Computer Science • 13 (2) 2012 http://dx.doi.org/10.7494/csci.2012.13.2.81

Piotr Poznański
Mariusz Wawrowski

IMPROVING SOFTWARE SYSTEMS BY FLOW CONTROL ANALYSIS

Abstract *Using agile methods during the implementation of the system that meets mission critical requirements can be a real challenge. The change in the system built of dozens or even hundreds of specialized devices with embedded software requires the cooperation of a large group of engineers. This article presents a solution that supports parallel work of groups of system analysts and software developers. Deployment of formal rules to the requirements written in natural language enables using formal analysis of artifacts being a bridge between software and system requirements. Formalism and textual form of requirements allowed the automatic generation of message flow graph for the (sub) system, called the "big-picture-model". Flow diagram analysis helped to avoid a large number of defects whose repair cost in extreme cases could undermine the legitimacy of agile methods in projects of this scale. Retrospectively, a reduction of technical debt was observed. Continuous analysis of the "big picture model" improves the control of the quality parameters of the software architecture. The article also tries to explain why the commercial platform based on UML modeling language may not be sufficient in projects of this complexity.*

Keywords software engineering, architecture definition, requirements management, testing, agile, systems modeling

1. Introduction

Nowadays system engineers are facing new challenges [13]. Finding a strategy that enables the rapid development of high-quality products being a hybrid of software components developed using numerous specific technologies is not a trivial task. Suitable technology and process should support efficient system evolution the management of technical debt should be element of this strategy. For example, the traditional embedded system should be developed the way to minimize potential cost of migration from/to the system on the chip (SoC) solution to/from the system in the box (SIB). This expectation usually leads to solution where implementation gets independent of hardware and technology as much as possible. Achieving greater scalability and viability of solutions in distributed systems without compromising on other quality parameters is next step.

Software developers often use UML language while system engineers work on quality attribute scenarios using six-part-templates (SPT). Fortunately, there is a natural convergence between UML diagrams and the model contained in the SPT notation. This has become an incentive for authors to create a tool allowing for automatic generation of high level message flow. The diagram is limited to collaboration between active objects. It can be generated based on SPT specification. It is interesting to see how SPT in conjunction with formal language supported by the tool impacts on software development in terms of:

- Productivity
  - Speed up startup work on implementation.
  - Optimize refactoring effort.
  - Reduce the cost of maintenance work on requirements and initial model.
- Quality
  - Reduce the total number of defects.
  - Minimize defect introduced in the requirements.
  - Improve traceability – control scope change (work out method allowing to get formal link between system and software implementation).
- Technical debt
  - Control modifiability, scalability, performance indicators.
  - Open architecture for Commodity-Off-The-Shelf (COTS) hardware components to enable rapid development.
  - Reduce dependency from IDE to minimize migration between (environments e.g. from TAU to IBM Rational Rhapsody).

2. Literature review

Since the "Structured Design" article [8] was published, the subject of software architecture modeling has been widely discussed among engineers in the IT industry. It

is worth paying attention to the Data Flow Diagram (DFD), not only for historical reason, but because it represents a procedural system analysis method being used often in legacy systems that need support until now [9]. Nowadays, the best-known modeling language is UML, representing object oriented design (OOD) techniques. An interesting comparison OOD and DFD is presented by Morris [10]. UML is also widely used during embedded software development. Samek presents the effective use of UML state-charts for modeling small embedded systems [11]. IBM Rational Rhapsody is an example of an environment that may address the expectations of engineers who work on more complex embedded systems. The methodology based on this tool is presented by Bruce Powel Douglass [3]. On the other hand, some inconveniences of the UML are listed by Scott W. Ambler [2]. For example, he stresses the lack of effective support for systems based on SOA. Ambler also shows how DFDs are used in projects driven by agile practices. Using DFD in today's modeling is also shown by Fairbanks [1]. Attention is drawn to the simplicity of DFD generation based on the static analysis of the source code. The lecture of the article of M. Dalgarno and M. Fowler [14] is also interesting. The authors discuss the benefits and disadvantages of domain specific languages (DSL). An important opinion contained in this work is the statement: "a basic DSL can be produced using UML profiles and this will often be a viable and relatively quick approach". It is necessary to mention the architecture tradeoff analysis method (ATAM) [12][5]. SPT is related to this method and assumes writing requirements as quality attribute scenarios (QA)[1].

3. The six-part template and formal language

Since the choice of software development process is associated with enabling optimal communication between all project stakeholders, the presented approach focuses on the information flow between requirement or architecture developers and code developers. Architects may use SPT to analyze and document use cases in text form (user scenario) stored in requirement repository e.g. DOORS. SPT at lower levels of design may be also used. Active Objects [13] behavior and communication is described in SPT. Therefore, linking the user scenarios and design artifacts in a natural way (traceability) is facilitated, and formal verification of the completeness and consistency between the requirements (architecture and code) is possible using the presented approach.

The approach consists of a specification language based on SPT and a self-developed toolsuite. It allows for:

- system message flow visualization on a directed graph,
- querying endpoints (objects) for protocols and protocols for endpoints,
- identification of orphaned messages and finding lacks in the design specification,
- identification of uncovered or duplicated areas and message flow paths,
- having plain-text model representation that could be testable.

As a result of the analysis of the data obtained by the toolsuite, it is possible to:

- identify critical paths through the system and plan for tests,
- identify possible points of failure and risk areas,
- identify possible flaws and suboptimal design such as overloaded components.

3.1. Language

The specification language is a DSL plain text based on SPT described in the literature [5]. Project legacy requirements had already had textual representation, deployed tools (DOORS) were text oriented and in such circumstances it was also faster to create processing tools for textual representation. An example of a specification requirement shows the basic concepts of the language (Listing 1).

```
Artifact: TService::TServiceManager,TService::TDownlinkService,
RFModemDst_I, TService::TRepeatController
Source: RFModemDst_I
Stimulus: acceptFrame() with ICW START (TRANS MODULA APCO ICW
or TRANS MODULA ACTLICW)
Environment:
TServiceManager is in IDLE state (no on-going Services)
TDownlinkService is in IDLE state (no on-going Services)
Response:
TServiceManager converts the ICW START to OutboundServiceRequest
TServiceManager sets the infrastructureActive flag
TServiceManager grants the Downlink Service
TServiceManager sends initServiceRequest() signal to TDownlinkService
TServiceManager sends TDownlink() signal to TRepeatController
```

Listing 1: A requirement specification example

The `Artifact` section specifies all the actors (objects) taking part in a specification requirement. The `Source` section specifies the originator of a message that triggers specified behaviour. `Stimulus` is a signal sent by the `Source` (in this example, RFModemDst_I). The purpose of the `Environment` section is to specify the internal state of the requirement's actor (in this case TServiceManager). Finaly, the `Response` section describes all the actions taken by the actor triggered by the `Stimulus` sent by the `Source`. In the `Reponse` section the following statements are recognised by the parser:

- X sends signal to Y,
- X sets SomeVariable/SomeFlag to value,
- X enters S state.

The language parser in contrast to a regular programming language parsers by the design choice allows statements that are not recognised. It is allowed to enter comments or very specific annotations, which are just skipped. This fact forced by legacy considerations introduces some relaxation of the expression rules, however, it opens up the door for entering erroneous statements. The parser generates warnings

on such occasions and it is possible to obtain simple reports on how many lines were found and parsed in requirements as a very simple countermeasure (listing 2). It is planned to remove this feature and leave only a strict parsing mode with stop on error behaviour, once legacy syntax is removed out of all requirements.

```
Number of parsed lines in responses
COMP-USE-CASE-6009 (Req): 5 of 5
COMP-USE-CASE-5717 (Req): 3 of 4
COMP-USE-CASE-5978 (Req): 5 of 5
COMP-USE-CASE-5714 (Req): 3 of 3
COMP-USE-CASE-5713 (Req): 3 of 6
COMP-USE-CASE-6154 (Req): 5 of 9
COMP-USE-CASE-6151 (Req): 3 of 8
COMP-USE-CASE-5738 (Req): 2 of 6
COMP-USE-CASE-5736 (Req): 3 of 11
COMP-USE-CASE-6152 (Req): 7 of 15
COMP-USE-CASE-6153 (Req): 5 of 13
...
```

Listing 2: An example of parser output

3.2. Graph Representation

Graph representation is generated out of the specification. Actors (objects) are the nodes, and vertices are marked with messages and numbers of requirements which describe specific interactions. An example of a subgraph generated for about a dozen and hundreds (out of several thousands) requirements are presented in Fig. 1 and Fig. 2, respectively.

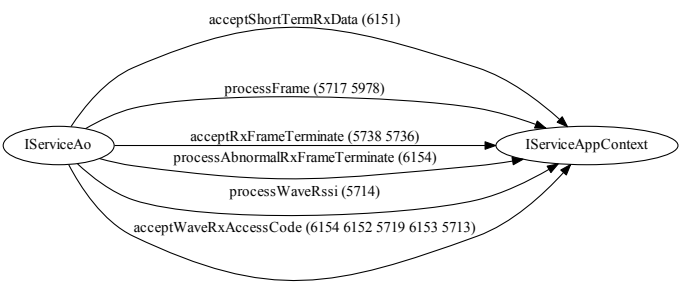

Figure 1. An example of a graph of 12 requirements

4. Results

A few main metrics were used to measure the productivity of a software development team and the quality of its result – software product. Some of them are interrelated

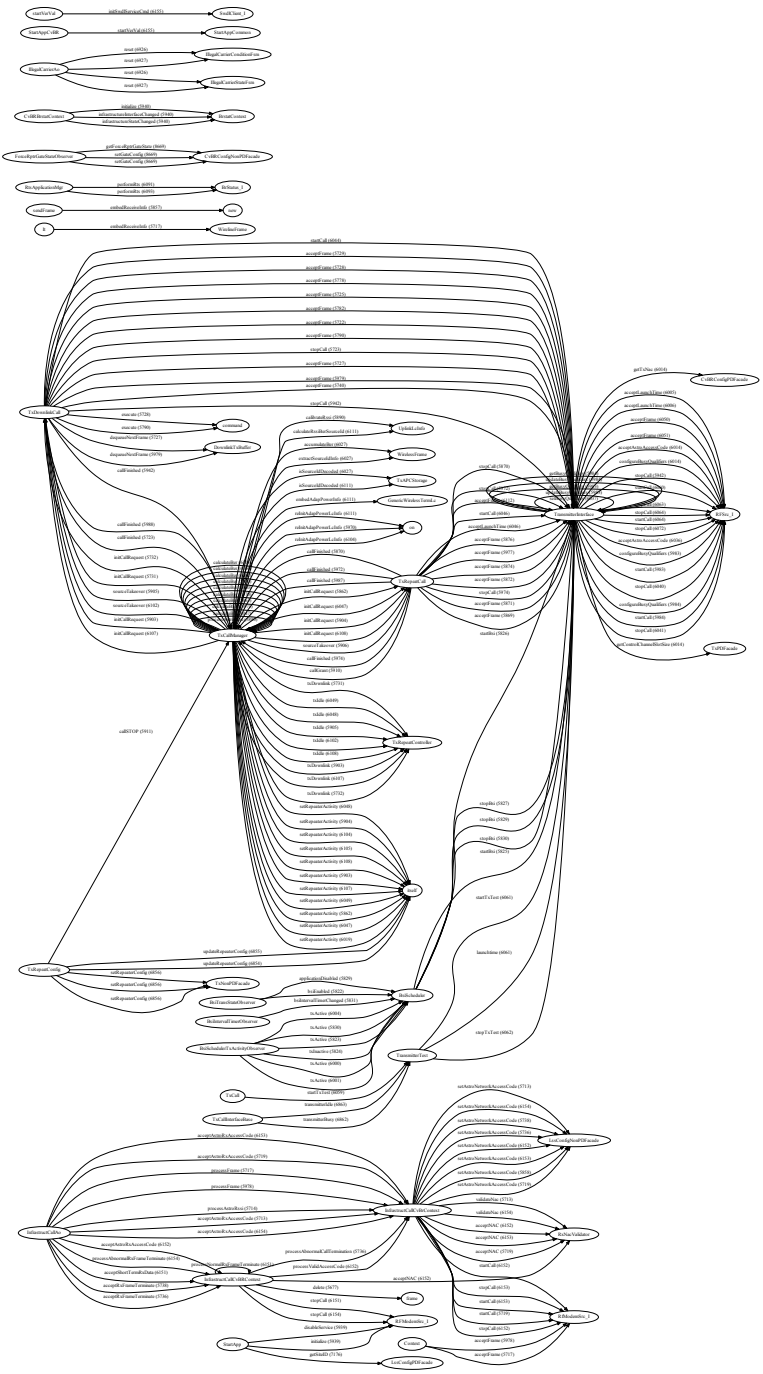

Figure 2. An example of a graph of hundreds requirements

not always in linear or sublinear manner, and sensitive to general project structure. This again can be analysed further by introducing more specific and detailed metrics. However, some qualitative indication with a reasonable trust level are conducted based on:

- effort used e.g. expressed in man-month,
- software and process quality expressed in defects found in software during integration and acceptance testing.

Two more metrics should be mentioned as very important:

- team size
- overall project time

because they impact the effort and quality strongly. The better the development project scales, the longer man-month, project time and development team size are interdepedent lineary. In the general defect number depends on project size, process, tools, and team experience neglecting other dependencies such as overall time. The better process and tools are, less the defect number is affected by the team experience.

Technical debt, a very important metric, very often estimated in man-month is measured indirectly as a difference of planned effort spent between two consecutive releases.

For the sake of clarity and to avoid blurring the discussion, the analysis will be conveyed with effort, defects number, and technical debt. Quantitive indication to draw conclusions will not suffer.

Results and conclusions are given based on metrics gathered during three consecutive software projects (Project 1, 2, 3) of adding new features to an existing code base. Essentially, the second project was a continuation of the first one, and the third a continuation of the second one. Sizes of the projects varied as the number and nature of the features did.

Project 1 was lead using an old development process without having in place the solution presented in this paper. The solution was introduced in Project 2 with roughly a new development and architecture team (team A). Project 3 was lead with the same process as Project 2, however, in addition to team A (with rotation about 50% of developers) an entirely new team (team B) was introduced. Team B was couched by team A to some extend, however, team B did not use strict rules for creating and managing SPT.

Projects metrics data are presented in Table 1. They had been estimated with:

- COCQALMO for estimating the number of defects injected by a particular development team.
- Function Points Analysis (FPA) and Wideband Delphi for estimating effort needed for software development.

Project 1 and 2 were estimated exactly with the same parameters in FPA and COCQALMO, thus did not take into account probable gains of the new development process in both resource demand and quality plains for Project 2. The estimation

Table 1
Project metrics data

project /team	estimated effort	effort	refactoring / estimated technical debt decrease	estimated defect numbers	actual defect number
1/x	220	220	n/a	70	70
2/a	220	200	20	65	40
3/a	400	380	20	120	80
3/b	500	550	n/a	200	210

methods were customised for the new process and advances in team A experience for Project 3. There are a few important observations based on presented metrics:

Technical debt decrease
Although, for Project 2 effort was consumed as estimated, the same as for Project 1, in contrast to Project 1 at the end legacy and new architecture and implementation were refactored to achieve more general architecture leading to enable a better fit of implementation of new features in Project 3. It took about 10% of the effort, which makes for 20 man-months. Therefore, it may be estimated that the technical debt was decreased by 20 man-months.

Software quality increase
The defects number (per man-month) was lower in Project 2, which proved the new process to result in better quality software

Project structuring
Comparison of defects number between teams in Project 3 shows that when a loose approach to SPT was excercised the quality of code decreased and effort increased beyond estimations.

5. Consequences and considerations

The presented solution has an impact on several aspects of the software process development. The most important benefits and disadvantages are highlighted and discussed below.

Architecture analysis
The flow diagram shows the so-called "big picture" and allows for rapid overview and semi-automate analysis of the entire system at this level. For example, this may be a qualitative analysis of architecture (scalability, performance, modifiability), in effect leading to the construction of a software development strategy, taking into account a compromise between factors such as "time to market" and quality.

Reverse and forward engineering
Possibility to generate specification in SPT based on the existing source code

(reverse engineering) is also an interesting feature. The auto-generated artifact is a model, a bridge between requirements (written in text form) and source code. The audit of this model can be done manually or automatically. It may lead to discovering incompatibilities between source code and requirements. Further analysis leads to the conclusion that model may be used during various software development phases.

- The forward engineering approach assumes optimizing the design phase effort. The new function can be visualized as a change in a relation between components and in message flow. Naturally, the "big picture" does not include implementation details that affect the overall quality of the product. It comes from the assumption that a detailed analysis of individual components takes place during code and unit test implementation e.g. test driven development methodology (TDD). Responsibility for low level details is delegated to developers working on source code. The high level view presented in the flow diagram is also useful to define an optimal test strategy [6, 7]. Enabling efficient ramp-up for less experienced engineers is another positive side effect of this approach.
- The reverse engineering approach is based on the review and audit of the design after software code creation. The result of this review leads to decisions helping to manage the source code refactoring in future projects. For example, after agile iteration engineers may execute a static source code analysis to capture the flow diagram. It facilitates design audit in the post-development phase and defines the strategy how technical debt should be paid.

Defect number reduction

As shown quality improvements were achieved mostly by catching errors in the early stages of software development.

Technical debt

Technical debt management designates the direction of long term software evolution. The optimal strategy of payoff the technical debt should be driven by business value referring to predicted revenue, effort distribution across system, product quality, cost and time to market factor. The "big picture" diagram helps us to assess the architecture quality indicators and estimate technical debt value.

Commercial of the shelf (COTS)

Usually the best option is to invest in commercial tools supporting modeling in UML. The message sequence chart or high level collaboration diagrams are sufficient to create "big picture". Unfortunately, there are also potential disadvantages:

- Model environment is not integrated with upstream artifacts that impacts the maintenance of traceability between requirements, model, and source code.

- Integration between modules developed in other commercial environment is difficult.
- Model representation written in non-standard language hinders the extension of commercial toolsets.
- The cost of migration between commercial environments impacts on technical debt value.
- The cost of license and maintenance impacts on return of investment (ROI)

Extra effort

The specific language using SPT, requirements repository, tools and integration it to existing development environment require additional investment. However, improving the development environment and deploying methods that fits to project domain can be supported by incremental development processes. The gradual change and frequent feedback allows for the assessment of the benefits and minimizes the risk of bad investments during the project.

Despite these considerations, industry results show that the commercial environment and toolkit give a lot of benefits in terms of quality and rapid development. However, there is still place for domain specific solution. Language and tool presented in previous section extends capability of the commercial toolkit, where mitigation of the above disadvantages may be as result. For example IBM Rational Rose model used by software developers may be easily linked to requirements written in SPT notation. This formal link reduces unintended differences between requirements and source code. This form of requirements enables migration to any new more advanced environment where the development of new features may be done more efficiently.

6. Future plans

The presented solution will be extended in two areas: specification language and the processing toolsuite. As for the language, it is foreseen to include UML and research in the area of workflow description specific languages. The toolsuite may be extended to allow for:

- Technical debt analysis.
- Using XML to facilitate integration with widely accepted tools such as Rational Rose Architect.
- Code reverse engineering – it is desired to recreate at least a partial specification by reading e.g. C++ language. It will be highly domain specific and assume some tight constraints on the expressiveness of a programming language being reverse engineered.
- Code generation same as some tools already supplied, e.g. IBM Rational Rose. The ideal vision is to have a one tool or workflow with possibility of system specification, implementation and creating tests, so to speak, to close loop, and encompass the formal development from the specification formulation down to very implementation and testing.

- Integration with a database system to enable efficient querying for specific properties such as definitions of protocols, endpoints, etc.

7. Summary

Formally, presented language and toolsuite are oriented on modeling. The main goal was to create specific collaboration diagram. Initially, presented language was created as an efficient method for auditing requirements and design, and optimization of project effort. Defect number was reduced. The tool is a promising method to manage technical debt efficiently therefore a method for identification of enablers leading to more scalable and open architecture in the future – similar to SOA.

References

[1] George H. Fairbanks; Just Enough Software Architecture: A Risk-Driven Approach Marshall & Brainerd, 2010

[2] Scott W. Ambler; The Object Primer: Agile Model-Driven Development with UML 2.0;Cambridge University Press; 2004

[3] Bruce Powel Douglass; Real-Time Agility: The Harmony/ESW Method for for Real-Time and Embedded Systems Development (Kindle Edition); Addison-Wesley Professional, 2009.

[4] Bruce Powel Douglass; Real Time UML Workshop for Embedded Systems (Embedded Technology); Newnes, 2006

[5] Robert L. Nord,Mario R. Barbacci,Paul Clements,Rick Kazman,Mark Klein, Liam O'Brien, James E. Tomayko; Integrating the Architecture Tradeoff Analysis Method (ATAM) with the Cost Benefit Analysis Method (CBAM); Carnegie Mellon University, 2004

[6] Cheng A.M.K.: Real-Time Systems: Scheduling, Analysis, and Verification. John Wiley & Sons, 2002

[7] Porter A.; Accelerated Testing and Validation. Elsevier, 2004

[8] Wayne P. Stevens, Glenford J. Myers, Larry L. Constantine: Structured Design (IBM Systems Journal, 13 (2), 115-139, 1974 IBM Systems Journal, 13 (2), 115-139, 1974)

[9] Bruza, P. D., Van der Weide, Th. P.:The Semantics of Data Flow Diagrams, University of Nijmegen, 1993

[10] Morris, M. G., Speier, C. and Hoffer, J. A. (1999), An Examination of Procedural and Object-oriented Systems Analysis Methods: Does Prior Experience Help or Hinder Performance?. Decision Sciences, 30: 107-136. doi: 10.1111/j.1540-5915.1999.tb01603.x

[11] Samek, M. (2006). UML Statecharts at $10.99. Dr.Dobbs Journal, May 24, 2006

[12] Bass, Len, Clements, Paul and Kazman, Rick: Software Architecture in Practice, 2nd edition.Addison-Wesley, 2003

[13] P. Poznański, M. Wawrowski, J. Smagłowski. Trendy rozwoju architektury aplikacji osadzonych na systemach czasu rzeczywistego. KKIO, 2011

[14] Mark Dalgarno, Matthew Fowler: "UML vs. Domain-Specific Languages", Methods & Tools – Summer 2008

Affiliations

Piotr Poznański
Institute of Teleinformatics, CUT Kraków, poznan@mars.iti.pk.edu.pl

Mariusz Wawrowski
Motorola Solutions, Kraków, mariusz.wawrowski@motorolasolutions.com

Received: 19.12.2011
Revised: 04.04.2012
Accepted: 23.04.2012