the conformance checking results: Through visual exhibition and through textual information. DSM and Alloy inform the constraints violations through a visualization of models generated by the concrete architecture and the conceptual constraints on the Alloy case and on a Matrix with converging cells on the DSM case. DCL and Reflexion Models output is simple, straightforward and also very informative. It tells through a textual form, which kind of violation was found, the caller and the called class and also what type of call was the one that violated the analyzed constraint. With this information, the developer team is capable of doing a guided source code adjustments.

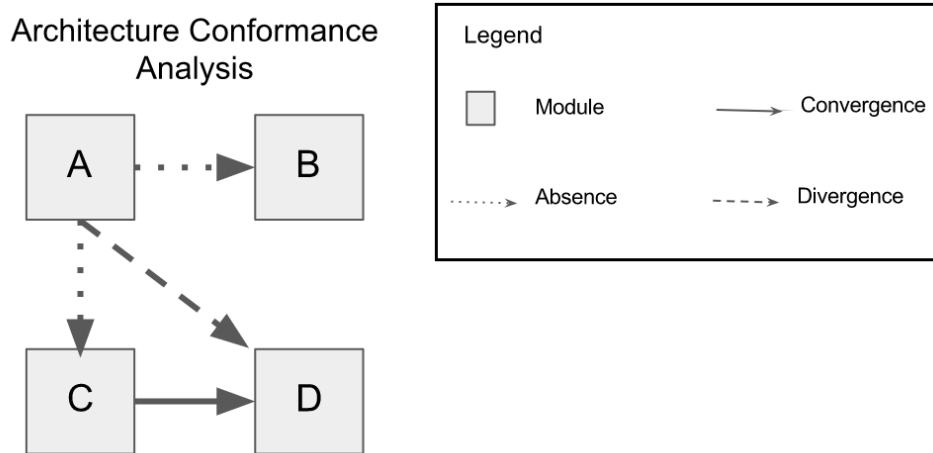


Figure 3.2: Architecture conformance analysis result example.

### 3.4.1 Alloy Output

The information given by the Alloy Analyzer console can be seen in Figure 3.3. The analyzer information shows the time taken to prepare the variables and searching for counterexamples that invalidate the constraint specified on the assertive. Figures 3.4 and 3.5 show counterexamples found, it is possible to see that the "*EmitirHistoricoMembro-Visao*" is the dependence that violates the constraint on both cases. The Alloy analyzer does not show all the relations violating the constraint, it might show but as its operation mode works, it stops if finds a counterexample instance breaks the assertion. As so, it might show only one violation, which were all the cases we found. Its objective is to tell if an assertion holds or not. Chances are that the result show all the violations on a single counterexample, but it is impossible to predict, if enough counterexamples are generated, different violations might appear.

Executing "Check no\_visao\_persistencia for 2"  
 Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20  
 43515 vars. 1168 primary vars. 123856 clauses. 248ms.  
 Counterexample found. Assertion is invalid. 327ms.

Figure 3.3: Console checking result of the Alloy model.

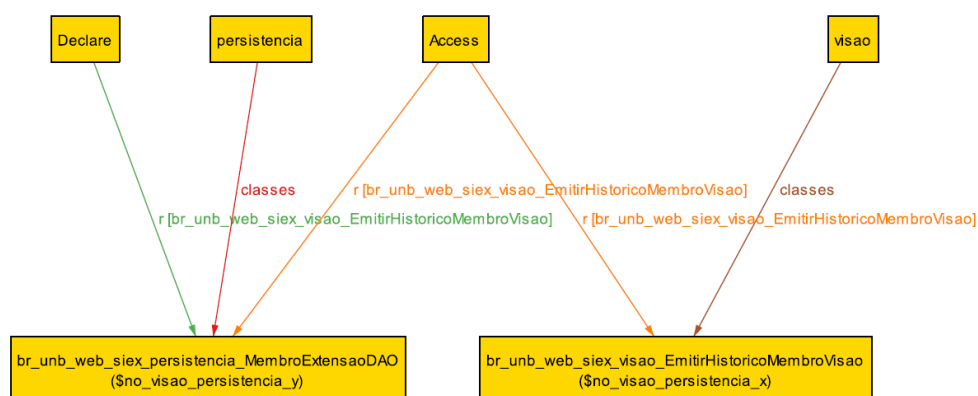


Figure 3.4: First Alloy counterexample found on the architectural conformance test of the Alloy model.

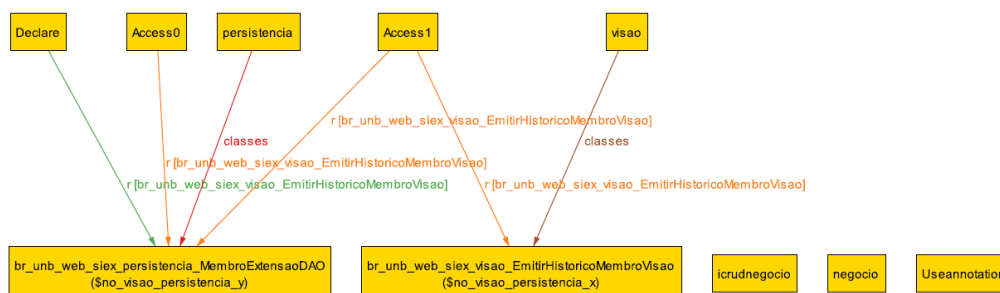


Figure 3.5: Second Alloy counterexample found on the architectural conformance test of the Alloy model.

### 3.4.2 DCL Output

The resulting violations file gotten from the execution of *pi-dclcheck* over the dependency file and the architecture file containing the architecture modules and constraints. The output has the following model:

```
1 [violation_type] , [dependency] , [constraint_violated]
```

Where the *violation\_type* might be *divergence* or *absence*, the *dependency* is the dependency given on the generic syntax format that violated the constraint and the *constraint\_violated* is the textual constraint rule violated. The output file is transcribed below:

```
1 [divergence] , [br.unb.web.siex.visao.EmitirHistoricoMembroVisao , access , br
  .unb.web.siex.persistencia.AlocaMembroExtensaoDAO] , [visao cannot-
  access persistencia]
2 [divergence] , [br.unb.web.siex.visao.EmitirHistoricoMembroVisao , access , br
  .unb.web.siex.persistencia.MembroExtensaoDAO] , [visao cannot-access
  persistencia]
3 [divergence] , [br.unb.web.siex.visao.ManterPropostaFormularioVisao , access
  , br.unb.web.siex.persistencia.OrgaoExternoParceriaDAO] , [visao cannot-
  access persistencia]
```

## 3.5 Final Remarks

The process is based on the reflexion approach, it must be used on iterative runs to refine the artifacts acquired. It was created to be independent of programming language and support tools. On the next Chapter, an evaluation takes place to investigate if the process is really capable of identifying violations on heterogeneous environments.



# Chapter 4

## Evaluation

The process was evaluated through an experiment on the CPD's released software being used by the university. CPD context has a software ecosystem environment which is heterogeneous and has serious technical debt. There was no well defined documentation neither the architect role but these problems were solved through the application of the process. The Goal Question Metric was used to plan the evaluation experiment, Table 4.1 brings its information.

Table 4.1: Goals for the experiment in GQM format

<b>Object of study</b>	Architecture conformance
<b>Purpose</b>	Evaluate
<b>Focus</b>	Identification of violations
<b>Stakeholder</b>	Software architect
<b>Context factors</b>	UnB software ecosystem, heterogeneous languages

The research questions raised through the development of the dissertation which will help to evaluate and understand the proposed process and its characteristics are:

RQ1 Is the proposed process capable of identifying architecture violations on different software languages?

RQ2 Which tool is more suited to identify violations on software ecosystems with different programming languages?

The next sections explain how the experiment was made, using the proposed process to support the conceptual architecture specification, dependencies extraction, conformance checking and result exhibition.

## 4.1 Experiment Setup

Although there was no architect neither a well defined documentation on the CPD's legacy systems, there was some spread knowledge between experienced developers. One of them received the architect responsibility for the evaluation of the process. As there was no documentation, the architect was asked to create one, he analyzed old related documentation, the software source code and interviews with the developer team to create a layering view of the biggest CPD systems which are written in Java and Visual Basic. There is on CPD a nomenclature pattern for classes and projects, it is well known but not documented. Anyway, it was very helpful to identify and divide the modules. Table 4.2 shows the CPD's Java projects nomenclature. The resulting Java view is shown in Figure 4.1. The full generated document of the view is in Appendix B.

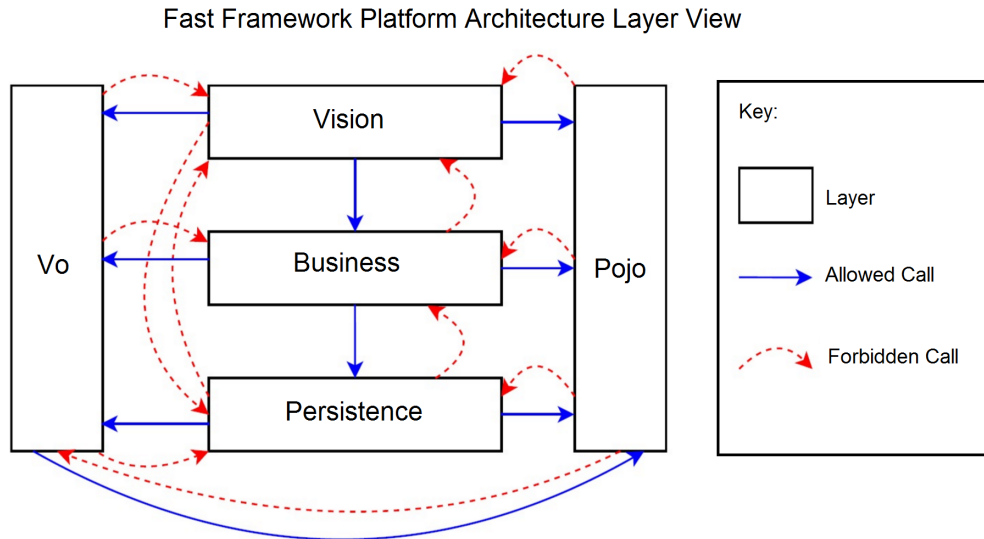


Figure 4.1: CPD's Java System Architecture Layer View.

Table 4.2: CPD/UNB's Layer Identifiers Nomenclature for Java Systems

Layer	Package	Identifier
View	br.unb.web.project_name.visao	Visao
Business	br.unb.web.project_name.negocio	Negocio NegocioImpl
Persistence	br.unb.web.project_name.persistencia	DAO DAOImpl
Pojo	br.unb.web.project_name.pojo	None
Vo	br.unb.web.project_name.vo	VO

The Visual Basic projects intrinsically lacks a refined module division form, although it still has some module division and specialization which permits a module architecture specification. It will be the one used to evaluate the process. Figure 4.2 shows the modular view of such Programs. The full documentation of the Visual Basic system view is also in the Appendix B.

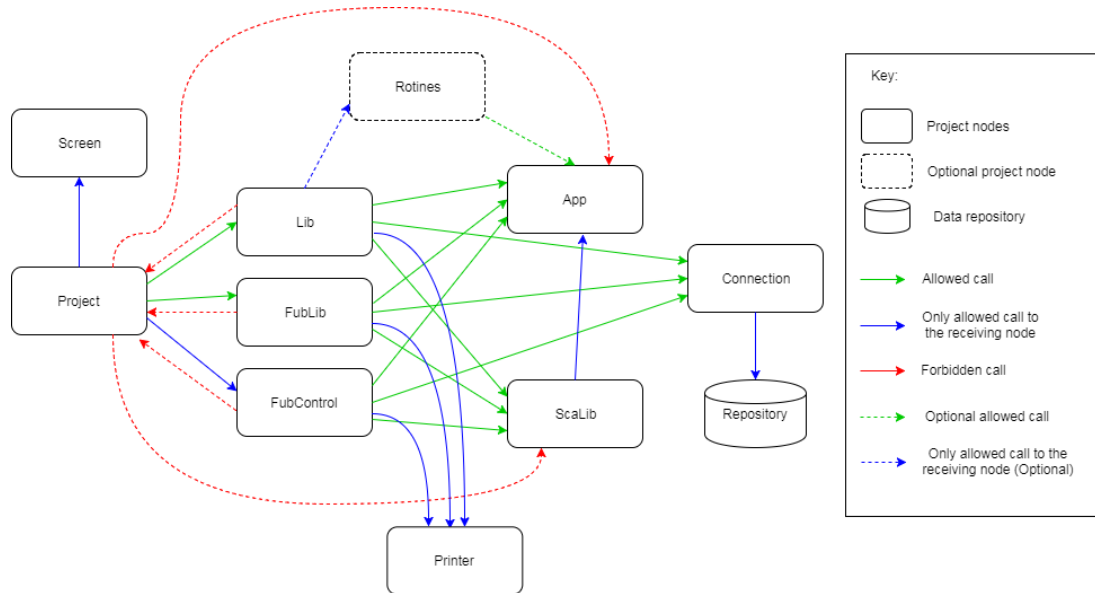


Figure 4.2: CPD/UnB Visual Basic Systems Modular View.

There is no specific layering pattern but to keep a good level of cohesion and facilitate maintenance, some modules were specialized in some tasks. So there is in fact architecture rules, Figure 4.2 shows how the modules relate to each other and also the allowed and forbidden dependencies. It was not possible to get the exact well defined and detailed architecture of the Visual Basic systems but a simple version of it was obtained. That is good enough as we need to validate if the process is able to make architecture conformance analysis platform independent not restricted to documentation quality.

To carry out the experiment, the top six largest software found on the CPD environment were used. Three of them were written on the Java language and the other three on the Visual Basic language. Table 4.3 shows the details of such software. Alloy and DCL were chosen to make a preliminary evaluation, an comparison is made and the the most fitted is used to identify the violations on all systems chosen. The characteristics of the experiment comprises:

- Six industrial systems released by the CPD (Table 4.3).
- Layer Architecture View for the Java software and Modular View for Visual Basic software.

- Dependencies file written on the generic language.
- Output file informing only the absences and divergences.

Table 4.3: CPD’s systems used on the experiment

System	Language	LOC	Methods	Files
SIEX	Java	38665	4189	265
SIPES	Visual Basic	32887	2211	188
SIGRA	Visual Basic	29316	3184	246
SIPAT	Visual Basic	26652	2272	201
SISRU	Java	18378	1820	187
SCA	Java	8509	909	73

## 4.2 Results

Using the model explained on Section 3.1, the architecture files were created using the constraint "No vision class may depend on a persistence class" on both models to evaluated the approaches. To standardize the comparison, the conformance checking on both languages used the same dependencies file (Appendix C), the Siex’s dependencies extracted by the *javadepextractor* tool. The file has 5599 Java dependencies written on the generic language.

Through the evaluation, both Alloy and DCL were able to specify the architecture and identify a violation of constraint. Table 4.4 shows the running time found on the tests when one constraint was checked.

Table 4.4: Single constraint conformance checking tests running time

Model	Run	Execution time (ms)			Average (ms)
		Modeling	Checking	Total	
Alloy	1st	248	327	575	509
	2nd	162	319	481	
	3rd	127	344	471	
DCL	1st	-	-	83	84,33
	2nd	-	-	81	
	3rd	-	-	89	

Although both are capable of identifying the same constraint violation, Alloy only identified one dependence that violates the constraint. On DCL case, it was able to identify all dependences violating the constraint on a single command execution. DCL was also much faster on the task. The detailed architecture files are in Appendix D

Alloy conformance checking took on average 509 milliseconds to identify an architectural violation. On other hand DCL took on average 84,33 milliseconds to identify the same violation. Therefore, DCL was six times faster then Alloy. DCL has advantage on this task since it searches for textual elements, Alloy uses much of its time preparing the variables and building models to only then, analyze the properties. Other downside of Alloy is that it is not capable to identify all the violations of a constraint on a single check command.

The process was then evaluated on six industrial systems where the DCL approach was used to make the conformance analysis since it had a better performance on the preliminary evaluation. Java and Visual Basic software were checked, each system had several checking runs and the results were consistent returning the same violations every time. The architecture files of all software evaluated may be found on a Git repository<sup>1</sup>. Table 4.5 shows the compiled result.

Table 4.5: CPD systems conformance checking result

System	Language	Dependencies	Time	Violations
Siex	Java	5599	109	8
			96	8
			112	8
Sisru	Java	3367	85	1
			86	1
			86	1
Sca	Java	1318	70	5
			69	5
			71	5
Sigra	Visual Basic	3768	71	3
			68	3
			77	3
Sipat	Visual Basic	2317	68	5
			64	5
			65	5
Sipes	Visual Basic	1833	61	5
			61	5
			62	5
Libs (VB)	Visual Basic	-	-	3
			-	3
			-	3

The table shows the system name, its language, the number of dependencies on the source code extracted file and the running time of three runs for each system. When analyzing the result files, we realized that the Visual Basic systems shared some features

<sup>1</sup><https://github.com/Sigfredo/Unbconformancefiles>

which were called in external projects and libraries, for that reason there were some intersected violations. An extra row was inserted in the table specifically to identify these violations, it was named Libs (VB).

## 4.2.1 Violations

All the experiment violations were informed by the *violations.txt* file generated by the *pi-dclcheck* tool. The resulting files are transcribed below.

### Siex violation file

```

1 [absence],[br.unb.web.siex.negocio.PessoaGenericaNegocio,extend,
   icrudnegocio],[basenegocio must-extend icrudnegocio]
2 [divergence],[br.unb.web.siex.visao.EmitirHistoricoMembroVisao,access,br
   .unb.web.siex.persistencia.AlocaMembroExtensaoDAO],[only negocio,
   daofactory can-access persistencia]
3 [divergence],[br.unb.web.siex.visao.EmitirHistoricoMembroVisao,access,br
   .unb.web.siex.persistencia.MembroExtensaoDAO],[only negocio,
   daofactory can-access persistencia]
4 [divergence],[br.unb.web.siex.visao.ManterPropostaFormularioVisao,access
   ,br.unb.web.siex.persistencia.OrgaoExternoParceriaDAO],[only negocio,
   daofactory can-access persistencia]
5 [divergence],[br.unb.web.siex.endpoint.SignBatchResponseService,access,
   br.unb.web.siex.persistencia.AssinaturaDigitalDAO],[only negocio,
   daofactory can-access persistencia]
6 [absence],[br.unb.web.siex.visao.ImportarAlunoMencaoVisao,extend,
   basevisao],[visao must-extend basevisao]
7 [absence],[br.unb.web.siex.visao.LoginVisao,extend,basevisao],[visao
   must-extend basevisao]
8 [absence],[br.unb.web.siex.negocio.CadastrarUsuarioNegocio,extend,
   icrudnegocio],[basenegocio must-extend icrudnegocio]

```

Of all violations, four of them were absences, the classes did not extended the required superclass (*basenegocio* to *icrudnegocio* module and *visao* to *basevisao* module). Three divergences were found where classes from the *visao* module were accessing the *persistencia* module. This violates the constraint "*only negocio, daofactory can-access persistencia*". The last violation is given by a class from the *endpoint* package, how the package would fit on the architecture was to be defined by the architect and the development team.

## Sisru violation file

```
1 [absence], [br.unb.web.sisru.negocio.UsuarioSisruNegocio, extend,
   icrudnegocio], [basenegocio must-extend icrudnegocio]
```

Only one violation found. It was the absence of a extend requirement for the *negocio* module. It must extend a class from the *icrudnegocio* module.

## Sca violation file

```
1 [absence], [br.unb.web.sca.visao.SistemaVisao, extend, basevisao], [visao
   must-extend basevisao]
2 [absence], [br.unb.web.sca.negocio.DadosLogNegocioImpl, implement,
   basenegocio], [negocio must-implement basenegocio]
3 [absence], [br.unb.web.sca.visao.MenuVisao, extend, basevisao], [visao must-
   extend basevisao]
4 [absence], [br.unb.web.sca.visao.AcessoVisao, extend, basevisao], [visao
   must-extend basevisao]
5 [absence], [br.unb.web.sca.visao.UsuarioVisao, extend, basevisao], [visao
   must-extend basevisao]
```

Three of the four violations were absences that occurred because the *visao* classes did not extend the required *basevisao* module. The last violation was also an absence caused because the *DadosLogNegocioImpl* class did not extended a class from the *basenegocio* module.

## Sigra violation file

```
1 [divergence], [fublib.frmMensagem, access, MSMask.MaskedTextBox], [only siac can
   -access msMask]
2 [divergence], [fublibs.Biblioteca, access, Screen.MousePointer], [biblioteca
   cannot-access screen]
3 [divergence], [Siac.frmALUDIP, access, App.Path], [siac cannot-access app]
4 [divergence], [fublibs.frmMensagem, access, MSMask.MaskedTextBox], [only siac
   can-access msMask]
5 [divergence], [Siac.frmALUCMP, access, Printer.ScaleTop], [only biblioteca
   can-access printer]
6 [divergence], [Siac.frmALUCMP, access, Printer.ScaleLeft], [only biblioteca
   can-access printer]
7 [divergence], [fublib.Biblioteca, access, Screen.MousePointer], [biblioteca
   cannot-access screen]
```

Four violations were identified where *biblioteca* classes (*fublib* and *fulibs*) were accessing the *msMask* and *screen* modules, that was forbidden by the declared architecture. The other three violations occurred because the *siac* module were accessing directly the

*printer* and *app* modules, what was also forbidden, they should access *biblioteca* first.

## Sipes violation file

```
1 [divergence],[sipes.frmIngresso,access,App.Path],[sipes cannot-access
  app]
2 [divergence],[fublib.frmMensagem,access,MSMask.MaskedTextBox],[only sipes
  can-access msMask]
3 [divergence],[fublibs.Biblioteca,access,Screen.MousePointer],[biblioteca
  cannot-access screen]
4 [divergence],[fublib.Biblioteca,access,Screen.MousePointer],[biblioteca
  cannot-access screen]
5 [divergence],[sipes.frmExportaDadosGFIP,access,App.Path],[sipes cannot-
  access app]
6 [divergence],[sipes.relConstrutor,access,App.Path],[sipes cannot-access
  app]
7 [divergence],[sipes.frmGenerico,access,Printer.FontCount],[only
  biblioteca can-access printer]
8 [divergence],[fublibs.frmMensagem,access,MSMask.MaskedTextBox],[only sipes
  can-access msMask]
9 [divergence],[sipes.frmRelBeneficios,access,App.Path],[sipes cannot-
  access app]
```

The same four violations given by *fublib* and *fublibs* happend. We then realized they must be analyzed separately. The other five violations occurred because the *sipes* module were accessing directly the *printer* and *app* modules.

## Sipat violation file

```
1 [divergence],[sipat.relConstrutor,access,App.Path],[sipat cannot-access
  app]
2 [divergence],[fublib.frmMensagem,access,MSMask.MaskedTextBox],[only sipat
  can-access msMask]
3 [divergence],[fublibs.Biblioteca,access,Screen.MousePointer],[biblioteca
  cannot-access screen]
4 [divergence],[fublib.Biblioteca,access,Screen.MousePointer],[biblioteca
  cannot-access screen]
5 [divergence],[sipat.frmRelHistoricoOS,access,Printer.PaperSize],[only
  biblioteca can-access printer]
6 [divergence],[fublibs.frmMensagem,access,MSMask.MaskedTextBox],[only sipat
  can-access msMask]
7 [divergence],[sipat.frmGenerico,access,Printer.FontCount],[only
  biblioteca can-access printer]
```



Taking aside the four library violations, the three remaining were given by the *sipat* module accessing *app* and *printer*, this kind of violation was recurring on the visual basic systems.

### 4.3 Research Questions Analysis

**RQ1 - Is the proposed process capable of identifying architecture violations on different software languages?**

Through the experiment it was possible to identify several architecture violations using different techniques (Alloy and DCL) on software with different programming languages. The results are shown on Section 4.2 and the results are summarized by Table 4.5.

**RQ2 - Which tool is more suited to identify violations on software ecosystems with different programming languages?**

In Chapter 2, several approaches characteristics were analyzed and compared. DCL and Alloy seemed the best options, an empiric evaluation took place through the process execution. Alloy is able to work with huge size models but the complexity of such models increase accordingly with the number of signatures. The main negative point of the use of Alloy to make software architecture conformance is its inability of identifying the relations who violated the constraints in a textual way, instead it shows a model which can be analyzed, the counterexample model generated contain some useless elements what might difficult its analysis and also if two or more dependencies violate a constraint, the alloy model is unable to show them all, it shows one at a time. It takes much more time to fix the code defects in that way. Other negative point of Alloy is its running time. It takes long to prepare the variables and looking for a counterexample, the DCL approach is specialized on the software architecture conformance analysis through static textual checking, for that reason it is much faster. DCL is much simpler, its declarative syntax is easy to learn and understand and has a better way of exhibiting the result.

Alloy is a powerful tool with much more uses and structure checking possibilities then DCL but the transformation of a simple textual searching into a SAT problem adds a layer of complexity to Alloy's approach.

Finally the answer is: DCL. It is a better choice to software architecture conformance analysis on software ecosystems with different programming languages because:

- Its syntax is simpler;
- It is faster;
- Shows all the violations at the same time;

## 4.4 Threats to Validity

This dissertation proposed a process capable of identifying software architectural constraints violation on ecosystems with different programming language. Although the evaluation showed that the process reaches its objectives, our evaluation was made over only two different languages, Java and Visual Basic. We investigated if the process was capable of identifying violations regardless of its quantity and as we found out the number was very low. The impact of the violations found were not well studied, we still need to gather information about the software code history to understand how critical are the violations found. It is also important to check how the maintenance on the evaluated systems are affected by the removal of the violations.

## 4.5 Critical Analysis

On the efforts of decreasing the costs of software maintenance, this dissertation brought a considered return to the Informatics Center. Not only because the violations identified provides the development team with information to fix the code increasing its quality and lowering its complexity, but also because through the development of the dissertation, a healthy discussion arouse on the Informatics Center. How the architecture should be modeled, the importance of the documentation, the development of architecture views and documentation update. The awareness given by the discussion is an unquestionable increase of value. We also analyzed and used several tools related to code analysis and conformance checking, that made us expand our understanding of the techniques and important points to look on the source code. Programs were created to automatize the process of conformance checking and also to extract dependencies on the systems source code. We were able to identify several code violations and warn the development team about it. That allows the team to decrease the architecture erosion existent. The number of violations were not very worrisome but as we found out, the process performance has a strict relation with the architect expertise and the documentation quality. Nevertheless, the technique brought here is still very relevant and might after the initial run, be automatized to keep an architecture enforcement policy.

# Chapter 5

## Conclusion

The costs of keeping a software alive increase as it ages. It is possible that it reaches over 85% of information services costs [16]. It is crucial to the information service provider to keep the engineering process well defined and supervised as it is capable of minimize those costs. That is not always the case, for several reasons the software gets diverted from its intended concept. Techniques capable of finding the software's source code flaws are important to keep the costs lower. If they are also capable of automatic identify these flaws, the supervision task gets easier and more efficient. The problem gets harder to attack when the software environment is heterogeneous, where there is several developing platforms and/or programming languages.

In this work it is proposed a software architecture conformance analysis process to overcome this problem. The conceptual architecture and its constraints are used to search for codes violations in the software. The process is independent of the software programming language and is capable of finding the architectural divergences and absences in different software using the same technique.

An investigation on how the Alloy language is capable of analyze the software architecture conformance was made. As it was found out, the Alloy is capable of doing this task but on the context used to evaluate the process, the DCL language seemed more suitable, mostly because it is a specialized conformance checking technique. It is simpler and faster than Alloy. The Alloy language might be more useful on different cases like dynamic conformance checking or to evaluate software metrics which DCL is not capable of doing. Nevertheless, the proposed process framework enables the use of any conformance language and tool chosen for the specific context being analyzed.

The process evaluation was made over 6 CPD's systems which had 2 different developing platforms, Visual Basic and Java. The DCL constraint language and the *pi-dclcheck* were used to define and analyze the software architecture conformance. The fact that there was no architect role on CPD, filling by an experienced developer but not a special-

ist, arouse an understanding: the process violation recognition is strongly attached to the architect's capacity of defining the architectural constraints. Anyhow, this is a problem inherent to the software engineering field and is shared to most of the software architecture conformance approaches like DCL, DSM, SCQL and Checkstyle custom checks. Nevertheless the process was able to use different conformance checking tools to capture architectural violation and also it was able to identify them on all systems analyzed (31 different violations on 6 systems). An interest finding was that the violation number were proportional to how well the system was documented. The Sisru system which had the lowest number of violations found (only 1) had the most extensive and up to date documentation. On the opposed side, the Siex had the highest number of violations found and it was the system which documentation were more diverted from the actual features found in the source code. The Visual Basic codes were not so well understood and for that, the architectural erosion might be larger then what the process output shows.

## 5.1 Future Work

Software architecture recovering techniques like the use of data clustering and software visualization [39] might increase the quality of the conceptual architecture modeling decreasing that way, the number of the output false negatives. To improve the architectural conformance quality, some improvements might be investigated. For instance, the DCL 2.0 has some improvements over DCL related to specification, verification, reuse and more [42]. Some of the software architecture of the systems studied were poorly defined and known. As so, the use of conformance techniques that are able to dig for architectural information like Archlint is able to do on repository history, might improve the architecture and constraints definition. The Alloy models are powerful but still complex. To make them more competitive, new forms of modeling the software architecture might be investigated on the intent of lowering its complexity and increasing the kind of constraints checked. Alloy might also be used when other kind of analysis are necessary, for example it is able to check for software metrics, code quality and dynamic constraints.

# Referências

- [1] Alloy a language & tool for relational models. <http://alloy.mit.edu/alloy/index.html>. Accessed: 2017-03-01. 11
- [2] CQLinq Syntax cqlinq documentation. <http://www.ndepend.com/docs/cqlinq-syntax>. Accessed: 2017-08-01. 9, 13, 14
- [3] DCL Suite dependency constraint language suite manual. <http://aserg.labsoft.dcc.ufmg.br/dclsuite/>. Accessed: 2017-04-10. x, 10
- [4] javadepextractor github project. <https://github.com/rterrabh/javadepextractor>. Accessed: 2017-07-20. 57
- [5] LDM lattix dependency manager. <http://lattix.com/>. Accessed: 2017-08-01. 14
- [6] Vbdepend vb6/vba static analysis and code quality tool. <http://www.vbdepend.com/>. Accessed: 2017-08-01. 14
- [7] vbdepextractor github project. <https://github.com/Sigfredo/vbdepextractor>. Accessed: 2017-08-01. 58
- [8] Bernd Amann and Michel Scholl. Gram: a graph data model and query languages. In *Proceedings of the ACM conference on Hypertext*, pages 201–211. ACM. 9
- [9] Simon Austin, Andrew Baldwin, Baizhan Li, and Paul Waskett. Analytical design planning technique (adept): a dependency structure matrix tool to schedule the building design process. In *Construction Management & Economics*. Taylor & Francis. 7
- [10] Oliver Burn. Checkstyle homepage. URL <http://checkstyle.sourceforge.net/>. last accessed in March, 2005. 14
- [11] Paul Clements, David Garlan, Len Bass, Judith Stafford, Robert Nord, James Ivers, and Reed Little. *Documenting software architectures: views and beyond*. Pearson Education. 4
- [12] Michelle L Crane and Juergen Dingel. Runtime conformance checking of objects using alloy. In *Electronic Notes in Theoretical Computer Science*. Elsevier. 13
- [13] Oege De Moor, Mathieu Verbaere, and Elnar Hajiyev. Keynote address: ql for source code analysis. In *Source Code Analysis and Manipulation, 2007. SCAM 2007. Seventh IEEE International Working Conference on*, pages 3–16. IEEE. 9

- [14] Slawomir Duszynski, Jens Knodel, and Mikael Lindvall. Save: Software architecture visualization and evaluation. In *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*, pages 323–324. IEEE. x, 7, 8
- [15] David Emery and Rich Hilliard. Every architecture description needs a framework: Expressing architecture frameworks using iso/iec 42010. In *Software Architecture, 2009 & European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on*, pages 31–40. IEEE. 13
- [16] Len Erlikh. Leveraging legacy system dollars for e-business. In *IT professional*. IEEE. 40
- [17] Martin Fowler. Technical debt. <https://martinfowler.com/bliki/TechnicalDebt.html>, 2003. 1
- [18] Herve Gallaire, Jack Minker, and Jean-Marie Nicolas. Logic and databases: A deductive approach. In *ACM Computing Surveys (CSUR)*. ACM. 9
- [19] David Garlan, Robert T Monroe, and David Wile. Acme: Architectural description of component-based systems. In *Foundations of component-based systems*. 12
- [20] David Garlan and Mary Shaw. An introduction to software architecture. In *Advances in software engineering and knowledge engineering*. Singapore. 4
- [21] Abram Hindle and Daniel M German. *SCQL: A formal model and a query language for source control repositories*, volume 30. ACM. 9, 14
- [22] Hongxin Hu and GailJoon Ahn. Enabling verification and conformance testing for access control model. In *Proceedings of the 13th ACM symposium on Access control models and technologies*, pages 195–204. ACM. 13
- [23] Daniel Jackson. Alloy: a lightweight object modelling notation. In *ACM Transactions on Software Engineering and Methodology (TOSEM)*. ACM. 11, 14, 46
- [24] Daniel Jackson. Automating first-order relational logic. In *ACM SIGSOFT Software Engineering Notes*, volume 25, pages 130–139. ACM. 11
- [25] Daniel Jackson, Ian Schechter, and Hya Shlyachter. Alcoa: the alloy constraint analyzer. In *Proceedings of the 22nd international conference on Software engineering*, pages 730–733. ACM. 12
- [26] JV Joshua, DO Alao, SO Okolie, and O Awodele. Software ecosystem: Features, benefits and challenges. In *International Journal of Advanced Computer Science and Applications*. 2
- [27] Jung Soo Kim and David Garlan. Analyzing architectural styles with alloy. In *Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*, pages 70–80. ACM. 13

- [28] Jens Knodel, Dirk Muthig, Uwe Haury, and Gerald Meier. Architecture compliance checking—experiences from successful technology transfer to industry. In *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, pages 43–52. IEEE. 7, 13, 14
- [29] Jens Knodel, Dirk Muthig, and Dominik Rost. Constructive architecture compliance checking—an experiment on support by live feedback. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 287–296. IEEE. 5, 13
- [30] Jens Knodel and Daniel Popescu. A comparison of static architecture compliance checking approaches. In *Software Architecture, 2007. WICSA'07. The Working IEEE/IFIP Conference on*, pages 12–12. IEEE. 13
- [31] Philippe B Kruchten. The 4+ 1 view model of architecture. In *IEEE software*, volume 12, pages 42–50. IEEE. 4
- [32] Alan MacCormack, John Rusnak, and Carliss Y Baldwin. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. In *Management Science*. INFORMS. 7
- [33] Cristiano Maffort, Marco Tulio Valente, Mariza AS Bigonha, Leonardo H Silva, and Gladston Junio Aparecido. Archlint: Uma ferramenta para detecção de violações arquiteturais usando histórico de versões. In *Congresso Brasileiro de Software: Teoria e Prática (CBSOFT), Sessão de Ferramentas*, 2013. 14
- [34] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying distributed software architectures. In *Software Engineering—ESEC'95*. Springer. 12
- [35] Nenad Medvidovic and Richard N Taylor. Software architecture: foundations, theory, and practice. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering—Volume 2*, pages 471–472. ACM. 4
- [36] Paulo Merson. Ultimate architecture enforcement: custom checks enforced at code-commit time. In *Proceedings of the 2013 companion publication for conference on Systems, programming, & applications: software for humanity*, pages 153–160. ACM. 13
- [37] Gail C Murphy and David Notkin. Reengineering with reflexion models: A case study. In *Computer*. IEEE. 6, 7
- [38] Gail C Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *ACM SIGSOFT Software Engineering Notes*. ACM. x, 6, 7
- [39] Renato Paiva, Genaína N Rodrigues, Rodrigo Bonifácio, and Marcelo Ladeira. Exploring the combination of software visualization and data clustering in the software architecture recovery process. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 1309–1314. ACM. 41
- [40] David Lorge Parnas. Software aging. In *Proceedings of the 16th international conference on Software engineering*, pages 279–287. IEEE Computer Society Press. 1

- [41] Dewayne E Perry and Alexander L Wolf. Foundations for the study of software architecture. In *ACM SIGSOFT Software engineering notes*. ACM. 4, 5
- [42] Henrique Rocha, Rafael Serapilha Durelli, Ricardo Terra, Sândalo Bessa, and Marco Tulio Valente. Dcl 2.0: Modular and reusable specification of architectural constraints. In *Journal of the Brazilian Computer Society*. 41
- [43] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. In *ACM Sigplan Notices*, volume 40, pages 167–176. ACM. 1, 7, 14
- [44] Donald V Steward. The design structure system: A method for managing the design of complex systems. In *IEEE transactions on Engineering Management*. IEEE. 7
- [45] Kevin J Sullivan, William G Griswold, Yuanfang Cai, and Ben Hallen. The structure and value of modularity in software design. In *ACM SIGSOFT Software Engineering Notes*, volume 26, pages 99–108. ACM. 7
- [46] Ricardo Terra. Conformação arquitetural utilizando restrições de dependência entre módulos. In *XXIII Concurso de Teses e Dissertações (CTD)*. 9, 14
- [47] Ricardo Terra and Marco Tulio de Oliveira Valente. Towards a dependency constraint language to manage software architectures. In *European Conference on Software Architecture*, pages 256–263. Springer. 9, 10
- [48] Ricardo Terra and Marco Tulio Valente. A dependency constraint language to manage object-oriented software architectures. In *Software: Practice and Experience*. Wiley Online Library. 9, 14
- [49] Christopher J Turner, TC Nicholas Graham, Christopher Wolfe, Julian Ball, David Holman, Hugh D Stewart, and Arthur G Ryman. Visual constraint diagrams: Runtime conformance checking of uml object models versus implementations. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 271–276. IEEE. 13



# Appendix A

## Alloy Syntax

The Alloy language mentioned on Chapter 2 is based on the Z formal language and uses the concepts of the set theory, it is considered a lightweight formal language and is capable of defining and analyzing automatically structures and its relations [23].

### A.1 Alloy Grammar

The Alloy grammar <sup>1</sup> is seen below:

```
1 Core Alloy4 Syntax (minus some obscure compatibility syntax retained for
  Alloy3)
2 =====
3
4 Precedence (from LOW to HIGH)
5
6 1)  let      all a:X|F    no a:X|F    some a:X|F    lone a:X|F    one a:x|F
      sum a:x|F
7 2)  ||
8 3)  <=>
9 4)  =>      => else
10 5)  &&
11 6)  !
12 7)  in      =          <          >          <=         >=         !in      !=       !<
      !>      !<=      !>=
13 8)  no X    some X    lone X    one X    set X    seq X
14 9)  <<      >>      >>>
15 10) +      -
16 11) #X
17 12) ++
18 13) &
```

<sup>1</sup>Alloy grammar documentation: <http://alloy.mit.edu/alloy/documentation/alloy4-grammar.txt>

```

19 14)  ->
20 15)  <:
21 16)  :>
22 17)  []
23 18)  .
24 19)  ~ * ^
25
26 All binary operators are left-associative, except the arrow operators
    (->),
27 the implication (a=>b), and if-then-else (a=>b else c).
28
29 =====
30
31 specification ::= [module] open* paragraph*
32
33 module ::= "module" name [ "[" ["exactly"] name ("," ["exactly"] num)
    * "]" ]
34
35 open ::= ["private"] "open" name [ "[" ref,+ "]" ] [ "as" name ]
36
37 paragraph ::= factDecl | assertDecl | funDecl | cmdDecl | enumDecl |
    sigDecl
38
39 factDecl ::= "fact" [name] block
40
41 assertDecl ::= "assert" [name] block
42
43 funDecl ::= ["private"] "fun" [ref "."] name "(" decl,* ")" ":" expr
    block
44 funDecl ::= ["private"] "fun" [ref "."] name "[" decl,* "]" ":" expr
    block
45 funDecl ::= ["private"] "fun" [ref "."] name ":" expr
    block
46
47 funDecl ::= ["private"] "pred" [ref "."] name "(" decl,* ")" block
48 funDecl ::= ["private"] "pred" [ref "."] name "[" decl,* "]" block
49 funDecl ::= ["private"] "pred" [ref "."] name block
50
51 cmdDecl ::= [name ":"] ("run"|"check") (name|block) scope
52
53 scope ::= "for" number ["expect" (0|1)]
54 scope ::= "for" number "but" typescope,+ ["expect" (0|1)]
55 scope ::= "for" typescope,+ ["expect" (0|1)]
56 scope ::= ["expect" (0|1)]
57

```

```

58 typescope ::= ["exactly"] number [name|"int"|"seq"]
59
60 sigDecl ::= sigQual* "sig" name,+ [sigExt] "{" decl,* "}" [block]
61
62 enumDecl ::= "enum" name "{" name ("," name)* "}"
63
64 sigQual ::= "abstract" | "lone" | "one" | "some" | "private"
65
66 sigExt ::= "extends" ref
67 sigExt ::= "in" ref ["+" ref]*
68
69 expr ::= "let" letDecl,+ blockOrBar
70         | quant decl,+ blockOrBar
71         | unOp expr
72         | expr binOp expr
73         | expr arrowOp expr
74         | expr ["!"|"not"] compareOp expr
75         | expr ("=>"|"implies") expr "else" expr
76         | expr "[" expr,* "]"
77         | number
78         | "-" number
79         | "none"
80         | "iden"
81         | "univ"
82         | "Int"
83         | "seq/Int"
84         | "(" expr ")"
85         | ["@"] name
86         | block
87         | "{" decl,+ blockOrBar "}"
88
89 decl ::= ["private"] ["disj"] name,+ ":" ["disj"] expr
90
91 letDecl ::= name "=" expr
92
93 quant ::= "all" | "no" | "some" | "lone" | "one" | "sum"
94
95 binOp ::= "||" | "or" | "&&" | "and" | "&" | "<=>" | "iff"
96         | "=>" | "implies" | "+" | "-" | "++" | "<:" | ">:" | "." | "<<"
97         | ">>" | ">>>"
98
99 arrowOp ::= ["some"|"one"|"lone"|"set"] "->" ["some"|"one"|"lone"|"set"]
100
101 compareOp ::= "=" | "in" | "<" | ">" | "=<" | ">="



```

```
102 unOp ::= "!" | "not" | "no" | "some" | "lone" | "one" | "set" | "seq" |  
      "#" | "~" | "*" | "^"  
103  
104 block ::= "{" expr* "  
105  
106 blockOrBar ::= block  
107  
108 blockOrBar ::= "|" expr  
109  
110 name ::= ("this" | ID) ["/" ID]*  
111  
112 ref ::= name | "univ" | "Int" | "seq/Int"
```

# Appendix B

## CPD Architectural Documentation

The evaluation on Chapter 4 used the CPD's architectural documentation, there was no Visual Basic and Java's layer views, for that reason the software architect got support related to the content needed on the view and how to create it. The resulting documentation is annexed below:

 	Architectural Documentation	VARQDES	
	Layer View	01	

## 1. Java Layer Architecture View



This document contains the Layer architecture view of the Java systems using the Fast Framework platform.

## 2. Modules and Constraints

<b>Summary</b>	The Fast Framework Platform uses 3 main layers to control the data flow: Visao, Negocio and Persistencia. Some other layers give support and encapsulate common features. The communication between layers are restricted as shown in the attached diagram.
<b>System Layers</b>	Visao Negocio Persistencia Pojo Vo
<b>System Module Division</b>	visao: br.unb.web.siex.visao.(all name ending with: "Visao") basevisao: br.unb.fast.core.camada.visao.BaseVisao negocio: br.unb.web.siex.negocio.(all name ending with: "NegociImpl") basenegocio: br.unb.web.siex.negocio.(all name ending with: "Negocio") icrudnegocio: br.unb.fast.core.camada.negocio.ICrudNegocio basepersistencia: br.unb.web.sca.(all name ending with: "DAO") persistencia: br.unb.web.sca.persistencia.(all name ending with: "DAOImpl") daofactory: br.unb.web.siex.persistencia.DAOFactory pojo: br.unb.web.siex.pojo.* vo: br.unb.web.siex.vo.* sql: java.sql.*
<b>Contraints</b>	<ul style="list-style-type: none"> <li>• All visao classes must implement a basevisao class.</li> <li>• All negocio classes must implement a basenegocio class.</li> <li>• All basenegocio classes must implement a icrudnegocio class.</li> <li>• The persistencia module classes can only be instanciated by classes from the modules: negocio and daofactory</li> <li>• The sql module can only be accessed by the persistence module.</li> <li>• The module negocio cannot access visao but visao can access negocio.</li> <li>• the modules basepersistencia and persistencia can only be accessed by classes from the modules: negocio and daofactory</li> <li>• All persistencia classes must implement a basepersistencia class.</li> </ul>

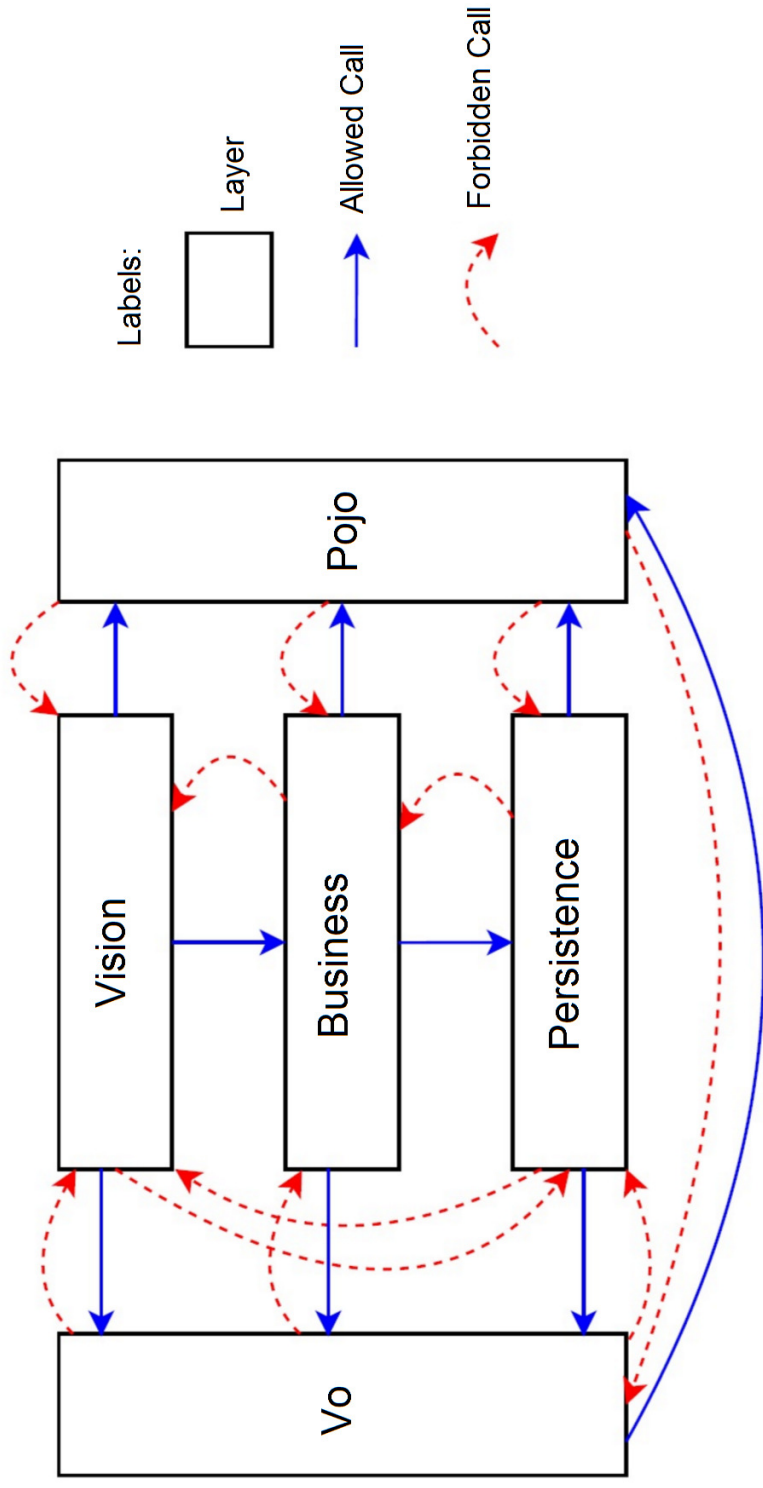
Architectural Documentation - Layer View - Layer Constraints

2017LayerView..CPD.UnB.doc	<b>Development Team CDP/ UnB</b> Universidade de Brasília – UnB – Brasília – DF Phone: (61) 3307 2383/e-mail: gtsistemas@unb.br	08/09/2017	Page 1
----------------------------	---	------------	--------


 	Architectural Documentation	VARQDES	
	Layer View	01	

	<ul style="list-style-type: none"> <li>• The module persistencia cannot access the visao and negocio modules.</li> <li>• The module pojo cannot access any other module but all the modules can access pojo.</li> <li>• The module vo can access only the module pojo but all other modules except pojo can access vo</li> </ul>
<b>Last revision:</b>	07/31/2017

# Fast Framework Platform Architecture Layer View







	Architectural Documentation	VARQDES	
	Layer View - Visual Basic	02	

## 1. Visual Basic Layer Architecture View

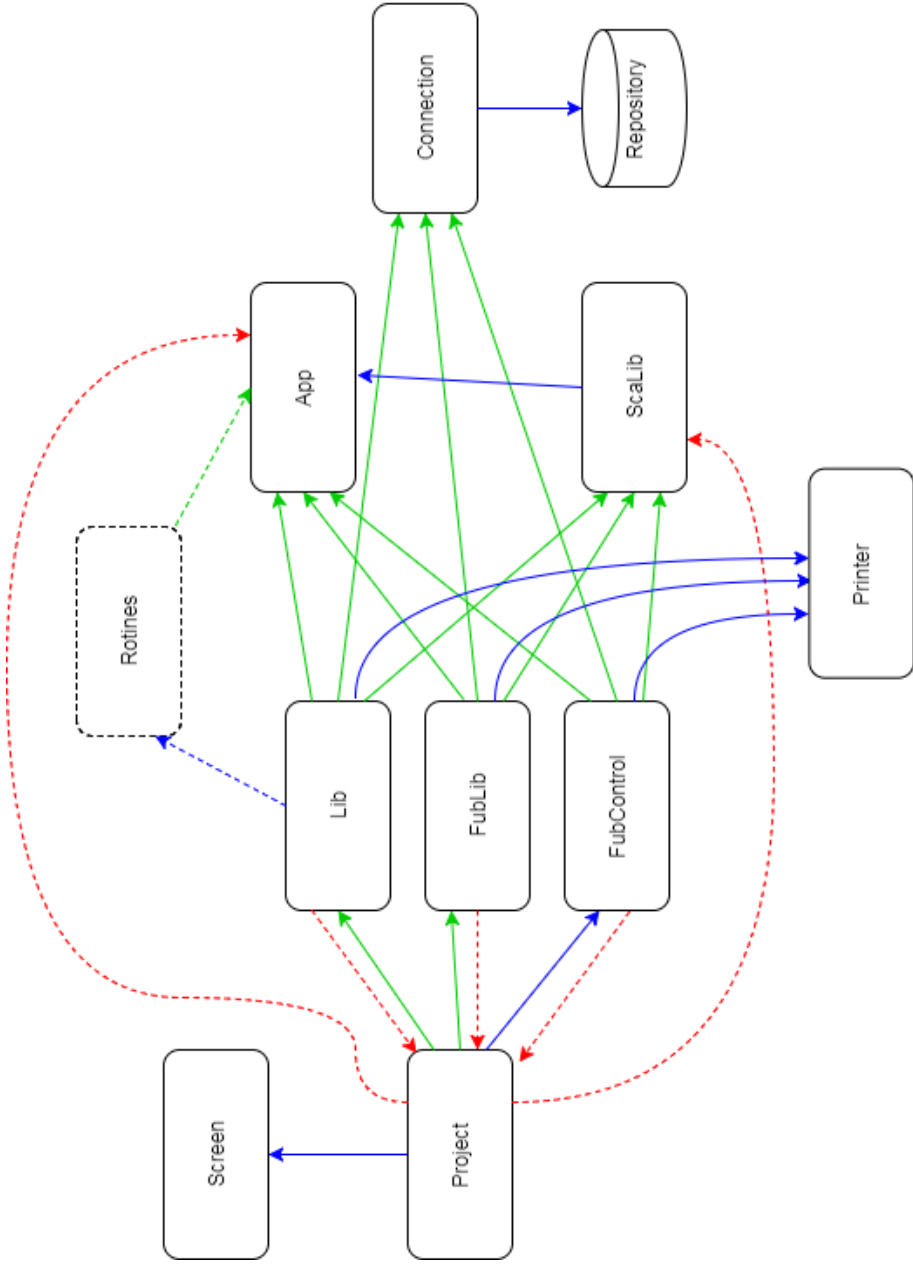
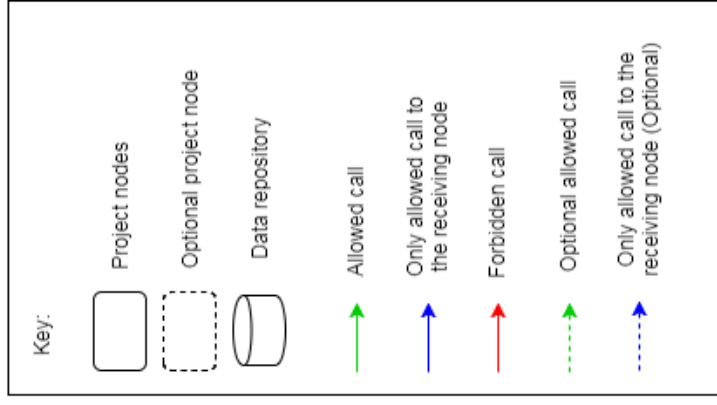
This document contains the Layer architecture view of the Visual Basic systems using the Fubcontrol Libraries.

## 2. Modules and Constraints

<b>Summary</b>	The CPD's Visual Basic systems uses several libraries to control an encapsulate some functionalities. Also some support systems are used to some features reuse. The communication between the project and these libraries and support systems is shown in the attached diagram.
<b>Observation</b>	Where it says [project_name], the name of the project system being analyzed shall be used.
<b>System Layers</b>	[project_name] [project_name]Lib [project_name]Rotinas (optional) Fublib Scalib VB
<b>System Module Division</b>	fubcontrol: prjCtrl.fubControl fublib: Fublib.* fublibs: fublibs.* scalib: scalib.* scalibs: scalibs.* [project_name]: [project_name].* [project_name]lib: [project_name]Lib.* [project_name]rotinas: [project_name]Rotinas.* sitablib: sitablib.* vb: VB.* screen: Screen.*, VB.Frame, VB.Form msMask: MSMask.* widowState: *.WindowState app: App.Path biblioteca: fublib.Biblioteca, fublibs.Biblioteca, fublib.Biblioteca, fublibs.Biblioteca, [project_name]rotinas.Biblioteca printer: Printer.*
<b>Contraints</b>	<ul style="list-style-type: none"> <li>• The [project_name] module cannot access the scalib neither the scalibs module.</li> <li>• The msMask, widowState and fubControl modules can only be accessed by the [project_name] module</li> <li>• The [project_name]rotinas module can only be accessed by the [project_name]lib module.</li> </ul>

 	Architectural Documentation	VARQDES	
	Layer View - Visual Basic	02	

	<ul style="list-style-type: none"> <li>• The app module cannot be accessed by the [project_name] module.</li> <li>• screen module can be accessed by any module except the biblioteca module.</li> <li>• The printer module can only be accessed by the biblioteca module.</li> <li>• The biblioteca module cannot access the [project_name] module but the [project_name] module can access the biblioteca module.</li> </ul>
<b>Last revision:</b>	07/31/2017



# Appendix C

## Dependencies Extraction Output

On Section 3.2.1 it is mentioned how the dependencies extraction occurs and in Chapter 4 this task is executed. The code must be analyzed to extract the relations between classes, the output must be given in the generic language. Some tools are used to that task, *javadepextractor* are used on the Java systems and the *vbdepextractor* on the Visual Basic systems.

### C.1 Javadepextractor

The *javadepextractor* tool is capable of extracting the Java code dependencies on the generic language syntax [4]. To better understanding about the tool, this appendix exhibits part of the output file generated after the extraction of the Siex system. The full file has 5599 lines each one containing a dependency between two of the software's classes.

```
1 br.unb.web.siex.visao.ManterPropostaListagemVisao ,access ,br.unb.web.siex
   .negocio.AlocaMembroExtensaoNegocio
2 br.unb.web.siex.persistencia.CertificadoTemaEquipeViewDAOImpl ,access ,
   java.sql.Connection
3 br.unb.web.siex.negocio.PessoaNegocio ,declare ,int
4 br.unb.web.siex.negocio.CadastrarUsuarioNegocioImpl ,declare ,br.unb.web.
   siex.persistencia.UsuarioDAO
5 br.unb.web.siex.visao.AbstractLoginPublicoVisao ,extend ,br.unb.fast.core.
   camada.visao.BaseVisao
6 br.unb.web.siex.visao.ArquivosAnexosVisao ,create ,java.io.File
7 br.unb.web.siex.negocio.AcaoPossuiSituacaoNegocioImpl ,declare ,br.unb.
   fast.core.pagina.Pagina
8 br.unb.web.siex.visao.AlocarMembroVisao ,create ,br.unb.web.siex.pojo.
   AlocaMembroExtensao
9 br.unb.web.siex.negocio.PessoaGenericaNegocio ,declare ,int
```

```

10 br.unb.web.siex.visao.EnviarLoteAssinatura ,declare , java.lang.String
11 br.unb.web.siex.visao.EmitirRelatorioColaboradoresVisao ,declare , br.unb.
    fast.core.pagina.Pagina
12 br.unb.web.siex.pojo.Acesso ,declare , java.lang.Integer
13 br.unb.web.siex.negocio.PessoaNegocio ,declare , long
14 br.unb.web.siex.visao.InscricoesConfirmadasVisao ,declare , int
15 br.unb.web.siex.vo.RelatorioAlocaMembro ,access , br.unb.web.siex.vo.
    RelatorioAlocaMembro
16 br.unb.web.siex.visao.ManterMembrosExtensaoVisao ,declare , br.unb.web.
    sitab.negocio.ManterPaisNegocio
17 br.unb.web.siex.pojo.AcaoPossuiSituacao ,extend , br.unb.fast.core.camada.
    persistencia.AbstractPojo

```

## C.2 Vbdepextractor

The vbdepextractor is capable of extrating the CPD's Visual Basic systems dependencies on the generic output [7]. The full file has 3768 lines each one containing one dependence between two classes. The transcription of part of the file generated by the tool when extracting the Sigra system dependencies is shown below.

```

1 Siac.frmSelecaoCurriculo ,access , Me.Visible
2 fublib.Biblioteca ,access , objOption.TabIndex
3 Siac.frmOCOETF ,access , FubLib.FormatarParLog
4 Siac.frmDISEQV ,access , TabDlg.SSTab
5 Siac.frmHPEDNA ,access , VB.Label
6 Siac.frmSelecaoPlanoEnsino ,access , Me.Visible
7 Siac.frmCURIEC ,access , VB.OptionButton
8 Siac.frmTRAALU ,access , prjCtrl.fubControl
9 Siac.frmOFEDLP ,access , VB.OptionButton
10 Siac.frmDISPRQ ,access , Screen.MousePointer
11 Siac.frmALUFUN ,access , SiacLib.AbandonarALUFUN
12 Siac.frmALUDIS ,access , VB.Form
13 Siac.frmDECPEC ,access , SiacLib.CriarAcompanhamentoAcademico
14 Siac.frmALUFUN ,access , VB.Label
15 Siac.frmDESENC ,access , SiacLib.CriarExameNacionalCursos
16 Siac.frmPRECGP ,access , Screen.MousePointer
17 Siac.frmHPEDNA ,access , FubLib.IniciarRelatorio
18 Siac.frmPREOCO ,access , Screen.MousePointer
19 Siac.frmOCOEXT ,access , MSMask.MaskedTextBox
20 Siac.frmSelecaoOrientador ,access , VB.Form
21 Siac.frmTRAQAP ,access , VB.Frame
22 Siac.frmFORATA ,access , Screen.MousePointer

```

```
23 Siac.frmCRRROD ,access ,MSMask.MaskedTextBox
24 Siac.frmCUREGP ,access ,VB.Frame
25 Siac.frmOFEANL ,access ,VB.TextBox
26 Siac.frmDECFPE ,access ,FubLib.AtivarForm
27 Siac.frmMATBDC ,access ,Me.Top
28 Siac.frmCRRMAT ,access ,FubLib.FormatarParLog
29 Siac.frmDADHOM ,access ,VB.Form
30 Siac.frmDECBEX ,access ,Me.txtAluNome
```

# Appendix D

## Conceptual Architecture Files

This appendix exhibits examples of the conceptual architecture files mentioned on Chapter 3 and used on the evaluation on Chapter 4. The academic extension system (Siex) developed by CPD is specified both on Alloy and DCL syntax to exemplify how is files are formed. All the files used on the evaluation are disposed on the Git repository<sup>1</sup>.

### D.1 Siex Architecture DCL File

*architecture.dcl* file containing the Siex system architecture on DCL syntax:

```
1 module $sql: java.sql
2 module visao: "br.unb.web.siex.visao.[a-zA-Z0-9/.]*Visao"
3 module basevisao: br.unb.fast.core.camada.visao.BaseVisao
4 module negocio: "br.unb.web.siex.negocio.[a-zA-Z0-9/.]*NegocioImpl"
5 module basenegocio: "br.unb.web.siex.negocio.[a-zA-Z0-9/.]*Negocio"
6 module icrudnegocio: br.unb.fast.core.camada.negocio.ICrudNegocio
7 module basepersistencia: "br.unb.web.siex.persistencia.[a-zA-Z0-9/.]*DAO
  "
8 module persistencia: "br.unb.web.siex.persistencia.[a-zA-Z0-9/.]*DAOImpl
  "
9 module daofactory: br.unb.web.siex.persistencia.DAOFactory
10 module pojo: br.unb.web.siex.pojo.*
11 module vo: br.unb.web.siex.vo.*
12
13
14
15 visao must-extend basevisao
16 negocio must-implement basenegocio
17 only negocio, daofactory can-create persistencia
```

---

<sup>1</sup><https://github.com/Sigfredo/Unbconformancefiles>

```

18 basenegocio must-extend icrudnegocio
19 only persistencia can-access $sql
20 negocio cannot-access visao
21 only negocio, daofactory can-access persistencia
22 only negocio, daofactory can-access basepersistencia
23 persistencia must-implement basepersistencia
24 persistencia cannot-access visao, negocio
25 pojo cannot-access visao, negocio, persistencia, vo
26 vo cannot-access visao, negocio, persistencia

```

## D.2 Siex Architecture Alloy File

Screenshot taken from part of the *architecture.als* file containing the Siex system architecture on Alloy syntax:

```

1 sig Objeto{}
2 sig visao, basevisao, negocio, basenegocio, icrudnegocio, daofactory, persistencia, pojo, vo, sql {classes: se
3 sig br_unb_web_siex_visao_ManterPropostaListagemVisao, br_unb_web_siex_negocio_AlocaMembroExtensaoNegocio, br_u
4 sig Access, Declare, Create, Extend, Implement, Throw, Useannotation{r: set Objeto -> Objeto}
5
6 fact {sql.classes = java_sql_Connection + java_sql_ResultSet + java_sql_PreparedStatement + java_sql_Blob + jav
7 fact {visao.classes = br_unb_web_siex_visao_ManterPropostaListagemVisao + br_unb_web_siex_visao_AbstractLoginPu
8 fact {basevisao.classes = br_unb_fast_core_camada_visao_BaseVisao + br_unb_fast_web_primefaces_visao_BaseVisaoW
9 fact {negocio.classes = br_unb_web_siex_negocio_CadastrarUsuarioNegocioImpl + br_unb_web_siex_negocio_AcaoPossu
10 fact {basenegocio.classes = br_unb_web_siex_negocio_AlocaMembroExtensaoNegocio + br_unb_web_siex_negocio_Pessoa
11 fact {icrudnegocio.classes = br_unb_fast_core_camada_negocio_ICrudNegocio}
12 fact {daofactory.classes = br_unb_web_siex_persistencia_DAOFactory}
13 fact {persistencia.classes = br_unb_web_siex_persistencia_CertificadoTemaEquipeViewDAOImpl + br_unb_web_siex_pe
14 fact {pojo.classes = br_unb_web_siex_pojo_AlocaMembroExtensao + br_unb_web_siex_pojo_Acesso + br_unb_web_siex_p
15 fact {vo.classes = br_unb_web_siex_vo_RelatorioAlocaMembro + br_unb_web_siex_vo_AvaliarTurmaVO + br_unb_web_sie
16
17 fact {Access.r = br_unb_web_siex_visao_ManterPropostaListagemVisao -> br_unb_web_siex_negocio_AlocaMembroExtens
18
19 fact {Declare.r = br_unb_web_siex_negocio_PessoaNegocio -> int_var + br_unb_web_siex_negocio_CadastrarUsuarioNe
20
21 fact {Create.r = br_unb_web_siex_visao_ArquivosAnexosVisao -> java_io_File + br_unb_web_siex_visao_AlocarMembro
22
23 fact {Extend.r = br_unb_web_siex_visao_AbstractLoginPublicoVisao -> br_unb_fast_core_camada_visao_BaseVisao + b
24
25 fact {Implement.r = br_unb_web_siex_persistencia_ManterAvaliacaoDAOImpl -> br_unb_web_siex_persistencia_ManterA
26
27 fact {Throw.r = br_unb_web_siex_visao_ManterHorarioVisao -> java_lang_Exception + br_unb_web_siex_visao_Avaliar
28
29
30 fact {Useannotation.r = br_unb_web_siex_visao_CadastrarUsuarioVisao -> javax_inject_Inject + br_unb_web_siex_pe
31
32 assert no_visao_persistencia{
33 no x: visao.classes, y: persistencia.classes | x->y in Access.r

```

Figure D.1: CPD's Java System Architecture Layer View.