Position Papers of the Federated Conference on Computer Science and Information Systems pp. 275–282
DOI: 10.15439/2016F587
ACSIS, Vol. 9. ISSN 2300-5963

Using LINQ as a universal tool for defining architectural assertions

Bartosz Frąckowiak
Institute of Informatics
University of Warsaw
Poland
Email: b.frackowiak@mimuw.edu.pl

Robert Dąbrowski
Institute of Informatics
University of Warsaw
Poland
Email: r.dabrowski@mimuw.edu.pl

Abstract—**We demonstrate that Microsoft LINQ can be used as a convenient tool to define architectural assertions. We introduce an abstract model of software based on a directed multi-graph and formalize the notion of software architecture and architectural assertions. We demonstrate how Microsoft Visual Studio can be harnessed to extract the architecture of a given software project and append it with assertions using LINQ notation. In particular we explain the flow of data processing that takes place within Visual Studio engine. We follow with examples of assertions selected to demonstrate the expressive power of our approach. We conclude by showing subsequent areas of research worth following in order to deepen the research indicated in this paper.**

I. Introduction

SOFTWARE engineering is concerned with development and maintenance of software systems. Properly engineered systems are reliable, satisfy user requirements while their development and maintenance is affordable.

In the past half-century computer scientists and software engineers have come up with numerous ideas for how to improve the discipline of software engineering. Edgser Dijkstra in his article [11] introduced structural programming which restricted imperative control flow to hierarchical structures instead of *ad-hoc* jumps. Computer programs written in this style were more readable, easier to understand and reason about. Another improvement was the introduction of the object-oriented paradigm [19] as a formal programming concept. Other improvements in software engineering included e.g. engineering pipelines and software testing.

In the early days software engineers perceived significant similarities between software and civil engineering processes. The waterfall model [23] that resembles engineering practices was widely adopted as such, even though despite its original description actually suggesting a more agile approach. It has soon turned out that building software differs from building skyscrapers and bridges. In the late 1990s the idea of extreme programming emerged [3], its key points being: keep the code simple, review it frequently and test early and often. Among numerous techniques, test-driven development was promoted, which eventually resulted in increased quality of produced software and the stability of the development process [14]. Contemporary development teams started to lean towards short iterations (sprints) rather than fragile upfront designs, and short feedback loops allowed customers' opinions to provide timely influence on software development. This allowed for creating even more complex software. Growing complexity of software required ability to describe the software on different levels of abstraction, and the notion of software architecture has developed. The emergence of patterns and frameworks had a similar influence on architecture as design patterns and idioms had on programming. Software became developed by assembling reusable software components, that interact using well-defined interfaces, while component-oriented frameworks and models provided tools and languages making them suitable for formal architecture design. However, a discrepancy between architecture level of abstraction and programming level of abstraction prevailed. While the programming phase remained focused on generating code within a preselected (typically object-oriented) programming language, the architecture phase took place in the disconnected component world. The discrepancies deepened as software gained features while not being properly refactored, development teams changed over time, worked under time pressure with incomplete documentation and requirements that were subject to frequent changes. Multiple development technologies, programming languages and coding standards made this situation even more severe. Unification of modeling languages failed to become the silver bullet.

The discrepancy accelerated research on software architectures, model-driven development or automated software engineering. Nowadays, we have ideas of how to craft the architecture, though we still require ways to both monitor the state of the architecture and enforce it during programming in an automated manner. This is the problem that we aim at in our reseach.

We start with a new vision for management of software architecture based on the idea of an architecture warehouse. *An architecture warehouse* is a repository of all software system and software process artifacts. Such a repository can capture architecture information which was previously only stored in design documents or simply in the minds of developers. *Software intelligence* is a tool-set for analysis and visualization of this repository's content [7], [8], [9]. That includes all tools able to extract useful information from the source code and other available artifacts (like version control history).

All software system artifacts and all software engineering process artifacts being created during a software project are represented in the repository as vertices of a *graph*. Multiple edges of this graph represent various kinds of dependencies among those artifacts. The key aspects of software production like quality, predictability, automation and metrics are then expressed in a unified way using graph-based terms.

The integration of source code artifacts and software process artifacts in a single model opens new possibilities. They include defining new metrics and qualities that take into account all architectural knowledge, not only the knowledge about source code. The state of software (the artifacts and their metrics) can be conveniently visualized on any level of abstraction required by software architects (i.e. functional level, package level) or by software programmers (i.e. class or method level).

Furthermore, the relations among those artifacts can be automatically governed, in particular by implementing the idea of assertions at the architectural level of abstraction.

In this article we introduce concepts and tools that allow architects to enforce architectural principles (constraints) upon programmers using architectural assertions, and we demonstrate their proof-of-concept implementation using Microsoft LINQ and Microsoft Visual Studio.

We introduce a new way of using internal Visual Studio components to provide a universal tool for discovering violation of architecture constraints. Typically, tools of this type provide functionality via a new standalone platform, doubling existing functionality of integrated development environments; or at best get integrated with existing environments (i.e. as their plugins). We take a different approach. Since developers spend most of their time using integrated development environments as the main tool for producing source code, we aim at reusing as much functionality of the developers' well known environment as possible. In this approach LINQ becomes a universal language for describing architectural assertions for all types of programming languages, and Visual Studio becomes a universal environment with software intelligence capabilities extending beyond its natively supported programming stack.

The paper is organized as follows. Section II briefly summarizes the works related to this research. Section III recalls the graph-based model for representing architectural knowledge that creates the backbone for architectural assertions, while Section IV describes their implementation using Visual Studio and LINQ. Section V demonstrates by example how this approach can be applied to handle selected architectural challenges. Section VI concludes.

II. Related work

In 2010 Tibermacine et al. [29] worked on a family of languages for architecture constraint specification.

They argued that during software development architectural decisions should be documented so that quality attributes guaranteed by these decisions and required in the software specification could be preserved. They stressed out that an important part of these architectural decisions is getting them formalized using constraint languages which differ between stages of the development process. Therefore they suggested a family of architectural constraint languages, where each member of the family, called a profile, was to be used to formalize architectural decisions at a given stage of the development process. All profiles were based on a certain core constraint language and a common architecture model. In addition to the family of languages, they introduced a transformation-based interpretation method for profiles and an associated tool.

In 2012 Fabresse et al. [13] have worked on bridging the gap between design and implementation. They observed that significant amount of software systems are designed in component-oriented approach but programmed in object-oriented languages. Unified Modeling Language (UML), Corba Component Model (CCM) or Enterprise Java Beans (EJB) were shown as examples of component-oriented models that were only used at design time, while implementation relied on object-oriented languages, with developers not actually adapting component-oriented programming. The authors identified decoupling, adaptability, unplanned connections, encapsulation and uniformity as important requirements for component-oriented programming and proposed a language that fulfilled these requirements, along with a prototype implementation and concrete experiments to validate their proposal.

As software evolution has become an integral part of the software lifecycle, Lytra et al. [16] focused their research on checking consistency between design decisions and design models. In 2012 they proposed a constraint-based approach for checking the consistency between the decisions and the corresponding component models. They argued that since maintenance of a software system involves among others the maintenance of the software system architecture, then software community must come up with additional models to capture architectural design decisions and their design rationale, and record the architectural knowledge. Their approach enabled explicit formalized mappings of architectural design decisions onto component models. Based on these mappings, component models along with the constraints used for consistency checking between the decisions and the component models were to be automatically generated using model-driven techniques. The approach was coping with changes in the decision model by regenerating the constraints for the component model. Thus, the component model got updated and validated as the architectural decisions evolved.

In 2013 and 2014 Spacek et al. [24], [25], [26] worked on wringing out objects for programming and for modeling of component-based systems, and bridging the gap between component-based design and implementation with a reflective programming language. They recalled that languages and technologies used to implement component-based software are not component-based, i.e. while the design phase happens in the component world, the programming phase occurs in the object-oriented world; and when an object-oriented language is used for the programming stage, then the original component-based design vanishes, because component concepts are not treated explicitly. They suggested a pure reflective, component-based programming and modeling language, where all core component concepts were treated explicitly and therefore kept the original component-based design alive. The language made it possible to model and program software using the same language, while its uniform component-based meta-model and integrated reflection capabilities aimed at making the language and its applications flexible.

III. Model

In our work we extend research summarized in sections I and II.

We follow the unified representation of software architecture as a collection of artifacts created during a software (development) process and the relations among those artifacts. We model it with a directed labeled multigraph [8].

Such model caters for the following key needs: (1) natural scalability, (2) abstraction from programming paradigms, languages, specification standards, testing approaches, etc, and (3) completeness, i.e. all software system and software process artifacts are represented.

Our goal is to ensure that the designed architecture is kept on track during the whole software process, in particular can be enforced upon programmers during software development.

We obtain this goal by harnessing Microsoft technology to deliver tools that allow to: (1) define architectural assertions in concise notation; (2) monitor breaking the assertions by programmers in automated way.

A. Architecture graph

Let *software architecture* be the structure or set of structures defined by existing software elements, and the relationships among them, best represented by *software architecure graph.*

Let *software architecture graph* be an ordered tuple:

$$(\mathcal{V}, \mathcal{E})$$

where $\mathcal{V}$ is the set of vertices that reflect design and implementation artifacts created during a software project; $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is the set of directed edges that represent dependencies (relationships) among those artifacts.

For simplicity of reasoning, in this paper we limit the model in respect to its original definition [7] by restricting the range of information about a project being collected by the architecture graph.

Let vertices of our architecture graph be limited to the following types:

$$\mathcal{V} = \{module; class; method\}.$$

These are the artifacts that are typically available in most modern object-oriented languages (including lambda expressions being anonymous methods).

Let edges of our architecture graph be limited to the following types:

$$\mathcal{E} = \{association : class \rightarrow class;$$

$$association : class \rightarrow method;$$

$$association : method \rightarrow class;$$

$$association : method \rightarrow method;$$

$$creation : method \rightarrow class;$$

$$inheritance : class \rightarrow class;$$

$$call : method \rightarrow method\}.$$

For example: an association denotes that a method is contained within (owned by) a class; also association relation exists when a method takes or returns as an argument an instance of a different class; or when a class defines a property within a different class; creation represents special methods responsible for construction statements in the source code, ie. using *new* keyword; inheritance denotes class hierarchical relations; call denotes steering being transferred from one method to another.

For the simplicity of the reasoning, in this paper we assume only static relations are represented in the architecture graph; though dynamic aspects, ie. dynamic calls, are possible to be automatically discovered and represented in the model [17].

We also omit types of static relations among artifacts, ie.

$$inclusion : module \rightarrow module \notin \mathcal{E}.$$

However please note that our approach to implementation of architectural assertions makes extending the scope of architectural knowledge represented in our architecture graph's easy; for more details on possible extensions see section VI. Also please note, that in our approach the relations can be implicitly extended by folding existing relations into new types of relations, ie.:

$$association : class \rightarrow method$$

and

$$creation : method \rightarrow class$$

in fact define

$$classcreation : class \rightarrow class,$$

that is a creator relation in which one class is responsible for creating objects of another class; see example of class factory in section V.

B. Architecture assertion

Let source project $\mathcal{P}$ be a software project created in any modern programming language (typically object-oriented) that is to be constrained using architectural assertions. Let $\mathcal{G} = \mathcal{G}(\mathcal{P})$ be the architecture graph derived for the project $\mathcal{P}$ (extracted from the project's source code).

In the remaining part of the paper please observe, that though we implement our approach using Visual Studio tools, this does not restrict the range of languages that our approach can be applied to.

Let architecture query denote a function that returns a subgraph of the given architecure graph

$$\mathcal{Q} : \mathcal{G} \rightarrow \mathcal{G}'$$

where $\mathcal{G}$ and $\mathcal{G}'$ are architecture graphs and $\mathcal{G}' \subseteq \mathcal{G}$. Then we can define architecture assertion as a comparison of the result set of an architecture query to the empty set.

Let architectural assertion $\mathcal{A}$ denote such an architecture query that the assertion is met (true) iff the executed query returns an empty graph; otherwise the assertions is broken (false):

$$\mathcal{A} : \mathcal{Q} \rightarrow \{true, false\}$$

defined as

$$\mathcal{A}(\mathcal{Q}) := \mathcal{Q}() == \emptyset \ ? \ true : false.$$

C. Architecture processing

We assume that tasks of software architects include defining constraints that bind software programmers during software development process. Put otherwise, architects create assertions that define desired (and also undesired) relations between the components of the system. A library of such assertions, when created, contributes to project's architectural knowledge. Consequently, the general approach to processing architectural assertions is as follows.

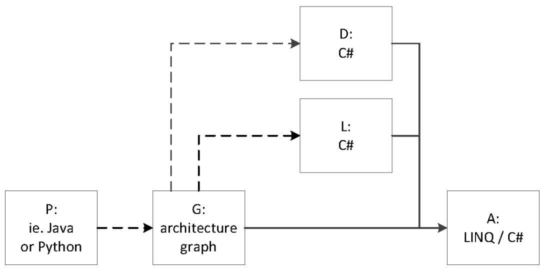

Figure 1. General approach to assertion processing

1) Process the source project $\mathcal{P}$ to extract its architecture graph $\mathcal{G}$;
2) Process the architecture graph $\mathcal{G}$ into the design project $\mathcal{D}$ denoted in a domain-specific language $\mathcal{L}$;
3) Create a collection of architectural assertions $\mathcal{A}$, where each assertion is defined in $\mathcal{L}$ in a notation referring to design project $\mathcal{D}$ (being an abstraction of the source project $\mathcal{P}$);
4) Evaluate assertions $\mathcal{A}$ to identify in $\mathcal{D}$ breaches of architectural rules;
5) Retract from $\mathcal{D}$ via $\mathcal{G}$ into $\mathcal{P}$ to identify fragments of $\mathcal{P}$ that violate architectural constraints imposed on the project.

See the following section IV for details on how those concepts have been assembled together using Microsoft technologies to constitute a general-purpose tool for defining and monitoring architectural assertions; see section V for examples of assertions.

IV. Implementation

Please recall the key design concept introduced in section III: (1) extracting from a given source project an abstract model that focuses only on architectural artifacts and their relations; (2) expressing the artifacts and relations in an intermediary layer denoted in a domain-specific language; (3) using an existing calculation environment capable of processing given domain-specific language as its calculation input.

For our proof-of-concept implementation of the design concept described in section III we harness the following Microsoft components:

- **IDE** Visual Studio Integrated Development Environment providing graphical user interface framework we extend for our purposes;
- **DSL** Visual Studio Domain Specific Language Tools allowing us to define an own domain-specific language to represent an abstraction of the source code;
- **T4** tools providing processing and persistence capabilities for our abstraction of the source code;
- **LINQ** syntactic sugar for concise notation of architectural assertions, ie. thanks to using anonymous methods (lambda queries);
- **Roslyn** for on-the-fly parsing, compiling and executing of the assertions.

A high-level overview of processing steps is depicted on figure IV. In subsequent parts of the section we provide more details on the goals of each step and how the components we selected are used to achieve those goals. We stress out that using these components, especially LINQ as the assertion notation, proves to be efficient in terms of: (1) high expressive power of notation used to define architectural assertions; and (2) small programming effort required to implement the automated verification of such assertions.

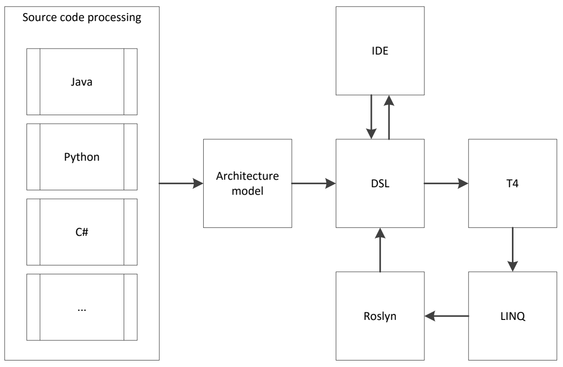

Figure 2. Implementation of processing steps using Microsoft components

A. Source code processing

A prerequisite for further adding and executing architectural assertions is processing source code (of the source project) in order to obtain its architectural model. In many cases such processing can be implemented effectively by analyzing the source code's abstract syntax trees (AST); ie. in case of languages like Java, Pythor or $C\#$ the compilers (interpreters) allow to analyze the source code's ASTs.

When examining a single node (of the tree) representing a method, we deduce the fact that the method calls another method (in some other class). Next we perform type solving; through a preliminary compilation of sources we check what is the type (class) of this method. Please note that though pre-compilation process is great for collecting information about software archiecture, there might be some cases when pre-compilation is not possible (ie. in case of interpreted programming languages). In this research we narrow focus only to languages for which source code compilers of adequate capabilities exist, and assume in our approach that abstract syntax trees constitute a first layer of abstraction between the source code files and the architecture graph.

B. DSL

To represent source code abstraction collected during source code processing, we utilize Visual Studio and its ability to provide an abstract mechanism for representing structures of any selected domain; namely we implement our architecture model using its Domain Specific Language Tools. To implement the graph structure we extend the DSL Tools' interfaces with own implementation classes, main ones being as follows.

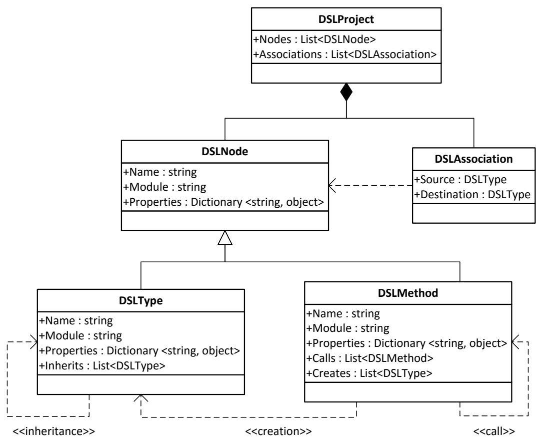

Figure 3. DSL representation of architecture graph

1) **DSLProject** is a container for all the artifacts of architectural model and the relations among them.
2) **DSLNode** is the superclass for all types of artifacts, sharing common properties of further entities. Those common properties are: (1) name (string), (2) module (string), and (3) dynamic dictionary of artifact's properties.
3) **DSLAssociation** allows to represent all type of ownership relations among the artifacts, like classes owning methods, methods owning (anonymous) methods, methods owning (anonymous) classes, classes owning (anonymous) classes.
4) **DSLType** represents the object-oriented concept of class entity, or any other similar architectural artifact, like structure, enumeration, etc.
5) **DSLMethod** represents architectural artifact for method statements in the software source code. Please note that to represent relation of binding methods to a class (like in object-oriented source code) we use DSLAssociation; however for several special relations (like creation, inheritance or call) we made a design decision of keeping direct pointers.

C. T4

In the next processing step we transform the DSL-represented data using T4 templates. The templates use instances of objects described in a DSL model as the input and generate any text containing the data from the DSL model. Templates allow iterations through data collections, conditional statements and text transformation using any software libraries provided for the .NET platform. In our case the templates are used to generate an intermediary code (in $C\#$) to be next appended with architectural assertions (in LINQ) and interpreted (with Roslyn). Regardless of the programming language of the source project, there are three types of $C\#$ files that get generated (for one source project).

1) **Architecture Model** The first file contains definition of artifacts that are used in definitions of architectural queries. Put otherwise, it contains the data model derived from the source code that gets referred to by architectural assertions. Technically, its is a reflection of the DSL representation of architecture graph. By design it is constructed to provide read-only data, ie. it should not be directly edited by software architects; if needed it should be re-generated from the source code. In some sense it resembles definitions of the objects in DSL layer with methods omitted (ie. status changing methods).
2) **Intermediary Code** While the first file contains an abstraction of the source code, the second file contains $C\#$ translation of the given source project. In particular the $C\#$ code contains the same structures as the corresponding structures in the source project, ie. if the source code includes some class, then in the intermediary $C\#$ code a class with exactly same name and properties will be created. The code inside the second file refers to the definitions from the first file. The second file is required for eventual execution of queries againts the source code that the queries aim to constraint. This file is also generated automatically from the source code and should not be edited manually by software architects.
3) **Query Definitions** The third file contains a collection of architectural queries. Definition of each query refers to definitions from the first file. Queries can be freely edited by architects, according to their personal experience. For the definitions of the queries to be interpreted correctly by VS, it must be combined with the previous two files.

Please note, that the first two files are delivered as a result of analysis of the source code and its transformation into $C\#$. They provide resources for software architects to define next their own architectural assertions. The resources are provided read-only, that is in case of architectural changes in the source code, the intermediary $C\#$ files must be re-generated. On the other hand, the third file contains a collection of queries as defined by architects themselves. Software architects are encouraged to build a collection of re-usable queries, so that introducing architectural rules or restrictions into new projects becomes quick and unexpensive.

D. Roslyn

Eventually the assertions are executed using Roslyn. It begins with combining all three types of files created for the given source project as described in the previous subsection into a $C\#$ program, and reparsing the respective $C\#$ program. What follows is an on-the-fly compilation of the program; compilation on-the-fly is a fully-fledged compilation, the same as for the creation of library files or other executables, with the exception that the compilation unit immediately goes into memory and is managed with a current thread. Within execution of the current thread it is possible to call any method of the loaded library and retrieve results, though architecture query results must be of types available in the unit of compilation.

In solution proposed in this paper, an architecture query result (graph) can be represented using simple types. That is the query may return either a tuple being a subset of vertices (list of strings) and a subset of edges (list of strings) of the architecture graph; or the subset of edges is empty, with a nonempty subset of vertices; or both subsets are empty (empty graph, with both lists being empty).

V. Main result

Please recall that an assertion is a comparison of an architecture query result to an empty set, where architecture query is a function returning a subset of the project's architecture graph. Hence the assertion is satisfied if and only if the query returns an empty graph. We demonstrate by the following examples how our approach can be applied to enforce architectural assertions:

(1) Unwanted Instability: only stable or unstable classes are allowed; (2) Factory Violation: object are constructed in factories only; (3) God Object: all-powerful objects are not allowed. For each example we summarize: (1) the architectural *problem* it aims to solve; (2) the subset of the graph *model* relevant for the assertion; and (3) the definition of the resulting *query*.

Please note the concise notation used to denote the query. Also please recall, that using this approach the assertions can be denoted in LINQ and $C\#$ for any type of source code, as long as the source code can be abstracted into a unified model of architecture graph; in particular it suffices that for the given source code language there exists an AST toolset, as it does ie. in respect to Java or Python.

A. Unwanted Instability

Problem: Instability metric (I) indicates module, package or class readiness for change [18]. It is calculated as the ratio of efferent coupling (Ce) to the efferent and afferent coupling (Ca), namely $I = Ce/(Ce+Ca)$. The range for this metric is $I \in [0..1]$. Accorind to metric author a module with instability close to value 0 is considered stable. A stable module has no references to other modules, can have number of internal references (among module's own artifacts). On the other hand a module with instability metric value close to 1 is considered instable. An instable module usually has a vast amount of outgoing references and a low amount of internal references. Modules with instability $I \in (0.3..0.7)$ are considered neither stable or unstable, as such being typically unwanted in a software project. The following query finds out which modules are unwanted in terms of instability metric (as defined above).

Model:

$$\mathcal{V} = \{class; method\}.$$

$$\mathcal{E} = \{association : class \to method;$$

$$call : method \to method\}.$$

Query:

```
[AssertAttribute]
public static IEnumerable<string>
      UnwantedInstability(){
  var en = from types in Project.Types.Where(x =>
    x.Callers.Count + x.Calls.Count > 0
  )
  group types by new {
    types.FullName,
    Ca = types.Callers.Count,
    Ce = types.Calls.Count,
    I = types.Calls.Count /
      (double)(types.Callers.Count +
      types.Calls.Count)
  }
  into rType
  where
    rType.Key.I >= 0.3 &&
    rType.Key.I <= 0.7
  select rType.Key.FullName;

  return en.ToList();
}
```

B. Factory Violation

Problem: The purpose of factory pattern is to hide the logic of creating individual objects. Creating objects outside the designated class factories violates the pattern. We assume that in our architecture model the factories are explicitly indicated, that is each such artifact has a name that contains *factory* suffix. We search for violations of factory pattern.

Model:

$$\mathcal{V} = \{class; method\}$$

$$\mathcal{E} = \{association : class \to method;$$

$$creation : method \to class;$$

$$call : method \to method\}$$

Query:

```
[AssertAttribute]
public static IEnumerable<string> FactoryViolation() {
  const string factoryLabel = "factory";

  var v = Project.Types
    .Where(x => x.Name.ToLower()
    .Contains(factoryLabel));

  if (!v.Any())
    return new[] { "" };

  var factoryArtifacts = Project.Types
    .Where(
      conn => conn is CreationConnection &&
      v.Any(y => y == x.Source)
    ).Select(x => x.Destination);

  return Project.Types
    .Where(
      conn => conn is CreationConnection &&
      factoryArtifacts.Any(y => y == x.Destination) &&
      !x.Source.Name.Contains(factoryLabel)
    );
}
```

C. God Object

Problem: One of previous examples shows how to use our approach to find out a violation of design patterns. Same idea can be used to find out if anti-patterns exist in the source project. One of well-known anti-patterns is *God Object*. This anti-pattern is a violation of *Single responsibility principle* rule defined as a part of SOLID rules [18]. Single responsibility principle constrains one class to provide logic for only just one functionality. Keeping classes inside source code simple and responsible for one thing each increases source code maintainability and software scalability. Placing one class which does too much things is a tempting phenomenon for inexperienced developers, hence we search for existence of *God Objects*.

Model:

$$\mathcal{V} = \{class; method\}$$

$$\mathcal{E} = \{association : class \rightarrow class\}$$

Query:

```
[AssertAttribute]
public static IEnumerable<string> GodObject()
{
  var types = Project.Types.Select(x => new
  {
    Type = x,
    Count = Project.Connections.
      Count(y => y.Destination == x || y.Source == x)
  });
  var v = types.OrderByDescending(x => x.Count);
  return v.Take(3).Select(x => x.Type.FullName);
}
```

VI. Conclusions

Our paper follows the research on architecture of software and software process. It promotes an approach that avoids separation between source code, software process and software architecture (design) artifacts. In this paper we demonstrate that an implementation of such approach is feasible. We demonstrate that LINQ, being a syntactic extension of $C\#$, can become a concise and expressive notation for defining architectural assertions. We also demonstrate that Visual Studio, being an integrated development environment, is a good platform to create a tool that allows software architects to enforce assertions upon software projects, uniformly treating source and architectural layers of the project.

The idea to extend functionality of existing integrated development environments is not novel, it has been alredy confirmed in practice and there exist plugins for specific domains of software engineering. The actual novelty of our approach lies in representing architectural artifacts using the same artifacts as the ones in the source code. More precisely, in parallel to the source project we generate an additional design project describing architecture of the source project. Additinally, if both projects follow the same syntactical rules (of the same programming language, ie. $C\#$ like in our example), then architectural artifacts can even melt with the actual source code hence be accessed and automatically processed just as any other parts of the source code, both from the perspective of integrated development environment, and from the perspective of a software programmer or software architect. Parallel execution of both projects - the source project and the design project - opens new opportunities. While the first project preserves its original business purpose, the second project becomes responsible for watching over internal architecture of the first project - validation of the architectural constraints placed upon artifacts and their interconnections. Another novel observation is the fact that to denote and automatically validate architectural assertions, software architects do not need to explore all the details of the source code. For this purpose only a certain abstraction of the source code is satisfactory - focusing on signatures of types and methods, their relations, but disregarding implementation specifics, ie. method conditional statements. Therefore we have observed that a source project in programming language A (ie. Java or Python, or any other language of similar characteristics) can be automatically transformed into a design project in another programming language B (ie. in $C\#$) such that we can express in language B architectural assertions in respect to architectural artifacts of the project in language A. It is actually practicable due to a common abstraction behind majority of current object-oriented programming languages. Thanks to this, for ie. a program in Java we can denote architectural assertions in LINQ, melt them into an automatically generated $C\#$ abstraction of the Java program and compile and execute the new $C\#$ program to get the assertions validated. In some sense this way $C\#$ becomes a *calculation description* and Visual Studio a *calculation engine*.

Our research can be extended in a few directions. Practical aspects include creating a publicly-available extension for Visual Studio containing all the described functionality, integrated and operationally verified. Theoretical aspects include extending the scope of graph model of software architecture (available as the domain for the architecture assertions) to include subsequent artifacts and subsequent relations; also researching its properties, expressive power, and the scope of programming languages compatible with this model; generally expanding the concept of collecting architectural knowledge [8], [1], [28]. Our approach requires also thorough verification and comprehensive, comparative testing; in particular creating a publicly available test-bed consisting of multiple source projects in multiple programming languages is a must. It can be anticipated that examples that include reflections, functional programming, dynamic method definition or other programming concepts may require refactoring of DSL implementation of the model, or redefinition of T4 transformations used for model processing, or introducing other concepts of model transformations [10]. In parallel we intend to start building a default library of architectural assertions denoted in LINQ, that would cover existing good architectural and design practices [4], [5]. Such library - created in abstraction from the source code language - would trigger another research stream, namely existing software projects could be systematically verified against the predefined architectural assertions. Another topic of research is appending information of architectural assertions into visual representation of software as a graph [2], [6], [15], [20]. Yet another direction is empowering software architects with tools (hence a notation) that would implement the long defined postulate that software process is a software as well [21], or even extended such approach to actually include all software project artifacts, not only source code artifacts [22], [27], [12].

References

[1] M. A. Babar and I. Gorton, editors. *Software Architecture, 4th European Conference, ECSA 2010, Copenhagen, Denmark, August 23-26, 2010. Proceedings*, volume 6285 of *Lecture Notes in Computer Science*. Springer, 2010.

[2] C. Bartoszuk, R. Dąbrowski, K. Stencel, and G. Timoszuk. On quick comprehension and assessment of software. In B. Rachev and A. Smrikarov, editors, *CompSysTech*, pages 161–168. ACM, 2013.

[3] K. Beck. Embracing change with extreme programming. *IEEE Computer*, 32(10):70–77, 1999.

[4] H. P. Breivold, I. Crnkovic, and M. Larsson. Software architecture evolution through evolvability analysis. *Journal of Systems and Software*, 85(11):2574–2592, 2012.

[5] N. Brown, R. L. Nord, I. Ozkaya, and M. Pais. Analysis and management of architectural dependencies in iterative release planning. In *WICSA*, pages 103–112, 2011.

[6] C. S. Collberg, S. G. Kobourov, J. Nagra, J. Pitts, and K. Wampler. A system for graph-based visualization of the evolution of software. In S. Diehl, J. T. Stasko, and S. N. Spencer, editors, *SOFTVIS*, pages 77–86, 212–213. ACM, 2003.

[7] R. Dąbrowski. On architecture warehouses and software intelligence. In T.-H. Kim, Y.-H. Lee, and W.-C. Fang, editors, *FGIT*, volume 7709 of *Lecture Notes in Computer Science*, pages 251–262. Springer, 2012.

[8] R. Dąbrowski, K. Stencel, and G. Timoszuk. Software is a directed multigraph. In I. Crnkovic, V. Gruhn, and M. Book, editors, *ECSA*, volume 6903 of *Lecture Notes in Computer Science*, pages 360–369. Springer, 2011.

[9] R. Dąbrowski, G. Timoszuk, and K. Stencel. One graph to rule them all software measurement and management. *Fundam. Inform.*, 128(1-2):47–63, 2013.

[10] J. Derrick and H. Wehrheim. Model transformations across views. *Sci. Comput. Program.*, 75(3):192–210, 2010.

[11] E. W. Dijkstra. Letters to the editor: go to statement considered harmful. *Commun. ACM*, 11(3):147–148, 1968.

[12] A. Egyed and P. Grünbacher. Automating requirements traceability: Beyond the record & replay paradigm. In *ASE*, pages 163–171. IEEE Computer Society, 2002.

[13] L. Fabresse, N. Bouraqadi, C. Dony, and M. Huchard. A language to bridge the gap between component-based design and implementation. *Computer Languages, Systems & Structures*, 38(1):29–43, 2012.

[14] R. Kaufmann and D. Janzen. Implications of test-driven development: a pilot study. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '03, pages 298–299, New York, NY, USA, 2003. ACM.

[15] R. Koschke. Software visualization for reverse engineering. In S. Diehl, editor, *Software Visualization*, volume 2269 of *Lecture Notes in Computer Science*, pages 138–150. Springer, 2001.

[16] I. Lytra, H. Tran, and U. Zdun. Constraint-based consistency checking between design decisions and component models for supporting software architecture evolution. *2011 15th European Conference on Software Maintenance and Reengineering*, 0:287–296, 2012.

[17] V. Markovets, R. Dąbrowski, G. Timoszuk, and K. Stencel. Know thy source code. is it mostly dead or alive? In C. K. Georgiadis, P. Kefalas, and D. Stamatis, editors, *Local Proceedings of the Sixth Balkan Conference in Informatics, Thessaloniki, Greece, September 19-21, 2013*, volume 1036 of *CEUR Workshop Proceedings*, page 128. CEUR-WS.org, 2013.

[18] R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[19] J. McCarthy, M. I. of Technology. Computation Center, and M. I. of Technology. Research Laboratory of Electronics. *Lisp one five programmer's manual*. Massachusetts Institute of Technology, 1965.

[20] R. L. Nord, I. Ozkaya, and R. S. Sangwan. Making architecture visible to improve flow management in lean software development. *IEEE Software*, 29(5):33–39, 2012.

[21] L. J. Osterweil. Software processes are software too. In W. E. Riddle, R. M. Balzer, and K. Kishida, editors, *ICSE*, pages 2–13. ACM Press, 1987.

[22] S. P. Reiss. Dynamic detection and visualization of software phases. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–6, 2005.

[23] W. Royce. Managing the development of large software systems: Concepts and techniques. In *WESCOM*, 1970.

[24] P. Spacek, C. Dony, and C. Tibermacine. A component-based meta-level architecture and prototypical implementation of a reflective component-based programming and modeling language. In *Proceedings of the 17th International ACM Sigsoft Symposium on Component-based Software Engineering*, CBSE '14, pages 13–22, New York, NY, USA, 2014. ACM.

[25] P. Spacek, C. Dony, C. Tibermacine, and L. Fabresse. Bridging the Gap between Component-based Design and Implementation with a Reflective Programming Language. Technical report, Laboratoire d'Informatique de Robotique et de Microélectronique de Montpellier - LIRMM , Unité de Recherche Informatique et Automatique - URIA, July 2013.

[26] P. Spacek, C. Dony, C. Tibermacine, and L. Fabresse. Wringing out objects for programming and modeling component-based systems. In *Proceedings of the Second International Workshop on Combined Object-Oriented Modelling and Programming Languages*, ECOOP'13, pages 2:1–2:6, New York, NY, USA, 2013. ACM.

[27] G. Spanoudakis and A. Zisman. Software traceability: a roadmap. *Handbook of Software Engineering and Knowledge Engineering*, 3:395–428, 2005.

[28] M. T. T. That, S. Sadou, and F. Oquendo. Using architectural patterns to define architectural decisions. In T. Männistö, A. M. Babar, C. E. Cuesta, and J. Savolainen, editors, *WICSA/ECSA*, pages 196–200. IEEE, 2012.

[29] C. Tibermacine, R. Fleurquin, and S. Sadou. A family of languages for architecture constraint specification. *Journal of Systems and Software*, 83(5):815–831, 2010.