e-Informatica

software engineering journal

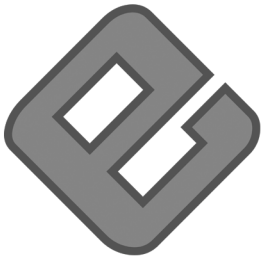
2011 volume 5 issue 1

e-Informatica

e-Informatica

software engineering journal

2011 volume 5 issue 1



e-Informatica

Wrocław University of Technology

Editors

Zbigniew Huzar (*Zbigniew.Huzar@pwr.wroc.pl*)
Lech Madeyski (*Lech.Madeyski@pwr.wroc.pl, http://madeyski.e-informatyka.pl/*)

Institute of Informatics
Wrocław University of Technology, 50-370 Wrocław, Poland

e-Informatica Software Engineering Journal
http://www.e-informatyka.pl/wiki/e-Informatica/

Printed in the camera ready form

OFICYNA WYDAWNICZA POLITECHNIKI WROCŁAWSKIEJ
Wybrzeże Wyspiańskiego 27, 50-370 Wrocław

ISSN 1897-7979

Drukarnia Oficyny Wydawniczej Politechniki Wrocławskiej. Order No. xxx/2011.

Editorial Board

Contents

e-Informatica Software Engineering Journal, Volume 5, Issue 1, 2011, pages: 7–23, DOI 10.2478/v10233-011-0027-z

VERSITA

Factors Determining Long-term Success of a Measurement Program: An Industrial Case Study

Miroslaw Staron*, Wilhelm Meding**

*Department of Computer Science and Engineering, Chalmers | University of Gothenburg

**Ericsson SW Research, Ericsson AB

miroslaw.staron@ituniv.se, wilhelm.meding@ericsson.com

Abstract

Introducing measurement programs into organizations is a lengthy process affected by organizational and technical constraints. There exist several aspects that determine whether a measurement program has the chances of succeeding, like management commitment or existence of proper tool support. The establishing of a program, however, is only a part of the success. As organizations are dynamic entities, the measurement programs should constantly be maintained and adapted in order to cope with changing needs of the organizations. In this paper we study one of the measurement programs at Ericsson AB in Sweden and as a result we identify factors determining successful adoption and use of the measurement program. The results of our research in this paper are intended to support quality managers and project managers in establishing and maintaining successful metrics programs.

1. Introduction

Several authors have already discussed factors that determine successful measurement program adoption at a company, e.g. [1, 2, 3]. The results usually are focused on addressing the question "How to establish a measurement program at a company?" which is a prerequisite for the success of the measurement program. Little, however, has been said about the factors that determine if a successfully implemented measurement program 'lives' longer than just the first project for which it was established (or until the first re-organization). In this paper we present a study which we conducted at Ericsson AB, which identifies and prioritizes factors important in long-term adoption of a measurement program. Ericsson, being one of the largest telecommunication equipment manufacturers in the world, has a distributed organization and a whole spectrum of projects (from small to very large).

The main processes are stable in the organization despite re-organizations, process customizations, and usage of various tools is normal situations in the company – conditions which are prevalent in software engineering and uncommon in manufacturing industries. These factors make the needs for measurement programs change constantly and require the program to evolve. In this paper we present results from a survey conducted at the company assessing the success of the measurement program and the measurement systems used in it. The results of this survey are combined with results of interviews with designers of measurement systems in industry to identify the success factors.

In contrast to existing body of knowledge in software engineering, instead of focusing on the establishment of the measurement program,

which most of the articles discuss, we focus on addressing the question of 'Keeping the measurement program alive' as identified by Clark [4]. Therefore in our research we address the following research question:

Which are the main factors determining a long-term success of a measurement program?

By using the term 'long-term' we mean that the measurement program is used in the organization in more than in a single project, that it gets extended over time, and that it becomes 'the new way of working' in the organization (gets integrated in the organization in the everyday work) – the studied measurement program is in existence for 5 years at the time of this study.

The main contribution of our work is identification of four key roles in establishing long-term measurement programs: section manager, stakeholder, quality manager and designer of measurement systems. A number of success factors which are associated with each of the role separately and with several roles together (which is shown through cluster analysis using K-Means tests for clusters). These factors help the roles in being effective and efficient when establishing measurement programs. By efficient we mean that it is possible to run measurement program for an organization of several hundred employees with small resources (ca. 2 full-time employees) dedicated for measure collection, analysis and presentation. We present our factors with short experience reports of how this worked on the case of the studied organization; these guidelines are intended to help other practitioners in realizing measurement programs in other companies.

The paper is structured as follows. Section 2 presents the most related research in the field. Section 3 presents the design of the study and with its subjects, objects, and instruments. Section 4 presents the elicited success factors preceded by the direct results of the case study. Section 5 evaluates validity of our study while section 6 presents the conclusions.

2. Related work

We investigated the following publications in order to elicit factors important when introducing metric programs into organizations in general, and not to be constrained only to Ericsson's context:

- Umarji and Emurian [1]: the study describes the use of technology adoption theory when implementing metric programs with focus on social issues. One of the important results from that study was the importance of the factor "ease of use". When developing our framework we invested in making the framework easy to use and making the presentation of the indicators easy to interpret.
- Gopal et al. [5] and Gopal et al. [6]: these studies present results and conclusions from a survey about metric program implementation conducted with managers at various levels (over 200 data points). The results indicated the importance of such factors as management commitment and the relative low importance of such factors as data collection. In order to check how important the framework is for the managers who we work with, we included the line manager and the project manager in our interviews when evaluating the framework.
- Atkins et al. [2]: among other aspects, this paper discusses how metrics can be reused by projects working on similar things in parallel. We used their experiences when reasoning about the reuse of metrics between different instances of the framework.
- Lawler and Kitchenham [7]: based on the experiences of several case studies, this paper discusses the issues of using metrics at different levels and combining metrics together (e.g. combining metrics from particular designers to provide the status of the whole project). This work affected the design of the framework in such a way that the metrics in the framework can be reused and combined in a way consistent with the study by Lawler and Kitchenham.

- Kilpi [3]: this paper describes how a metric program was implemented at Nokia. We used their experiences when evaluating the framework.
- Niessink and van Vliet [8, 9]: these studies describe external factors important for software metric implementation, including the importance of the goal of software measurement processes. Our experiences support this conclusion, and the need for the monitoring status and progress resulted in finally choosing the ISO/IEC 15939 standard as a basis for our work with metrics.
- de Panfilis et al. [10]: this study describes experiences from introducing a GQM-based metric program. Our experiences showed slightly contradicting picture that one of the most important aspects is not the sole moment of adoption of a program (as advocated by GQM) and possibilities of using subjective metrics, but the use of objective metrics to monitor entities over longer periods of time. A more detailed guidelines supporting the introduction of metric programs can be found in Goodman [11] or [12].
- framework presented by Diaz-Ley et al. [13] can be seen as suitable for smaller enterprises whereas the set of success factors and the framework from Ericsson [14] is targeted mainly for larger enterprises with a number of management levels. The main difference between the large and small-medium enterprises in the context of our work is the fact that the larger enterprises are organized using significantly more levels of management and multiple dimensions of management – e.g. project managers are usually not line managers.

One of the observed issues in program adoption is the reuse of measures. As Jorgensen [15] shows, this is not an easy task due to the potential different definitions of measures. Jorgensen shows contrasting definitions of measures if quality is defined as "a set of quality factors", "user satisfaction", and "software quality related to errors". Our research recognizes the needs for viewing the same aspects (e.g. quality) from different perspectives – depending on the stakeholder. These needs are also recognized by the Ericsson's measurement team which we collaborated with.

The concept of a measurement system is not new in engineering or in software engineering – measurement instruments and systems are one of the cornerstones of engineering. In software engineering, we are used to working with metric tools rather than measurement systems. The difference is that metric tools and measurement instruments seem to be very similar, but metric tools and measurement systems are not. Measurement instruments (in other engineering disciplines) are suited for single purposes and usually collect one metric (e.g. voltage) whereas metric tools collect usually a number of metrics at the same time (e.g. length of the program, its complexity). Our framework is placed on top of metric tools with the focus on presenting calculating and presenting indicators rather than collecting metrics and is intended to be composed of multiple measurement instruments (metric tools). Other examples of measurement systems built in the same principles are:

- A measurement system presented by Wisell [16]: where the concept of using multiple measurement instruments to define a measurement system is also used widely at the studied organization.
- Computerized measurement systems in other disciplines facilitating the concept of measuring instruments, as presented in the following papers: [17, 18, 19, 20, 21, 22, 23, 24]. All these measurement systems are (i) using the concept of measurement instruments, (ii) used in established engineering fields or physics, (iii) focused on monitoring current value of an attribute (status in our case) not on collecting metrics. Although differing in domains of applications these measurement systems show that concepts which the measurement team adopted from the international standards (like [25]) are successfully used in other engineering disciplines.
- Lawler and Kitchenham [7] present a generic way of modeling measures and building more advanced measures from less complex ones. Their work is linked to the TychoMetric [26] tool. The tool is a very powerful measure-

ment system framework, which has many advanced features not present in the Ericsson's framework (e.g. advanced ways of combining metrics). TychoMetric provides a possibility of setting up advanced and distributed (over several computers) filters and queries for multiple data sources as it is intended to cover all (or at least very many) kinds of metrics and projects.

3. Study design

In our case study we study the measurement program at Ericsson where several measurement systems are used (over 200 at the time of studying). The concept of a measurement system has been adopted from the existing standards on metrology [25] where it is defined as a set of measuring instruments assembled in order to measure quantities of specific kinds. In the case of software engineering the quantities are dependent on the purpose of measurement and the measured entities. An entity can be a project, process, product, team, etc. and a quantity can be project length, number of activities in the process, lines-of-code in the product, team size, etc. The measurement systems built by the organization are developed according to the ISO/IEC 15939:2007 standard [27]. More details about the measurements are presented in subsection 3.2.

3.1. Sample

The sample in our study was chosen using convenience sampling with blocking: we asked experts with different roles:

- Stakeholder (1 person): A project manager for whom a measurement system was built. The project manager used the measurement system to monitor and control his project during the whole project execution.
- Manager (1 person): A section manager responsible for resources and competence.
- Quality manager (2 persons during 1 interview – i.e. 1 data point): Two quality managers working with measurement in the organization. They do not develop measurement systems, but are involved in their design and evaluation.
- Designer of measurement systems/quality manager (1 person): A quality manager responsible for designing, developing, and maintaining measurement systems in the organization. This manager was the most insightful into the details of how measurement systems are structured and about their limitations.

These roles covered all persons involved in establishing, development, and maintenance of both measurement programs and measurement systems. All interviewees have several years of experience with working with measurements at Ericsson.

3.2. Objects

The study object in this case study is the measurement program at one of the units of Ericsson which develops large products for the mobile telephony network. The size of the organization is several hundred engineers and the size of the projects can be between 80 and 200 engineers. Projects are more and more often executed according to the principles of Agile software development and Lean production system referred to as Streamline development (SD) within Ericsson [28]. A noteworthy fact is that in SD the releases are frequent and that there is always a release-ready version of the system: referred to as Latest System Version [28]. This means that the measurement program used in the organization was designed to monitor and control software development on a continuous basis as opposed to controlling projects which have beginning and end. The streamline development also posed requirements on measures – they should guide the operation of the Streamline development programs towards improvements during the execution, i.e. without the possibility of doing post-mortem analyses or baselining towards previous projects.

The measurement program was a continuous activity for a number of years and was constantly improved. The last year, however, the organization succeeded in establishing the 'measurement culture' in the organization and developed sev-

eral measurement systems according to ISO/IEC 15939 standard [27]. This standard contributed to establishing common measurement processes and vocabulary of indicators, base/derived measures, and information products. The studied organization complemented this standard with the ISO VIM (Vocabulary in Metrology, [25]) which contributed with the definitions and understanding of such concepts as measurement system, measuring instrument, base quantity, measurement process.

ISO/IEC 15939 was used to structure the measurement process at the studied organization and all documentation and information about it. In particular the web pages were named "Indicators", "Base/derived measures", "Measurement systems", etc. This ambient use of ISO/IEC 15939 quickly resulted in spreading the vocabulary of the standard in the organization.

ISO VIM standard was used to structure the information within the measurement systems (i.e. MS Excel files) and to provide definitions of the concepts measured. When possible the measurement team also reused definitions from ISO/IEC 25000 series of standards (Software Quality Requirements and Evaluation) and ISO/IEC 9126 [29].

The goal of the measurement program was to constantly improve the operational excellence of the unit of Ericsson w.r.t. productivity, product and process quality and technology leadership. The measurement program was designed using the ISO/IEC 15939:2002 (and later using :2007 edition) with the purpose to support stakeholders at multiple levels of organizations, for example:

- Project managers: to support them in monitoring the progress of the project and assisting them in addressing questions like "Will we finish on time?" or "How much resources do we need to maintain/improve the quality of the product?"
- Product managers/owners: to support them in monitoring and improving quality of products, i.e. assisting them in addressing questions like "How to achieve 0-defects at the release date?" or "Will we have good quality at <milestone>?"
- Line managers (at the section, department and unit level): to support them in monitoring the status of the organization and making long-term decisions about products, projects and competence in the organization, i.e. assisting them in addressing questions like: "Will we have enough resources to satisfy needs of <project X>?"

The measures used in the measurement program varied from management measures (e.g. financial) to technical (e.g. number of defects discovered during testing), and used at several levels of abstraction. We were able to study a number of measurement systems, e.g. measurement systems for :

- Measuring reliability of network products in operation for the manager of the product management organization; example measures in this measurement system are:
  - Product downtime per month in minutes
  - Number of nodes in operation
- Measuring project status and progress – for project managers who need to have daily updated information about such areas as requirements coverage in the project, test progress, costs, etc.; example measures in this measurement system are:
  - Number of work packages finished during the current week
  - Number of work packages planned to be finished during the current week
  - Number of test cases executed during the current week
  - Cost of the project up till the current date
- Measuring post-release defect inflow – for product managers who need to have weekly and monthly reports about the number of defects reported from products in field; examples of measures:
  - Number of defects reported from field operation of a product during the last month
  - Number of nodes in operation last month
  - Number of nodes which reported defects
- Summarizing status from several projects – for department manager who needs to have an overview of the status of all projects conducted in the organization, e.g. number of projects with all indicators "green"

These measurement systems were instantiated for a number of projects and products. Each of these instances had a distinct individual as stakeholder (in the role of project manager, product manager, etc.) who used the measurement system regularly.

Measures used in these measurement systems were both collected automatically from databases or manually from persons when the data is not stored in databases (e.g. by asking the project manager how many designers are assigned to remove defects from the software in a particular week, with detailed measures are described in [30]). The sources of information were defined in the measures specification and the infrastructure specification for the particular measurement systems (e.g.[31]).

The measures were designed using an in-house developed framework [32] based on the ISO/IEC 15939 standard. The framework was structured around the concepts of information product and indicator; the development of measurement systems started with discussions with stakeholders with two questions: "What do you need to know?" and "Why do you need to know it?" in the context of their management role. Model-Driven-Engineering approach was used when designing, implementing and validating measurement systems [31]. This approach has led to optimizing the number of data collected and the reduction from over 3000 measures to ca. 30 reusable (indicators).

The measurement program was built upon the concept of tools present in every desktop at the company – MS Office. Automated tools were built on top of MS Excel 2003 to collect data, perform measurements, store data, and present the most important information in form of indicators – all according to the ISO/IEC 15939:2007 standard. Detailed description of the technologies used behind this program are described by Staron and Meding [14].

3.3. Instruments

The main instrument used in our study was questionnaire which we used during the interviews. Another instrument was an interview with the measurement systems designer/quality manager. The questionnaire was originally used by Jeffery and Berry [33] as a means of predicting the success of a measurement program in industry. The analysis of answers to these questions and a further interview result in identifying the main factors which determined successful implementation of measurement program, in a similar way as identifying the factors in other industrial case studies [34, 35].

The questionnaires contained a list of questions; each of these was to be evaluated how well it was fulfilled. The evaluation was done by assigning a score on the scale 0 – 3, where 0 – this requirement is not fulfilled at all, 1 – this requirement is fulfilled to some extent, 2 – this requirement is fulfilled almost fully, and 3 – this requirements is completely fulfilled. This scale was according to the original questionnaire presented by Jeffery and Berry [33]. We modified the scale by adding N/A (Not Applicable) to the scale. An example question is presented in subsection 3.3.

We also added new questions, which were identified as factors important in successful implementation of measurement programs by [36]. All questions, including the ones added, were grouped according to the categories from the original paper [33]:

- Context(C) – questions about the background of the measurement program, the needs for it,
- Inputs(I) – questions about the input to the measurement program and its resources,
- Process – questions about the process of collecting measurements, process responsibilities and measure teams, with subcategories
  - Process motivation and objectives (PM)
  - Process responsibility and metrics team (PR)
  - Process and data collection (PC)
  - Process training and awareness (PT)
- Product(P)- questions about the measurements as products of the measurement process.

The full list of questions from the original questionnaire can be found in [33]. Our complete list of questions is presented below, and the

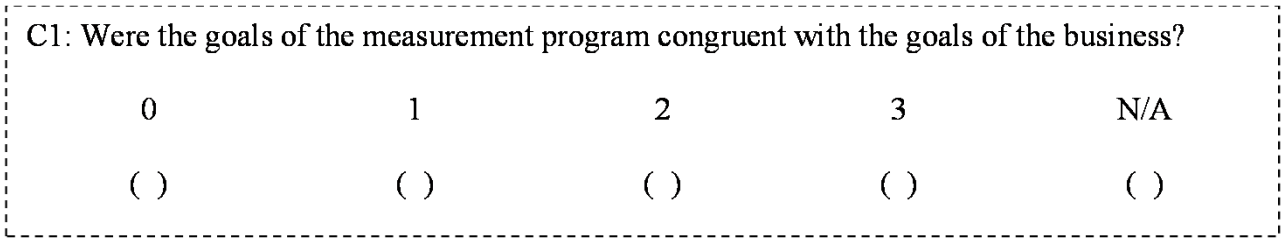


Figure 1. Example question in the questionnaire

added questions are annotated with (A) before the question:

- C1: Were the goals of the measurement program congruent with the goals of the business?
- C2: Could the measured staff participate in the development of the measures?
- C3: Had a quality environment been established?
- C4: Were the processes stable?
- C5: Could the required granularity be determined and was the data available?
- C6: Was the measurement program tailored to the needs of the organization?
- C7: Was senior management commitment available?
- C8: Were the objectives and goals clearly stated?
- C9: Were there realistic assessments of pay-back period (e.g. 2 years)?
- (A) C10: Was the process planned to be incrementally implemented?
- I1: Was the program resourced properly?
- I2: Were resources allocated to training?
- I3: Were at least three people assigned to the measurement program?
- I4: Was research done?
- (A) I5: Were existing metrics materials used?
- (A) I6: Was the data seen to have integrity?
- (A) I7: Was the data easy to collect collected?
- (A) I8: Was the data set determined incrementally?
- P1: Were the measures clear and of obvious applicability?
- P2: Did the end result provide clear benefits to the management process at the chosen management audience levels?
- P3: Was feedback on results provided to those being measured?
- P4: Was the measurement system flexible enough to allow for the addition of new techniques?
- P5: Were measures used only for pre-defined objectives?
- PC1: Were the important initial metrics defined?
- PC2: Were tools for automatic data collection and analysis developed?
- PC3: Was a metric database created?
- PC4: Was there a mechanism for changing the measurement system in an orderly way?
- PC5: Was measurement integrated into the process?
- PC6: Were capabilities provided for users to explain events and phenomena associated with the project?
- PC7: Was the data cleaned and used promptly?
- PC8: Did the objective determine the measures?
- (A) PC9: Was the measurement program constantly improved?
- PM1: Was the program promoted through the publication of success stories and encouraging exchange of ideas?
- PM2: Was a firm implementation plan published?
- PM3: Was the program used to assess the individuals?
- PR1: Was the metrics team independent of software developers?
- PR2: Were clear responsibility assigned?
- PR3: Was the initial collection of metrics sold to data collectors?

– PT1: Was adequate training in software metrics carried out?
– PT2: Did everyone know what was being measured and why?

The interviewees were not presented with additional material during the interview, as they understood the measurement program and had extensive experience with it.

As an addition to the questionnaire, we send a question to the designer of measurement systems/quality manager before the interview in order not to influence his answers by the questions in the questionnaire. The question was: "What are the most important factors that determine whether a measurement system is successfully implemented and used in the organization?" We deliberately narrowed the question to measurement system as we wanted to obtain information which covered the issues not addressed by the questionnaire.

In the end we performed also a workshop with the quality managers, section manager, and designer of measurement systems/quality manager where we presented the results and validated our findings.

3.4. Analysis methods

In the study we use descriptive statistics when analysing the results from the questionnaires. We provide a total percentage of score for each category from Section 3.3. The max score (i.e. 100%) is when all applicable questions are ranked as 3 (requirements are completely fulfilled) by all stakeholders (i.e. 3 * 4 = 12, and 12 is the 100% score for each questions applicable for all stakeholders). We do not account for non-equal variances in the descriptive statistics as we do not perform hypotheses testing methods that would require doing so.

To test for significant differences between roles, we use also the Friedman test [37]. Our hypotheses are:

– *H0*: There is no difference between roles.
– *H1*: There is a difference between roles.

Testing these hypotheses allows for assessing whether the different respondents perceived (assessed) the measurement program differently, or whether there is a consensus on how the program is implemented.

In order to further test for which questions the respondents were uniform and for which their answers were disperse, we use the hierarchical cluster analysis for between-variable (roles) and between-treatment (questions) clusters [38]. We use dendrograms for visualizing the results.

Using the cluster analysis provided us with the statistical means of suggesting groups of success factors. The suggested groups were then evaluated together with the study subjects whether they should be grouped into a more compound success factor.

4. Results and analysis

The results are presented in the following parts: (i) results from questionnaires, (ii) success factors identified by the designer of measurement systems/quality manager, and (iii) the list of success factors identified and generalized from both (i) and (ii).

4.1. Questionnaire results

The percentage of requirements fulfilled for each category is presented in table 1.

Table 1. Percentage of requirements fulfilled

Category	Number of questions	Score
Context	10	79%
Input	8	80%
Process	17	64%
Product	5	76%

The table shows that the input and context are categories with the requirements fulfilled to the largest extent. The process is the category with requirements fulfilled to the least extent. This seems to be natural as the organization and its measurement program constantly evolves, and so do the measurement processes. The summarizing descriptive statistics per respondent are presented in table 2.

The descriptive statistics show that stakeholder was the most positive respondent, which

Table 2. Descriptive statistics per respondent

Respondent	Median	Number of 3's	Number of 0's
Designer of measurement systems/Quality manager	2	19	7
Quality manger	2	14	6
Stakeholder	3	25	2
Section manager	3	21	1

was a desired effect (since the 'survival' of the measurement program depends on stakeholders using the measurement systems). After the presentation of these results the designer of measurement systems/quality manager provided us with feedback on his low assessment results. The results were caused by the designer of measurement system/quality manager having a complete picture of the further work to improve the existing measurement program in the company.

The Friedman test resulted in rejecting the null hypothesis with the p-value of 0.00042. With the total number of questions over 30, the β-value was below 0.05. Having rejected the null hypothesis we can conclude that the respondents had different view on the measurement program and perform the hierarchical cluster analysis.

The hierarchical cluster analysis for between-variables (roles) clusters results in the dendrogram presented in figure 2.

The dendrogram shows that the quality manger(s) and the section manager have the most similar opinions. The stakeholder's opinion was the least similar to the rest of the respondents. A closer analysis (indicated in Table 2) showed that the stakeholder was more positive than other respondents to the measurement program and its fulfilment of requirements. This, in turn indicated that the organization was successful in spreading the measurement systems and establishing the measurement program.

The hierarchical cluster analysis for between-treatments (questions) cluster results in the dendrogram presented in Figure 3.

The results show that there are questions where the different respondents do not agree – e.g. question 21. After a closer analysis we found that these are the questions about aspects not familiar to some of the respondents – e.g. stakeholder (project manager) was not aware that we have a large metrics database. An example of a group of questions where the respondents agreed is: PR1, PR2, C5, C6, I5, I6, and I8 (in the middle of the figure). A closer analysis revealed that these were the questions which scored 3 (the top rank) by all stakeholders.

4.2. Measurement Systems Designer's perception: success factors

The list of factors which are identified as important by the designer concerned the way in which measurement systems are developed and deployed in the organization. These factors were not added to the questionnaire, because they were at a much lower level than the questionnaire – they concerned technical aspects of building measurement systems and measuring instruments rather than establishing a measurement program in the organization.

The measurement systems designer/quality manager identified the following factors (without prioritizing them):

1. Work according to the standards (also identified in [39]), which is important as it ensures that:
   a) all measurement systems are built and presented in the same way
   b) there is a well known nomenclature regarding measurement systems
   c) all steps regarding building and maintaining of measurement systems are well defined.
   d) ISO/IEC 15939 is a very solid standard that is recommended for Software Engineering.
2. Always providing certain base measures, e.g. defect statistics for projects and products.
   a) Using standards like ISO/IEC 25000 (SQUARE) is recommended.
3. Definition and use of a known process to get information about all main elements of a measurement system (e.g. stakeholder, information need, indicators). In particular there

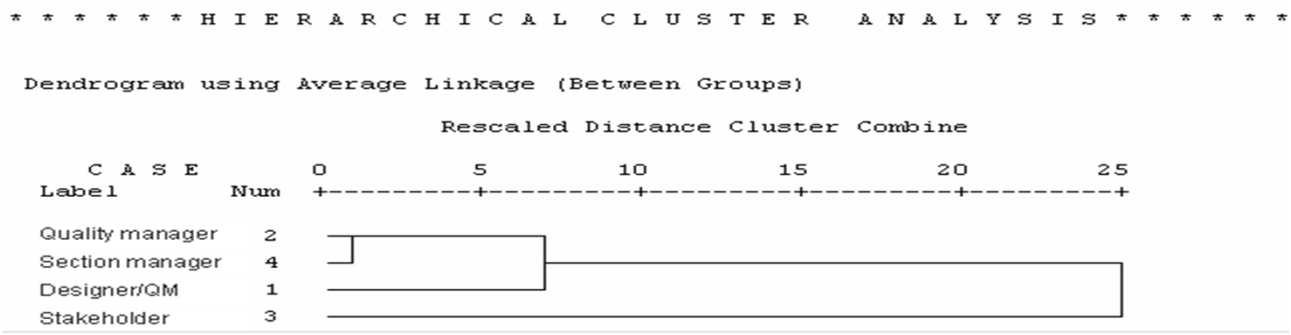


Figure 2. Dendrogram for between-variables clusters

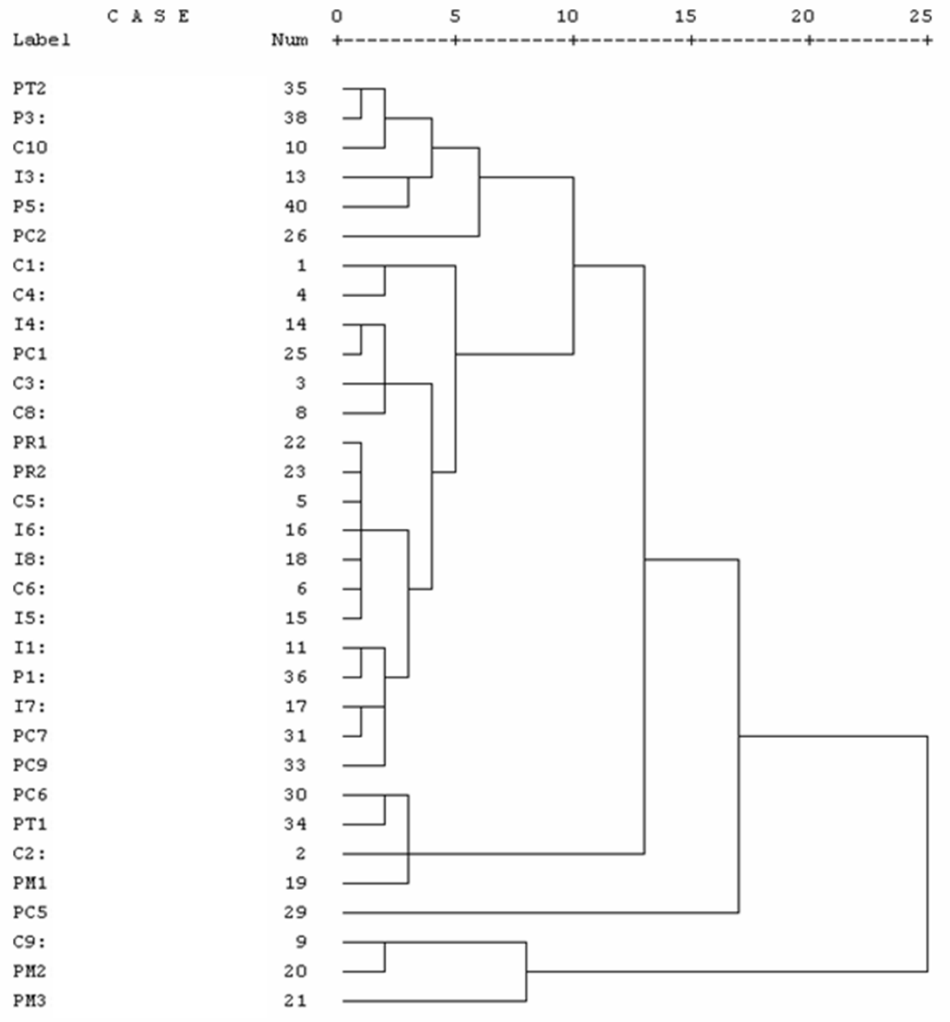


Figure 3. Dendrogram for between-treatment clusters

should always be a stakeholder for the measurement system
   a) The stakeholder should have a real and legitimate power in the project – e.g. project manager or section manager. Otherwise there is a real risk of waste, i.e. measurements are not used for decision making.
4. Specify and implement measurement systems in a constant way, e.g. logical and physical views of architectural design, implementation technology, and/or knowledge base.
   a) maintain the infrastructure and measures so that it can be deployed on large scale
5. Use pre-defined infrastructure and allocated areas for storing measures and information about the measures (define the measurements database).
   a) It is important to keep the values of measures for future use and future analyses. Using simple databases with structure of information in accordance to ISO/IEC 15939 is recommended.
6. Present the main information (e.g. indicators) in a simple, non-ambiguous, and succinct manner
   a) present details in another place, which is linked from the main information presentation
   b) Gadget in MS Windows Vista/7 or Widgets for MacOS are recommended since they provide the stakeholders with information without the need for them to be active (for example, please see [30]).
7. Ensure reliability of the measurement system – provided information should be reliable and up-to-date.
   a) We recommend using indicators of information quality [40].
8. Ensure that the necessary knowledge is in place (for details see also [14])
   a) stakeholders should know how to interpret the information and make adjustments to measurement systems
   b) designers of measurement systems should know the standards and implementation technology for the measurement systems.

The above factors are related to how measurement systems are built and deployed in the organization. They have an effect on the measurement program, to which other factors apply as well.

4.3. Success factors

In this section we focus on the factors, which have not been identified previously, and do not re-consider the importance of such factors as:

– Management commitment [6]: Measurement program as a "shadow" activity of employees without management support stand no chances of success as it is the managers who decide whether new methods/tools/ways of working are introduced or not. When we designed the first measurement systems the commitment was rather hard to obtain. The turning point came when we showed the results of our predictions to one of the project managers and his response was "If these predictions are correct, then we cannot let this happen"; this was followed by his actions to adjust resources and avoiding problems in the project. This fist "success" helped us to get strong commitment from the project manager and in turn (gradually) from other project managers and line managers.
– Team commitment [6]: Without the commitment from the team being measured the information quality might be low, which jeopardizes the reliability of the data. In the case of the studied organization the team commitment was obtained after about 1 year of using measurement systems for making decisions for one project. The team has realized that the measurements help them to visualize the goal and achieve it.
– Making measurements part of processes [41]: Putting new burdens on persons in the organization is never popular and should be avoided. It is much better to use 'probes' which measure in-process data from the tools already used at the organization. This minimizes the threat that other activities are prioritized over measuring for the persons being measured. In our case this was reduced by using automation based on MS Excel. Since everyone in the organization knew MS Excel

virtually no learning was involved; automation reduced even the burden of processing and presenting the information (see [14]).

We see the above factors being prerequisites for a successful program and these factors were present in the studied organization. What we have observed in the organization was the gradual (over ca. 2 years) change of culture. The concept of "main measures" was discussed in the organization at the beginning whereas in the end only the indicators were considered.

Table 3 presents factors which we identified as important when implementing measurement programs when performing the program evaluation at Ericsson. These factors are important for different roles, which is indicated by a cross in the column denoting particular stakeholder (D/QM – Designer/Quality manager; QM – Quality manager; SH – Stakeholder, SM – Section manager).

The above factors have already been identified and they are mostly related to the process of establishing the measurement program. After being established, the program needs to be maintained in order not to be dropped. Therefore we identify the following:

1. *Working according to the ISO/IEC 15939 standard*: A standardized nomenclature (ISO/IEC 15939 [27] and ISO/IEC Vocabulary on Metrology [25]), terminology and proven processes are key factors in the long-term adoption. Using standards make the effort less person-dependent and interpretation dependent. It makes reuse across organizations easier, as also indicated in [43]. In our case we follow: ISO/IEC 15939:2007, ISO/IEC Vocabulary on Metrology, and ISO/IEC 9126.
2. *Providing information quality indicators*: Information is as good as it is reliable and up-to-date. Providing information, especially automatically should also indicate the quality of the information provided. An existing model can be used (e.g. [44, 45]) or a dedicated one can be developed. The issues to address when indicating information quality are: providing the data which is up-to-date, correctly processed, complete, and unbiased. In our work we use the following indicators of information quality:
   a) Timeliness (the information presented to the stakeholder is up-to-date, e.g. from today, this month, or current – depending on the purpose),
   b) Completeness (the information contains no missing values),
   c) Correctness (there were no errors in calculation),
   d) Accuracy (the data sources contain the updated information)
3. *Automated data collection based on simple software tools* (also identified in [46]): measures should be collected automatically to minimize the burden of data collection to the (usually) already busy organization. If not automated the program will eventually be rejected. In our work we use MS Excel and Visual Basic for Applications to automate the data collection and processing. By developing measurement systems, the organization gains competence on working with measures and does not rely on external entities when building and maintaining the measures.
4. *Individual stakeholders for each measurement system*: (related to "Use in decision making" from [6]): there is one role/individual in the organization whose information need is satisfied with the measurement system (a.k.a. producing data inside their range of validity as identified in [46]; identified also in [47] as using different strokes for different people). If this is not the case, then the measurements are not used in the decision process and thus become ineffective. Stakeholders should be able to adjust the measurements to the situations that can happen over time (e.g. by adjusting decision criteria for indicators).
5. *Direct benefits to the organization*: The results from the measurement program should be applicable in the organization "now" and not after a period of time. The most current activities are usually prioritized, and benefiting from measures in decision process depends on using current data to satisfy current information needs.

Table 3. Factors important for long-term success identified in our study

Factor	D/QM	QM	SH	SM
Congruence of measurement goals with business goals	x	x		
Incremental implementation of the program	x			x
Participation of measured staff in program development	x		x	
Quality environment	x		x	
Process stability	x			
Availability of data at the required granularity	x	x	x	x
Tailoring measurements to organization needs	x	x	x	x
Clear objectives and goals (also in [1])	x		x	x
Proper program resourcing (proper metric team)		x	x	
Conducting research prior to/during measurement program development	x		x	x
Using existing metric materials	x	x	x	x
Integrity of the data	x	x	x	x
Using existing data for processing		x	x	x
Data set determined incrementally	x	x	x	x
Clear measures of obvious applicability			x	x
Clear benefits for the management process	x			
Providing feedback to those being measured			x	
Flexibility for adding new measurements (also in [1])	x	x		x
Pre-defined objectives for the measures			x	
Initial definition of important metrics (also in [42])	x		x	x
Automatic data collection and processing (also in [1, 42])	x	x	x	x
Metric database	x	x		x
Mechanisms for adjusting measurement systems to changing needs	x			
Integrating measurement into the process	x			
Stakeholders are able to explain the meaning of metrics values	x		x	
Using data in clean and prompt way (also in [42])	x	x	x	x
Measures are determined by objectives	x		x	x
Constant improvement of the measurement program	x	x	x	x
Independent metric team (from developers)	x	x	x	x
Clear assignment of responsibilities	x	x	x	x
Adequate training in software metrics	x		x	

6. *Devoted measurement team*: the measurements are collected throughout the organization, but there is a team of specialists who help to define and introduce measurements. These specialists are also responsible for maintenance of the measurement program. Evidence of such a team being a positive factor has also been found when introducing modelling into large organizations [48], which, although seems unrelated, is similar to introducing measures (as a new way of working). In the case of the studied organization the measurement team consists of quality managers, section managers, technology specialists and researchers – which is similar to the team of specialists when introducing models – modelling specialists, technology specialists and researchers.
7. *Measurement collection effort should be minimal*: (also identified in [46, 47]), which means that using already collected data (at least initially) is a good point. Every organization collects data from their processes (e.g. such high level data as project cost), and such data should be used when the measurement program is being established to show that measurement programs provide positive support. After the measurement program has been adopted, the measures should be refined to optimize the data collection and fulfilment of stakeholders' information needs.
8. *Providing standard base measures*: Certain base measures, e.g. product performance,

should always be provided if applicable to support benchmarking and reuse. However, the number of measures provided in this way should be optimized w.r.t. needs and costs for collecting them. Example base measures provided in the studied organization are: In-Service-Performance, resource allocation, number of work packages completed.

9. *Reusing base measures*: Costs of measures collection should be optimized and measures should be aimed at being reused. Therefore the measures should be specified, described, and stored in repositories which would enable reusing them – e.g. for benchmarking or measuring improvements over time.
10. *Using measures specifications and specification of their instantiation*: The measures are specified in relation to the kinds of measured entities – e.g. measures of a project (one single project), like number of designers. These measures are then instantiated for different projects. The distinction is important since measurement systems might be different because they measure different projects (entities) or measure projects and processes (different kinds of entities).
11. *Do not use the program to assess individuals*: It is important not to create negative attitude to the program (a.k.a. Fear of adverse consequences in [47, 1]) by creating situations that measurements are to assess the work/performance of individuals.

The above factors are ordered according to their importance – factors 1 being the most important one.

5. Validity evaluation

We identify the threats to validity of our study using the categories presented by Wohlin et al. [49].

The main external validity threat of our study is the fact that we studied only a single organization. However, the found success factors are consistent with the trends observed in literature and do not seem to be organization or process specific. The underlying technology for implementing automation is based on MS Excel which is used in almost every company and is not an Ericsson-specific tool. The add-ons for Excel with measurement instruments are specific, but these do not influence the generalizability of the results.

The main construct validity threat is related to mono-operation bias, which is a bias introduced by observing a single phenomenon at a single point of time and thus not capturing the full breadth of the phenomenon. This is a typical threat to operationalizations in single-case case studies. Our research is a summary of a 2 year action research project research and the respondents in the study were involved in measurement activities for a number of years.

The main threat to the internal validity of our findings is the maturation effect as it was a 2 year project. Naturally this is a threat, but to some extent the maturity effect is desired in studies like this. The primary goal of our action research project was not to observe whether the measurement program was correct, but to establish and maintain a measurement program. In this manner, the maturity effect is a desired "cultural change" effect in the organization.

Finally, the main threat to conclusion validity is related to the fact that we have not used grounded theory to analyze interview material, but rather asked direct questions to the respondents and the interviewee. It was a deliberate choice since the authors were part of the team establishing the measurement program and we had this opportunity to reduce the 'noise' in the interview data by asking direct questions and using experience to reason about the answers. We use the statistical analysis when possible to evaluate the significance of some of the claims we made.

6. Conclusions

Software development projects are entities where change is prevalent and constant adaptations are predominant – especially if the projects are to meet their goals and deliver quality software. A long-term success of a measurement program

requires its constant adaptation towards the change in software projects, a situation unlike in manufacturing industries. The studied organization has chosen not to use GQM in order to be more flexible when adopting their measurement program and take advantage of adjusting interpretations of measures (embedded in the concept of indicator) and to be able to combine the ISO/IEC 15939 standard with measurement theory from other engineering disciplines. The decision to remain independent from tool vendors and do not purchase off-the-shelf solution provided the organization with ability to remain the core measurement competence in-house, and hence be more reactive to changing needs of the organization.

The organization combined three key elements when establishing and maintaining the measurement program: the use of international standards, significant experience base, and research activities. This combination contributed to the success of a measurement program constantly grows in the organization. By including researchers in the process of developing, establishing, and maintaining both the measurement program and the measurement systems, the company benefited from external competence, but did not rely on external entities to establish the program. This elevated the competence of the measurement team and resulted in publications related to measures, e.g. [50].

In this paper we described factors contributing positively to the success of a long-term measurement program. These factors are based on the experience of the team working with the measurement program and have been obtained through interviews and surveys.

Our further work is focused on observing threats to the working measurement program and identifying these threats over a longer period of time (at least 3 years). Identifying such threats would help to prevent withdrawing from the measurement program in the organizations.

7. Acknowledgements

The project has been partially sponsored by the Swedish Strategic Research Foundation (www.stratresearch.se) under the program Mobility in IT. It was also partially sponsored by Ericsson Software Architecture Quality Center and Ericsson Software Research.

We would like to thank the involved managers and engineers at Ericsson for their support in this study.

References

[1] M. Umarji and H. Emurian, "Acceptance issues in metrics program implementation," in *11th IEEE International Symposium Software Metrics*, H. Emurian, Ed., 2005, pp. 10–29.

[2] K. L. Atkins, B. D. Martin, J. M. Vellinga, and R. A. Price, "Stardust: implementing a new manage-to-budget paradigm," *Acta Astronautica*, vol. 52, no. 2-6, pp. 87–97, 2003, tY - JOUR.

[3] T. Kilpi, "Implementing a software metrics program at nokia," *IEEE Software*, vol. 18, no. 6, pp. 72–77, 2001.

[4] B. Clark, "Eight secrets of software measurement," *IEEE Software*, vol. 19, no. 5, pp. 12–14, 2002.

[5] A. Gopal, T. Mukhopadhyay, and M. S. Krishnan, "The impact of institutional forces on software metrics programs," *IEEE Transactions on Software Engineering*, vol. 31, no. 8, pp. 679–694, 2005, 0098-5589.

[6] A. Gopal, M. S. Krishnan, T. Mukhopadhyay, and D. R. Goldenson, "Measurement programs in software development: determinants of success," *IEEE Transactions on Software Engineering*, vol. 28, no. 9, pp. 863–875, 2002, 0098-5589.

[7] J. Lawler and B. Kitchenham, "Measurement modeling technology," *IEEE Software*, vol. 20, no. 3, pp. 68–75, 2003, 0740-7459.

[8] F. Niessink and H. van Vliet, "Measurement program success factors revisited," *Information and Software Technology*, vol. 43, no. 10, pp. 617–628, 2001, tY - JOUR.

[9] ——, "Measurements should generate value, rather than data," in *6th International Software Metrics Symposium*, 2000, pp. 31–38.

[10] S. De Panfilis, B. Kitchenham, and N. Morfuni, "Experiences introducing a measurement program," *Information and Software Technology*, vol. 39, no. 11, pp. 745–754, 1997, tY - JOUR.

[11] P. Goodman, *Practical implementation of software metrics*, ser. International software quality assurance series. London: McGraw-Hill, 1993, lc92042989 Paul Goodman.

[12] K. H. Moeller, *Software metrics: a practitioner's guide to improved product development*. London: Chapman-Hall, 1993.

[13] M. Diaz-Ley, F. Garcia, and M. Piattini, "Implementing a software measurement program in small and medium enterprises: a suitable framework," *IET Software*, vol. 2, no. 5, pp. 417–436, 2008.

[14] M. Staron, W. Meding, and C. Nilsson, "A framework for developing measurement systems and its industrial evaluation," *Information and Software Technology*, vol. 51, no. 4, pp. 721–737, 2008.

[15] M. Jorgensen, "Software quality measurement," *Advances in Engineering Software*, vol. 30, no. 12, pp. 907–912, 1999.

[16] D. Wisell, P. Stenvard, A. Hansebacke, and N. Keskitalo, "Considerations when designing and using virtual instruments as building blocks in flexible measurement system solutions," in *IEEE Instrumentation and Measurement Technology Conference*, P. Stenvard, Ed., 2007, pp. 1–5.

[17] A. Sehmi, N. Jones, S. Wang, and G. Loudon, "Knowledge-based systems for neuroelectric signal processing," *IEE Proceedings-Science, Measurement and Technology*, vol. 141, no. 3, pp. 215–23, 2003.

[18] J. Feigin and K. Pahlavan, "Measurement of characteristics of voice over ip in a wireless lan environment," in *IEEE International Workshop on Mobile Multimedia Communications*, K. Pahlavan, Ed., 2003, pp. 236–240.

[19] M. Foote and D. Horn, "Video measurement of swash zone hydrodynamics," *Geomorphology*, vol. 29, no. 1-2, pp. 59–76, 1999, tY - JOUR.

[20] N. P. Kolev, S. T. Yordanova, and P. M. Tzvetkov, "Computerized investigation of robust measurement systems," *Instrumentation and Measurement, IEEE Transactions on*, vol. 51, no. 2, pp. 207–210, 2002, 0018-9456.

[21] R. F. Kunz, G. F. Kasmala, J. H. Mahaffy, and C. J. Murray, "On the automated assessment of nuclear reactor systems code accuracy," *Nuclear Engineering and Design*, vol. 211, no. 2-3, pp. 245–272, 2002, tY - JOUR.

[22] A. N. Zaborovsky, D. O. Danilov, G. V. Leonov, and R. V. Mescheriakov, "Software and hardware for measurements systems," in *The IEEE-Siberian Conference on Electron Devices and Materials*, D. O. Danilov, Ed. IEEE, 2007, pp. 53–57.

[23] H. Zhiyao, W. Baoliang, and L. Haiqing, "An intelligent measurement system for powder flowrate measurement in pneumatic conveying system," *IEEE Transactions on Instrumentation and Measurement*, vol. 51, no. 4, pp. 700–703, 2002, 0018-9456.

[24] G. Kai, "Virtual measurement system for muzzle velocity and firing frequency," in *8th International Conference on Electronic Measurement and Instruments*, 2001, pp. 176–179.

[25] *International vocabulary of basic and general terms in metrology = Vocabulaire international des termes fondamentaux et généraux de métrologie*, 2nd ed. Genève, Switzerland: International Organization for Standardization, 1993.

[26] "Tychometrics," Predicate Logic, 2007.

[27] *ISO/IEC 15939:2007 Systems and software engineering – Measurement process*, International Standard Organization and International Electrotechnical Commission Std., 2007.

[28] P. Tomaszewski, P. Berander, and L.-O. Damm, "From traditional to streamline development - opportunities and challenges," *Software Process Improvement and Practice*, vol. 2007, no. 1, pp. 1–20, 2007.

[29] *ISO IEC 9126, Software engineering, Product quality Part: 1 Quality model*, International Standard Organization / International Electrotechnical Commission Std., 2001.

[30] M. Staron and W. Meding, "Defect inflow prediction in large software projects," *e-Informatica Software Engineering Journal*, vol. 4, no. 1, pp. 1–23, 2010.

[31] ——, "Using models to develop measurement systems: A method and its industrial use," vol. 5891, pp. 212–226, 2009.

[32] M. Staron, W. Meding, G. Karlsson, and C. Nilsson, "Developing measurement systems: an industrial case study," *Journal of Software Maintenance and Evolution: Research and Practice*, pp. n/a–n/a, 2010.

[33] R. Jeffery and M. Berry, "A framework for evaluation and prediction of metrics program success," pp. 28–39, 1993.

[34] M. Staron, L. Kuzniarz, and L. Wallin, "Factors determining effective realization of mda in industry," in *2nd Nordic Workshop on the Unified Modeling Language*, K. Koskimies, L. Kuzniarz, J. Lilius, and I. Porres, Eds., vol. 35. Abo Akademi, 2004, pp. 79–91.

[35] ——, "A case study on industrial mda realization - determinants of effectiveness," *Nordic Journal of Computing*, vol. 11, no. 3, pp. 254–278, 2004.

[36] T. Hall and N. Fenton, "Implementing effective software metrics programs," *Software, IEEE*, vol. 14, no. 2, pp. 55–65, 1997, 0740-7459.

[37] D. Altman, *Practical Statistics for Medical Research*. Chapman-Hall, 1991.

[38] D. F. Morrison, *Multivariate statistical methods*, 3rd ed., ser. McGraw-Hill series in probability and statistics. New York: McGraw-Hill, 1990.

[39] L. Westfall, "Are we doing well, or are we doing poorly?" p. 20, 2005.

[40] M. Staron and W. Meding, "Ensuring reliability of information provided by measurement systems," vol. 5891, pp. 1–16, 2009.

[41] C. A. Dekkers and P. A. McQuaid, "The dangers of using software metrics to (mis)manage," *IT Professional*, vol. 4, no. 2, pp. 24–30, 2002, 1520-9202.

[42] J. D. Herbsleb and R. E. Grinter, "Conceptual simplicity meets organizational complexity: case study of a corporate metrics program," in *Software Engineering, 1998. Proceedings of the 1998 International Conference on*, R. E. Grinter, Ed., 2003, pp. 271–280.

[43] F. Garcia, M. F. Bertoa, C. Calero, A. Vallecillo, F. Ruiz, M. Piattini, and M. Genero, "Towards a consistent terminology for software measurement," *Information and Software Technology*, vol. 48, no. 8, pp. 631–644, 2006.

[44] Z. R. Pendic, L. Kovacevic, and J. Stupar, "An approach to evaluation of quality of integrated information systems," *Annual Review in Automatic Programming*, vol. 14, no. Part 2, pp. 63–68, 1988.

[45] B. K. Kahn, D. M. Strong, and R. Y. Wang, "Information quality benchmarks: Product and service performance," *Communications of the ACM*, vol. 45, no. 5, pp. 184–192, 2002.

[46] F. J. Buckley, "Standards-establishing a standard metrics program," *Computer*, vol. 23, no. 6, pp. 85–86, 1990, 0018-9162.

[47] S. L. Pfleeger, "Lessons learned in building a corporate metrics program," *Software, IEEE*, vol. 10, no. 3, pp. 67–74, 1993, 0740-7459.

[48] P. Baker, S. Loh, and F. Weil, "Model-driven engineering in a large industrial context - a motorola case study," in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, vol. 3713, 2002, pp. 476–491.

[49] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, and A. Wesslèn, *Experimentation in Software Engineering: An Introduction*. Boston MA: Kluwer Academic Publisher, 2000.

[50] M. Staron and W. Meding, "Predicting weekly defect inflow in large software projects based on project planning and test status," *Information and Software Technology*, p. (available online), 2007.

e-Informatica Software Engineering Journal, Volume 5, Issue 1, 2011, pages: 25–37, DOI 10.2478/v10233-011-0028-y
VERSITA

Examining Correlations in Usability Data to Effectivize Usability Testing

Jeff Winter*, Mark Hinely**
*School of Computing, Blekinge Institute of Technology
**, UIQ Technology AB
jeff.winter@bth.se, mark.hinely@bredband.net

Abstract

Based on a case study performed in industry, this work deals with a statistical analysis of data collected during usability testing. The data is from tests performed by usability testers from two companies in two different countries. One problem in the industrial situation is the scarcity of testing resources, and a need to use these resources in the most efficient way. Therefore, the data from the testing is analysed to see whether it is possible to measure usability on the basis of one single metric, and whether it is possible to judge usability problems on the basis of the distribution of use case completion times. This would allow test leaders to concentrate on situations where there are obvious problems. We find that it is not possible to measure usability through the use of one metric, but that it may be possible to gain indications of usability problems on the basis of an analysis of time taken to perform use cases. This knowledge would allow the collection of usability data from distributed user groups, and a more efficient use of scarce testing resources.

1. Introduction

The background to this study is the situation faced by companies developing and testing consumer products for a mass market. The study is based on a long research cooperation between Blekinge Institute of Technology (BTH) and UIQ Technology AB (UIQ), an international company established in 1999. UIQ, who developed and licensed a user interface platform for mobile phones, identified a need to develop a flexible test method for measuring the usability of mobile phones, to give input to design and development processes, and to present usability findings for a number of stakeholders at different levels in the organization. This need resulted in the development of UIQ Technology Usability Metrics (UTUM). UTUM was successfully used in operations at UIQ until the closure of the company in 2009.

Together with UIQ we found that there is a need for methods that can simplify the discovery of usability problems in mobile phones. There is also a desire to find ways of identifying usability problems in phones without having to engage the test leader in every step of the process, with the ability to do it for geographically dispersed user groups. However, we also realise that even if it is found to be possible to identify problem areas, for example through a simple measurement of one metric, or through an analysis of completion times, this would not identify the particular aspects of the use cases that are problematic for the users. It would simply indicate use cases where the users experienced problems. This means that further studies would still have to be performed by test leaders together with users, to examine and understand what the actual problems consist of, and how they affect the way that users experience the use of the phone. This must be done

in order to create design solutions to alleviate the problems.

As we discuss in greater detail in section 3 of this article, the role of the usability tester is central in many ways, and it is a role that is not easily filled. It demands particular personal qualities, knowledge and experience. It involves the ability to communicate with people on many organisational levels, the ability to observe, record and analyse the testing process, and the ability to present the results of testing to many different stakeholders. Since there is a scarcity of people who can fill this role, it would ease the situation for companies wanting to perform usability testing if these resources could be used in the most efficient way possible. This is the principle behind the need to identify problematic use cases without having to involve the test leader in every step of the process.

If it is possible to identify use cases that are problematic, without requiring the presence of the test leader, this will allow companies to pinpoint which areas require further testing, so that test leaders can work more efficiently. Since we are working with a mass-market product, being able to do this remotely, for widely dispersed groups, would also be an advantage for the company, in order to test solutions in different geographical areas without requiring the usability tester to travel to these areas before there is seen to be a need, and to reduce the amount of testing that needs to be done on-site.

These needs are the basis of this article. In this work, we examine the metrics collected in the UTUM testing, to study the correlations between the metrics for efficiency, effectiveness, and satisfaction, to see whether we can measure usability on the basis of one metric, and we examine whether it is possible to develop a simple method of automatically identifying problem areas simply by measuring and analysing the time taken to perform different use cases.

2. Research Questions

The aim of this study is to examine whether there is a simple measurement to express usability, and to find if it is possible to streamline the discovery of problematic use cases. To do this, we examine the correlation between metrics for efficiency, effectiveness and satisfaction that have been collected during the testing process. These are the different elements of usability as specified in ISO 9241-11:1998 [1],). This is done in order to see whether there are correlations that allow us to discover usability problems on the basis of a simple metric. To satisfy the needs within industry, this metric should preferably be one that can easily be measured without the presence of the test leader. Based on this situation, we have formulated two research questions:

- **RQ1**: What is the correlation between the different aspects of usability (Effectiveness, Efficiency and Satisfaction)?
- **RQ2**: Can a statistical analysis of task-completion time allow us to discover problematic use cases?

The first research question is based on the idea that there may be a sufficiently strong correlation between the 3 factors of usability that measuring one of them would give a reliable indication of the usability of a mobile phone. The second research question is based on the theory that there is an expected distribution of completion times for a given use case and that deviations from goodness of fit indicate user problems.

This study is a continuation of previous efforts to examine the correlations between metrics for efficiency, effectiveness and satisfaction. A previous study by Frøkjær et al [2] found only weak correlations between the different factors of usability, whereas a study by Sauro [3] showed stronger correlations between the different elements. The results of this study will be placed in relation to these studies, to extend knowledge in the field. This is also a continuation of our previous work, where we have examined how the UTUM test contributes to quality assurance, and how it balances the agile and plan-driven paradigms (see e.g. [4, 5, 6, 7]).

3. Usability and the UTUM Test

UTUM is an industrial application developed and evolved through a long term cooperation between BTH and UIQ. UTUM is a simple and flexible usability test framework grounded in usability theory and guidelines, and in industrial software engineering practice and experience.

According to ISO 9241-11:1998 [1], Usability is the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use. Effectiveness is the accuracy and completeness with which users achieve specified goals. Efficiency concerns the resources expended in relation to the accuracy and completeness with which users achieve goals. Satisfaction concerns freedom from discomfort, and positive attitudes towards the use of the product.

UTUM measures the usability of products on a general level, as well as on a functional level. According to Hornbæk [8], amongst the challenges when measuring usability are to distinguish and compare subjective and objective measures of usability, to study correlations between usability measures as a means for validation, and to use both micro and macro tasks and corresponding measures of usability. Emphasis is also placed on the need to represent the entire construct of usability as a single metric, in order to increase the meaningfulness and strategic importance of usability data [3]. UTUM is an attempt to address some of these challenges.

An important characteristic of the UTUM test is the approach to understanding users and getting user input. Instead of simply observing use, a test expert interacts and works together with the users to gain insight into how they experience being a mobile phone user, in order to gain an understanding of the users' perspective. Therefore, users who help with UTUM testing are referred to as testers, because they are doing the testing, rather than being tested. The representative of the development company is referred to as the test leader, or test expert, emphasising the qualified role that this person assumes.

The test experts are specialists who bring in and communicate the knowledge that users have, in accordance with Pettichord [9], who claims that good testers think empirically in terms of observed behaviour, and must be encouraged to understand customers' needs. Evidence in Martin et al [10] suggests that drawing and learning from experience may be as important as taking a rational approach to testing. The fact that the test leaders involved in the testing are usability experts working in the field in their everyday work activities means that they have considerable experience of their products and their field. They have specialist knowledge, gained over a period of time through interaction with end-users, customers, developers, and other parties that have an interest in the testing process and results. However, these demands placed on the background and skills of test leaders mean that these types of resources are scarce, and must be used in the most efficient way possible.

A second characteristic of UTUM is making use of the inventiveness of phone users, by allowing users to participate actively in the design process. The participatory design tradition [11] respects the expertise and skills of the users, and this, combined with the inventiveness observed when users use their phones, means that users provide important input for system development. The test expert has an important role to play as an advocate and representative of the user perspective. Thus, the participation of the user provides designers, with the test expert as an intermediary, with good user input throughout the development process.

The user input gained through the testing is used directly in design and decision processes. Since the tempo of software development in the area of mobile phones is high, it is difficult to channel meaningful testing results to recipients at the right time in the design process. To address, this problem, the role of the test expert has been integrated into the daily design process. UTUM testing is usually performed in-house, and results of testing can be channelled to the most critical issues. The continual process of testing and informal relaying of testing results to designers leads to a short time span between discovering a problem and implementing a solution.

The results of testing are summarised in a clear and concise fashion that still retains a focus on understanding the user perspective, rather than simply observing and measuring user behaviour. The results of what is actually qualitative research are summarised by using quantitative methods. this gives decision makers results in the type of presentations they are used to dealing with. Statistical results are not based on methods that supplant the qualitative methods that are based on PD and ethnography, but are ways of capturing in numbers the users' attitudes towards the product they are testing.

A UTUM test does not take place in a laboratory environment, but should preferably take place in an environment that is familiar to the person who is participating in the test, in order that he or she should feel comfortable. When this is not possible, it should take place in an environment that is as neutral as possible. Although the test itself usually takes about 20 minutes, the test leader books one hour with the tester, in order to avoid creating an atmosphere of stress. The roles in testing are the test leader, who is usually a usability expert, and the tester.

In the test, the test leader welcomes the tester, and tries to put the tester at their ease. This includes explaining the purpose of the test, and saying that it is the telephone that is being tested, not the performance of the tester. The tester is instructed to tell the test leader when she or he is ready to begin the use case, so that the test leader can start the stopwatch to time the use case, and the tester should also tell the test leader when the use case is complete.

The tester begins by filling in some of their personal details and some general information about their phone usage. This includes name, age, gender, previous telephone use, and other data that can have an effect on the result of the test, such as which applications they find most important or useful. In some circumstances, this data can also be used to choose use cases for testing, based on the tester's use patterns.

For each phone to be tested, the tester is given time to get acquainted with the device. If several devices are to be tested, all of the use cases are performed on one device before moving on to the next phone. The tester is given a few minutes to get acquainted with the device, so that he or she can get a feeling for the look and feel of the phone. When this has been done, the tester fills in a Hardware Evaluation, a questionnaire based on the System Usability Scale (SUS) [12] about attitudes to the look and feel of the device. The SUS was developed in 1986 by John Brooke, then working at the Digital Equipment Company. The SUS consists of 10 statements, where even-numbered statements are worded negatively, and odd-numbered statements are worded positively.

1. I think that I would like to use this system frequently.
2. I found the system unnecessarily complex.
3. I thought the system was easy to use.
4. I think that I would need the support of a technical person to be able to use this system.
5. I found the various functions in this system were well integrated.
6. I thought there was too much inconsistency in this system.
7. I would imagine that most people would learn to use this system very quickly.
8. I found the system very cumbersome to use.
9. I felt very confident using the system.
10. I needed to learn a lot of things before I could get going with this system.

The answers in the SUS are based on Likert style responses, ranging from "Strongly disagree" to "Strongly agree". The Likert scale is a widely used summated rating that is easy to develop and use. People often enjoy completing this type of scale, and they are likely to give considered answers and be more prepared to participate in this than in a test that they perceive as boring ([13] p. 293).

Brooke characterised the SUS as being a "Quick and Dirty" method of measuring usability. However, Lewis and Sauro state that although SUS may be quick, it is probably not dirty, and they cite studies that show that SUS has been found to be a reliable method of rating usability [14]. SUS has been widely used in the industrial setting, and Lewis and Sauro state that the SUS has stood the test of time, and they encourage practitioners using the SUS to continue to do

so, and show how SUS can be decomposed into Usability and Learnability components, beyond showing the overall SUS score [14]. In a study of questionnaires for assessing the usability of a website, Tullis and Stetson found that the SUS, which was one of the simplest questionnaires studied, was found to yield amongst the most reliable results across sample sizes, and that SUS was the only questionnaire of those studied that addressed all of the aspects of the users' reactions to the website as a whole [15].

In a UTUM test, the users perform the use cases, the test leader observes what happens, and records the time taken to execute the tasks, observes hesitation or divergences from a natural flow use, notes errors, and counts the number of clicks to complete the task. Data is recorded in a form where the test leader can make notes of their observations. The test leader ranks the results of the use case on a scale between 0 - 4, where 4 is the best result. This judgement is based on the experience and knowledge of the test leader. This means that the result is not simply based on the time taken to perform the use case, but also on the flow of events, and events that may have affected the completion of the use case.

After performing each use case, the tester completes a Task Effectiveness Evaluation, a shortened SUS questionnaire [12] concerning the phone in relation to the specific use case performed. This is repeated for each use case. Between use cases, there is time to discuss what happened, and to explain why things happened the way they did. The test leader can discuss things that were noticed during the test, and see whether his or her impressions were correct, and make notes of comments and observations. Even though the test leader in our case does not usually actively probe the tester's understanding of what is being tested, this gives the opportunity to ask follow up questions if anything untoward occurs, and the chance to converse with the tester to glean information about what has occurred during the test.

The final step is an attitudinal metric representing the user's subjective impressions of how easy the phone is to use. This is found through the SUS [12], and it expresses the tester's opinion of the phone as a whole. The statements in the original SUS questionnaire are modified slightly, where the main difference is the replacement of the word "system" with the word "phone", to reflect the fact that a handheld device is being tested, rather than a system. This SUS questionnaire results in a number that expresses a measure of the overall usability of the phone as a whole. In general, SUS is used after the user has had a chance to use the system being evaluated, but before any debriefing or discussion of the test. In UTUM testing, the tester fills in the SUS form together with the test leader, giving an opportunity to discuss issues that arose during the test situation.

The data collected during the test situation is used to calculate a number of metrics, which are then used to make different presentations of the results to different stakeholders. These include the Task Effectiveness Metric, which is determined by looking at each use case and determining how well the telephone supports the user in carrying out each task. It is in the form of a response to the statement "This telephone provides an effective way to complete the given task". It is based on the test leader's judgement of how well the use case was performed, recorded in the test leader's record and the answers to the Task Effectiveness Evaluation. The Task Efficiency Metric is a response to the statement "This telephone is efficient for accomplishing the given task". This is calculated by looking at the distribution of times taken for each user to complete each use case. The distribution of completion times is used to calculate an average value for each device per use case. The User Satisfaction Metric, is calculated as an average score for the answers in the SUS, and is a composite response to the statement "This telephone is easy to use". For more information regarding different ways of presenting these metrics and data, see ([7], Appendix A).

A previous study by Winter et al [6] showed that two different groups of stakeholders existed within UIQ. The first group was designated as Designers represented by e.g. interaction designers and system and interaction architects, representing the shop floor perspective. The second group

was designated as Product Owners, including management, product planning, and marketing, representing the management perspective. These two groups were found to have different needs regarding the presentation of test results. These differences concerned the level of detail included in the presentation, the ease with which the information can be interpreted, and the presence of contextual information included in the presentation. Designers prioritised presentations that gave specific information about the device and its features, whilst Product Owners prioritised presentations that gave more overarching information about the product as a whole, and that were not dependent on including contextual information.

These results, and more information on UTUM in general, are presented in greater detail in [7] (chapter 4 and Appendix A). A video demonstration of the test process (ca. 6 minutes) can be found on YouTube [16].

4. Research Method

The cooperative research and development work that led to the development of UTUM has been based on an action research approach according to the research and method development methodology called Cooperative Method Development (CMD) (see e.g. [17]). CMD is an approach to research that combines qualitative social science fieldwork, with problem-oriented method, technique and process improvement. CMD has as its starting point existing practice in industrial settings, and although it is motivated by an interest in use-oriented design and development of software, it is not specific for these methods, tools and processes.

This particular work is based on a case study [18] and grounded theory [19] approach. A case study is "an empirical enquiry that investigates a contemporary phenomenon within its real-life context, especially when the boundaries between phenomenon and context are not clearly evident" ([18], p. 13). The focus is on a particular case, taking the context into account, involving multiple methods of data collection; data can be both qualitative and quantitative, but qualitative data are almost always collected ([13] p. 178). Case studies have their basis in a desire to understand complex social phenomena, and are useful when "how" or "why" questions are being asked, and where the researcher has little control over events ([18], p. 7). A case study approach allows the retention of characteristics of real life events [18].

The data in this case study has been analysed in a grounded theory approach. Grounded theory (GT) is both a strategy for performing research and a style of analysing the data that arises from the research ([13], p. 191). It is a systematic but flexible research style that gives detailed descriptions for data analysis and generation of theory. It is applicable to a large variety of phenomena and is often interview-based and ([13], p. 90) but other methods such as observation and document analysis can also be used ([13], p. 191). We have not attempted to work according to pure GT practice, and have applied a case study perspective, using ethnography [20] and participatory design [11].

5. Subjects and Context

The data in this study were collected in tests performed by UIQ in Sweden and by a tester from a mobile phone manufacturer in England. The testing was performed in a situation where there are complex relationships between customers, clients, and end-users, and complexities of how and where results were to be used. The phones were a UIQ phone, a "Smart phone" of a competing brand, and a popular consumer phone. The use cases were decided by the English company, and were chosen from their 20 most important use cases for a certain mobile phone. The use cases were:

- UC1. Receive and answer an incoming call
- UC2. Save the incoming call as a new contact - "Joanne"
- UC3. Set an alarm for 8 o'clock tomorrow morning
- UC4. Read an incoming SMS and reply with "I'm fine"

– UC5. Make a phone call to Mårten (0708570XXX)
– UC6. Create a new SMS - "Hi meet at 5" and send to Joanne (0708570XXX)

The test group consisted of 48 testers. The group consisted of 24 testers from Sweden, and 24 testers from England, split into 3 age groups: 17 - 24; 25 - 34; 35+. Each age group consisted of 8 females and 8 males. The size of the group was in order to get results from a wide range of testers to obtain general views, and to enable comparisons between age groups, cultures and genders. Normally, it was not deemed necessary to include so many testers, as small samples have been found to be sufficient to evaluate products. Dumas and Reddish [21] for example, refer to previous studies that indicate in one case that almost half of all major usability problems were found with as few as three participants, and in a second case that a test with four to five participants detected 80% of usability problems, whilst ten participants detected 90% of all problems. This indicates that the inclusion of additional participants is less and less likely to contribute new information. The number of people to include in a test thus depends on how many user groups are needed to satisfy the test goals, the time and money allocated for the test, and the importance of being able to calculate statistical significance.

However, even though this can be seen from the point of view of the participating organisations as a large test, compared to their normal testing needs, where the data collected consisted of more than 10 000 data points, the testing was still found to be a process where results were produced quickly and efficiently. In this case, the intention of using a larger number of testers was to obtain a greater number of tests, to create a baseline for future validation of products, to identify and measure differences or similarities between two countries, and to identify issues with the most common use-cases. Testers were drawn from a database of mobile phone users who have expressed an interest in being testers, and who may or may not have been testers in previous projects.

6. Validity

Regarding internal reliability, the data used in this study have been collected according to a specified testing plan that has been developed over a long period of time, and that has been used and found to be a useful tool in many design and development projects. The risk of participant error in data collection is small, as the test is monitored, and the data is verified by the test leader. The risk of participant bias is also small, as the testers are normal phone users, using a variety of different phones, and they gain no particular benefits from participating in the tests or from rating one device as being better than another. The fact that much of the data has been in the form of self evaluations completed by the testers themselves, and that the testing has been performed by specialized usability experts minimizes the risk of observer error. The risk of observer bias is dealt with by the presence of the two independent test leaders, allowing us to compute inter-observer agreements. The use of multiple methods of data collection, including self assessment, test leader observation and measurement, and the collection of qualitative data, allow us to base our findings on many types and ways of collecting data.

In regard to external validity, the fact that the testing has been performed in two different countries may be seen as a risk, but the two countries, Sweden and England, are culturally relatively close, which should mean that the results are comparable across the national boundaries. The tasks performed in the testing are standard tasks that are common to most types of mobile phones, and should therefore not affect the performance or results of the tests. The users are a cross section of phone users, and the results should thus be generalisable to the general population of phone users.

To ensure the statistical conclusion validity, we use statistical methods that are standard in the field, and use the SPSS software package PASW Statistics 18 for statistical analysis.

7. Data Sets, Possible Correlations and Analysis Steps

The test data has been split into three sets of data. This division is based on the metrics collected in the attitudinal questionnaires and the times recorded by the test leader during testing. These data sets concern satisfaction, effectiveness and efficiency, as called for by ISO 9241-11:1998 [1]. The sets of test data are:

Set 1: SUSuapp - Based on the System Usability Scale (SUS) [12], which consists of 10, 5-scale Likert questions. The evaluation is a user appraisal of satisfaction, based on one evaluation per phone and tester. It is a summary of the use cases performed on the individual phones. It provides us with a total of 144 data points - 48 per phone (48 testers, 3 phones, 1 SUS per phone).

Set 2: TEEuapp - Based on a Task Effectiveness Evaluation (TEE), which consists of 6, 5-scale Likert questions. It is a user appraisal based on one evaluation per phone, use case and tester. The tester fills in this evaluation directly after completing each of the 6 use cases on each of the three phones. It provides us with a total of 864 data points - 144 per use case task (48 testers, 6 use cases, 3 phones).

Set 3: TIMEreal - This is used to represent efficiency, and is the time taken in seconds to complete a use case task. It is a test leader measurement based on one number per phone, use case and tester. The test leader measures the time for the tester to complete each of the use cases on each of the phones. This provides us with 864 data points- 144 per use case task (48 testers, 6 use cases, 3 phones).

As a complement to these data, we also make use of a spreadsheet, the Structured Data Summary (SDS) [22] that is used to record qualitative data based on the progress of the testing. This contains some of the qualitative findings of the testing and the SDS shows issues that have been found, for each tester, and each device, for every use case. Comments made by the testers and observations made by the test leader are stored as comments in the SDS.

The first step in the data analysis is to investigate the strength of the correlations between the metrics for satisfaction, effectiveness and efficiency. The second step is to investigate if the distribution of time taken to perform use cases can provide a reliable indication of problematic use cases, and in which way this should be analyzed and shown. If this is successful, it should be possible to discover use cases that exhibit poor usability by looking at the shape of the distribution curve. The third step is to verify the fact that the distribution of time can be used to illustrate the fact that certain use cases exhibit poor usability. This can be done by comparing with the data recorded in the SDS for these use cases, to see if the test leader has noted problems that users experienced. If this is found to be the case, this indication could be used when testing devices, to identify the areas where test leader resources should be directed, thus allowing a more efficient use of testing resources.

STEP 1: Investigate the correlation between Satisfaction, Effectiveness and Efficiency. For each phone each tester completed a SUS-evaluation (SUSuapp). SUSuapp gives an appraisal score from 0-40. The correlation between the SUSuapp, and TEEuapp might be calculated using Pearson's correlation coefficient, Spearman rank correlation coefficient, or Kendall's rank correlation coefficient (Kendall's Tau). The most reasonable method could be Spearman or Kendall's tau, as these deal with data in the form of ranks or ordering of data, and do not assume normal distribution of the data, on which the Pearson coefficient is based. Spearman is preferred by some, as it is in effect a Pearson coefficient performed on ranks, but Kendall's Tau is usually preferred, as it deals with ties more consistently [13]

The SUSuapp data is the result the 144 Likert appraisals, which could normally be assumed to exhibit a normal distribution. However, in some of the other data distributions, we have observed a positive skew that also suggests that Spearman may be a better choice. Also, the central concentration of the data causes many ties in ranks, which could make Kendall's Tau more appropriate.

The tests that include TIMEreal may be more difficult to deal with. Since the TIMEreal data is continuous, while the other data is of Likert-type, it may be difficult to see any linear relationships. However, the same tests should still be performed. The results of the analysis are found in Table 1.

The analysis shows only weak to moderate correlations between the different factors. This is particularly obvious regarding Kendall's tau, which as previously mentioned is probably the best indicator given the type of data involved here. This supports the findings of Frøkjær [2], who state that all three factors must be measured to gain a complete picture of usability. It contradicts the results of Sauro et al [3], who showed stronger correlations, although even Sauro et al state that it is important to measure all three factors, since each measure contains information not contained in the other measures.

These results do not support our conjecture that there is a sufficiently strong correlation between the 3 factors of usability that simply measuring one of them would give a reliable indication of the usability of a mobile phone.

STEP 2: Investigating if the distribution of time can provide a reliable indication of problematic use cases. We find that TIMEreal data, for time taken to complete a given use case, corresponds well with a Rayleigh distribution (Ray(2*mean)) with a shape parameter that is twice the mean of the data. Data points that end up in the tail fall under a specific degree of probability of belonging to the Ray(2*mean) distribution. This means that the use cases with a "long tail" are those that the testers found to be troublesome (see Fig 1).

Figure 1 illustrates one use case. The right hand diagram is the seconds to complete the use case divided into ten evenly spaced frequency intervals. The diagram to the left is the Ray(2*mean) probability distribution. For example, we see that 2 on the x-axis has a 28% chance belonging to the Rayleigh distribution, and that is where we have a frequency of 70+ data points. 6 on the x-axis has a less than 1% chance of belonging to the distribution and we see that 6 in our data is empty. This would mean that the points in our data set in ranges 7-10 are beyond all probability influenced by something more than the excepted random difference between different testers. Our interpretation is that these are the use cases where "something went wrong". This result suggests that it may be possible to discover use cases where users have problems, by examining the distribution of the time taken to perform the use case.

STEP 3: Verifying the "long tail" method of identifying troublesome use cases. Here we analyse which use cases the testers have experienced as exhibiting poor usability by analysing the distribution of time taken to complete the use case. This is cross tabulated with data from the SDS [22], the spreadsheet containing some of the qualitative findings of the testing. The SDS shows issues that have been found, for each tester, and each device, for every use case. Comments made by the testers and observations made by the test leader are stored as comments in the spreadsheet.

Given the fact that the intention of this work is to find ways that simplify the discovery of problematic use cases, and the fact that the test is designed to be flexible and simple to perform and analyse, we attempted to find some simple heuristic that could help us differentiate between the use cases with high and low levels of problems. We ordered the use cases according to their coefficients of variation, which is the standard deviation divided by the mean time taken to perform the use case. This allows us to standardize the use cases, in order to give a basis for comparison.

This calculation gave us a spread between 0.481 and 1.074. To give a simple cut-off point between Lower and Higher problem use cases, we set a boundary where a coefficient of variation of 0.6 is regarded as High problem, thus dividing the set of use cases into two groups. We also use a simple heuristic to judge an acceptable level of problems when performing a use case. The test leader registers problems observed whilst performing the use case by a letter "y" in the SDS, with an explanatory comment. One tester may have experienced more than one problem, and all of these are noted separately, but we chose to count the number of individuals who

	Kendell's tau_b		Spearman's rho			
	Correlation coefficient	Sig. (2-tailed)	Correlation Coefficient	Sig. (2-tailed)	Pearson Correlation	Sig. (2-tailed)
SUSuapp/ TEEuapp	0.599**	0.000	0.758**	0.000	0.710**	0.000
SUSuapp/ TIMEreal	-0.408**	0.000	-0.573**	0.000	-0.485**	0.000
TIMEreal/ TEEuapp	-0.490**	0.000	-0.663**	0.000	-0.595**	0.000
**Correlation is significant at the 0.01 level (2-tailed)						

Table 1. Correlations between elements of usability

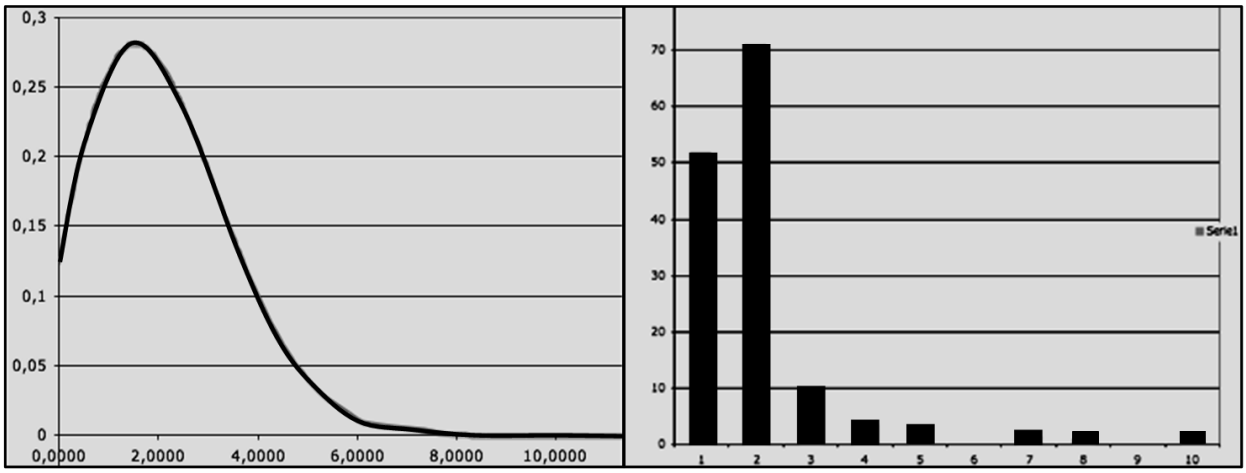


Figure 1. Rayleigh distribution and spread of times to perform use case

had experienced problems, rather than the number of problems. The seriousness of the problems could range from minor to major. However, since the ambition was to find a simple heuristic, we have not performed any qualitative analysis of the severity of the problems, but have simply noted number of users who had problems. We refer to this as No_USERS.

In this case, the cut-off point was set as being less that 33% of the total number of testers. We assume that use cases where more than 33% of users had some kind of problem are High problem, and worthy of further examination.

Table 2 illustrates the cases and their categorization as High or Low problem for Coefficient of variation and No_USERS.

We performed Fisher's exact test on the set of data shown in Table 2. This test can be used in the analysis of contingency tables with a small sample. It is a statistical test that is used to determine if there are non-random associations between two categorical variables. The results of performing Fisher's exact test are shown in Table 3.

Since the values given by Fisher's exact test are below 0.05 they can be regarded as significant, meaning that there is a statistically significant association between Coefficient of variation and No_USERS as we have defined them.

As can be seen in table 3, all of the cases (5) where No_USERS indicated a high rate of problems are discovered by the coefficient of variation being high. On the other hand, a high coefficient of variation also points to just as many cases that do not have a high rate of problems. However, the results still show that a number of use cases (8) can, with high probability, be excluded from the testing process, allowing for more efficient use of testing resources. Simply by calculating

Case No.	1	2	3	4	5	6	7	8		
Coefficient of variation	L	L	L	L	L	L	L	L		
Severity	L	L	L	L	L	L	L	L		
Case No.	9	10	11	12	13	14	15	16	17	18
Coefficient of variation	H	H	H	H	H	H	H	H	H	H
Severity	H	H	H	H	H	L	L	L	L	L

Table 2. Use cases and their categorization as High or Low problem

		No_USERS		
		High Problem	Low Problem	Total
Coefficient of variation	High Problem	5	5	10
	Low Problem	0	8	8
Total		5	13	18
		Exact Sig. (2-sided)	Exact Sig. (1-sided)	
Fisher's Exact Test		0.036	0.029	

Table 3. Coefficient of variation * No_USERS & Fisher's exact test

the coefficient of variation, 8 of 18 cases could be excluded from more expensive testing.

To conclude, the SDS records the fact that the test leader observed that users experienced problems when performing use cases, and there is found to be an association between the use cases where a larger proportion of users experienced problems, and those use cases with a high coefficient of variation. This suggests that it is possible to identify potentially problematic use cases simply by measuring the time taken to perform use cases and analysing the distribution of those times.

This article is based on research that was performed previous to the cessation of activities in UIQ. The limited number of tests that were available to be included for analysis in this study, the fact that the testing as it was performed was not designed as an experiment with this purpose in mind, and that this is a post factum analysis mean that the results must be read with some caution. However, the results we have obtained from this analysis do indicate that this is an interesting area to study more closely.

This means that it may be possible to formulate a "time it and know" formula that can be tested in new trials. This could be used to give a "problem rating" to individual use cases that could categorize the degree of problems that the user experienced. It would allow a simple categorization of use cases without needing the presence of the test leader, simply by measuring the time taken to perform the use cases, in order to identify the areas where test leader resources should be directed, thus allowing a more efficient use of testing resources.

8. Discussion

The aims of this study have been twofold: to examine the correlation between the different aspects of usability (Effectiveness, Efficiency and Satisfaction) to find whether there is one simple measurement that would express usability, and; to discover if it is possible to streamline the discovery of problematic use cases through a statistical analysis of task-completion time,

which would allow scarce testing resources to be concentrated on problematic areas.

The analysis detailed above shows that, for the material collected in our study, the correlations between the factors of usability are not sufficiently strong to allow us to base usability evaluations on the basis of one single metric. This means that it is important that all three factors are measured and analysed, and as discussed previously, the test leader is an important figure in this process. This supports previous work that stresses the importance of measuring all of these aspects. This was stated to be the case even by those researchers who found stronger correlations between the different aspects measured.

However, we do find that it may be possible to discover potentially problematic use cases by analysing the distribution of use case completion times. This would mean that it is possible to collect data which indicate which use cases are most important to concentrate testing resources on. This could be done without without the presence of a test leader. Many companies involved in developing and producing mass-market products already have a large base of testers and customers who participate in different ways in evaluating features and product solutions. By distributing trial versions of software to different user groups, and by using an application in a mobile phone that measures use case completion time, and submits this data to the development company, it should be possible to collect data in a convenient manner. The development company could distribute instructions to users and testers, who could perform use cases based on these instructions, and the telephone itself could transmit data to the company, which could form the basis of the continued analysis and testing process. This data would be especially valuable since it could be based more on the use of the telephone in an actual use context, rather than in a test situation.

From an analysis of the distribution of completion times it is thus possible to gain indications of problem areas that need further attention. However, it is impossible to say, simply by looking at the completion times, what the problem may be. To discover this, and to develop design suggestions and solutions, it is still necessary for the test leader to observe and analyse the performance of the use cases that are indicated as problematic.

Future work would be to test the findings made here, by performing further tests on a greater number of devices, and comparing the results with UTUM testing as it is normally performed. It is also possible to study cases where the statistics indicate that there are problems, and other devices where this was not apparent in the statistics, and compare the results. Further work would also be to test the heuristics used in our analysis, to find if there are more accurate ways of distinguishing between low and high problem use cases.

9. Acknowledgements

This work was partly funded by The Knowledge Foundation in Sweden under a research grant for the software development project “Blekinge – Engineering Software Qualities”, www.bth.se/besq. We would like to thank Associate Professor Claes Jogréus for reading our work and giving advice regarding statistical methods, and all of our partners from UIQ Technology AB, for their help and cooperation.

References

[1] *ISO 9241-11 (1998): Ergonomic Requirements for Office Work with Visual Display Terminals (VDTs) - Part 11: Guidance on Usability*, International Organization for Standardization Std., 1998.

[2] E. Frøkjaer, M. Hertzum, and K. Hornbæk, “Measuring usability: Are effectiveness, efficiency, and satisfaction really correlated?” in *Conference on Human Factors in Computing Systems*, vol. Proceedings of the SIGCHI conference on Human factors in computing systems. The Hague, Netherlands: ACM Press, 2000, pp. 345–352.

[3] J. Sauro and E. Kindlund, “A method to standardize usability metrics into a single score,” in *CHI 2005*, ser. Proceedings of the SIGCHI conference on Human factors in computing systems. Portland, Oregon, USA: ACM Press, 2005, pp. 401–409.

[4] J. Winter, K. Rönkkö, M. Ahlberg, M. Hinely, and M. Hellman, "Developing quality through measuring usability: The UTUM test package," in *ICSE 2007*, ser. 5th Workshop on Software Quality, at ICSE 2007, 2007.

[5] J. Winter, K. Rönkkö, M. Ahlberg, and J. Hotchkiss, "Meeting organisational needs and quality assurance through balancing agile & formal usability testing results," in *CEE-SET 2008*, ser. Preprint of the third IFIP TC2 Central and East European Conference on Software Engineering Techniques, Z. Huzar, J. Nawrocki, and J. Zendulka, Eds., Brno, 2008.

[6] J. Winter and K. Rönkkö, "Satisfying stakeholders' needs - balancing agile and formal usability test results," *e-Informatica Software Engineering Journal*, vol. 3, no. 1, p. 20, 2009.

[7] J. Winter, "Measuring usabiilty - balancing agility and formality," Licentiate Thesis, Blekinge Institute of Technology, 2009.

[8] K. Hornbæk, "Current practice in measuring usability: Challenges to usability studies and research," *International Journal of Human-Computer Studies*, vol. 64, no. 2, pp. 79–102, 2006.

[9] B. Pettichord, "Testers and developers think differently," *STGE magazine*, vol. Vol. 2, no. Jan/Feb 2000 (Issue 1), 2000. [Online]. Available: http://www.io.com/~wazmo/papers/testers_and_developers.pdf

[10] D. Martin, J. Rooksby, M. Rouncefield, and I. Sommerville, ""Good" organisational reasons for "Bad" software testing: An ethnographic study of testing in a small software company," in *ICSE '07*. Minneapolis, MN: IEEE, 2007.

[11] D. Schuler and A. Namioka, *Participatory Design - Principles and Practices*, 1st ed. Hillsdale, New Jersey: Lawrence Erlbaum Associates, 1993.

[12] J. Brooke, "SUS: A quick-and-dirty usability scale," 1986.

[13] C. Robson, *Real World Research*. Oxford, England: Blackwell Publishing, 1993, vol. 2.

[14] J. R. Lewis and J. Sauro, "The factor structure of the system usability scale," in *LNCS 5619*, vol. Proceedings of the human computer interaction international conference (HCII 2009),. Springer Verlag, 2009, pp. 94–103.

[15] T. S. Tullis and J. N. Stetson, "A comparison of questionnaires for assessing website usability," 2004. [Online]. Available: http://home.comcast.net/~tomtullis/publications/UPA2004TullisStetson.pdf

[16] BTH, "UIQ, usability test," Aug. 2008. [Online]. Available: http://www.youtube.com/watch?v=5IjIRlVwgeo

[17] Y. Dittrich, K. Rönkkö, J. Erickson, C. Hansson, and O. Lindeberg, "Co-operative method development: Combining qualitative empirical research with method, technique and process improvement," *Journal of Empirical Software Engineering*, vol. 13, no. 3, pp. 231–260, 2007.

[18] R. K. Yin and S. Robinson, *Case Study Research – Design and Methods*, ser. Applied Social Research Methods Series. Thousand Oaks, Cal.: SAGE publications, 2003, vol. 3.

[19] B. G. Glaser and A. L. Strauss, *The discovery of grounded theory : strategies for qualitative research*. Piscataway, NJ.: Aldine Transaction, 1967.

[20] K. Rönkkö, "Ethnography," in *Encyclopedia of Software Engineering (accepted for publication)*, P. Laplante, Ed. New York: Taylor and Francis Group, 2010.

[21] J. Dumas and J. Redish, *A Practical Guide to Usability Testing*. Exeter, England: Intellect, 1999.

[22] G. Denman, "The structured data summary (SDS)," 2008.

e-Informatica Software Engineering Journal, Volume 5, Issue 1, 2011, pages: 39–49, DOI 10.2478/v10233-011-0029-x
VERSITA

ARINC Specification 653 Based Real-Time Software Engineering

Sławomir Samolej*
*Faculty of Electrical and Computer Engineering, Rzeszow University of Technology
ssamolej@prz.edu.pl

Abstract

This paper reports successive steps of a real-time avionic pitch control application creation. The application structure follows a new real-time systems development profile published in ARINC specification 653. The paper mentions some main ARINC specification 653 features and shows the subsequent application creation levels: control system units distribution, timing requirements definition, application implementation and tests. It describes the author's experience gained during an avionic hard real-time system development and focuses on real-time software engineering details of the application creation.

1. Introduction

Real-time applications gradually evolve form simple one-task embedded programs to large multi-task and distributed systems. Modern large real-time applications are usually based on real-time operating systems (such as VxWorks [1], PikeOS [2], LynksOS [3], WindowsCE [4], and Linux RTAI [5]), or are written in real-time languages (Ada 2005 [6] or Spakr [7]). These applications should be developed according to well defined practical rules expressed e.g. in [8, 9]. Real-time tasks are given priorities according to a predictable policy (such as Rate Monotonic or Deadline Monotonic Policies). Inter-task communication protocols prevent the system from deadlocks and unpredictable delays during execution (by Priority Inheritance or, if possible, by Priority Ceiling Resource Access Protocol application). The data exchange between distributed applications should be predictable. Naturally, such rules can be applied in modern real-time operating systems APIs or in Ada language, but less advanced developers often encounter problems, especially in case of complex software.

To make the real-time system development less cumbersome, several groups of experts have developed some practically applicable real-time design patterns. Among the set of well defined code-generation oriented design patterns the POSIX standard [10], HOOD [11] and HRT-HOOD [12] methods and Ada Ravenscar Profile [13] can be indicated. They define good real-time system programming techniques and are partly supported by some automatic real-time software code generators. Subsequently, the real-time design patterns have been integrated with dedicated software toolkits such as Matlab-Simulink [14], SCADE SUITE [15] or IBM Rational Rose RealTime [16]. These tools allow to design graphically the real-time software, giving as the output a real-time system structure (IBM Rational Rose), both structure and selected algorithms implementation (SCADE SUITE), or complete real-time application code and environment generated for a specific hardware (Matlab-Simulink). Moreover, SCADE SUITE enables to develop a complete hard real-time system source code that conforms to international safety standards DO-178B (up to level A for

Military and Aerospace Industries), IEC 61508 (at SIL 3 by TÜV for Heavy Equipment, and Energy), EN 50128 (at SIL 3/4 by TÜV for Rail Transportation), and IEC 60880 (for Nuclear Energy).

It is also worth noting that international standardization organizations have recently published some new standards or specifications regarding modern real-time systems development. The ARINC (AERONAUTICAL RADIO, INC) specification 653 [17] or the SAE (Society of Automotive Engineers) Standard AS5506 [18] are examples of such documents. To the author's opinion the new documents bring a new quality in real-time systems development. They provide a new abstraction layer in the real-time software design process which makes possible to create complete large distributed real-time systems. This paper deals with the ARINC specification 653 (ARINC 653) based hard real-time systems development.

The ARINC 653 was developed by aviation experts to provide "the baseline environment for application software used within Integrated Modular Avionics (IMA) and traditional ARINC 700-series avionics". It is closely connected with the Integrated Modular Avionics (IMA) concept [19, 20, 21]. Primary objective is to define a general purpose APEX (APplication/EXecutive) interface between Operating System (O/S) of the avionics computer and application software.The specification includes interface requirements between application software and O/S and list of services which allow the application software to control scheduling, communication, and status of internal processing elements. Over a period of 6 years from the specification announcement:

- the ARINC 653 based software has been implemented in A380, A400M and B787 airliners,
- at least three commercial real-time operating systems (WxWorks 653, PikeOS, LynksOS-178 RTOS) have been updated to offer the APEX,
- four European-founded and focused on the ARINC 653 – based software development research projects (PAMELA, NEVADA, VICTORIA and SCARLETT) have been created.

This paper describes some details concerning real-time programming rules included in ARINC 653. A Pitch Control Application created within SCARLETT [22] project by Research Group from Rzeszów University of Technology (RGRUT) illustrates the ARINC 653 based development process. The following sections are organized as follows. At first, the ARINC 653 is introduced. Secondly, the Pitch Control Application development is presented. The final part describes the future RGRUT's research and implementation plans.

2. ARINC specification 653

Airborne real-time systems have been evolving from the so-called "federated" structure to Integrated Module Avionics (IMA) [19, 20, 21]. The IMA concept has been introduced in European funded research projects, PAMELA, NEVADA and VICTORIA. The result of the projects was the first generation of IMA (IMA1G), currently on-board of A380, A400M and B787 aircraft. Following the IMA concept, modern on-board avionic subsystems (software applications) are grouped in a limited set of standard microprocessor units. The units and other electronic devices communicate via standard network interface – Avionics Full Duplex Switched Ethernet (AFDX) [23, 24]. The group of federated applications executed until now on separate microprocessor units (and communicating by means of ARINC standard 429 based devices [25], for example) must become a set of real-time processes executed on one hardware module. This module will be managed by a dedicated real-time operating system. Provided that the operating system offers a standard API and fulfills safety requirements, such solution significantly broadens portability of avionic applications and allows to develop and certify hardware and software independently. Current implementation of IMA covers limited range of aircraft functions but shows some significant benefits, i.e. aircraft weight reduction and lower maintenance costs.

2.1. Partitions

The IMA assumes that a set of time-critical and safety-critical real-time applications (avionics units) may be executed on one microprocessor module. To cope with this level of criticality, new real-time operating system architecture has been suggested. ARINC 653 [17] defines generic structure of the system. Figure 1 shows the logical structure of RTOS suggested in it.

The key concept introduced in the specification is the partition. It creates a kind of container for an application and guarantees that execution of the application is both spatially and temporally isolated. The partitions are divided into two categories, application partition and system partition. The application partitions execute avionics applications. They exchange data with the environment by means of specific interface – APEX (APplication/EXecutive). The system partitions are optional and their main role is to provide services not available in APEX, such as device drivers or fault management.

2.2. Hardware-Software Module Architecture

The ARINC 653 also includes some recommendations on microprocessor module architecture for the dedicated real-time operating system. General diagram of the architecture is presented in figure 2. Each module includes one or more microprocessors. The hardware structure may require some modification of core operating system but not the APEX interface. All processes that belong to one application partition (real-time tasks) must be executed on one microprocessor. It is forbidden to allocate them to different microprocessors within the module or split them between modules. The application program should be portable between processors within the module and between modules without any modifications of the interface to operating system core. Processes that belong to one partition may be executed concurrently. A separate partition-level scheduling algorithm is responsible for this. Inter-application (partition) communication is based on the concept of ports and channels. The applications do not have the information at which partition the receiver of data is executed. They send and receive data via ports. The ports are virtually connected by channels defined at separate level of system development.

Temporal isolation of each partition has been defined as follows. A major time frame, activated periodically, is defined for each module. Each partition receives one or more time partition windows to be executed within this major time frame. Generally, time partition windows constitute a static cyclic executive [9]. Real-time tasks executed within the partition can be scheduled locally according to priority-based policy. The order of the partition windows is defined in a separate configuration record of the system.

Health Monitor (HM) is an important element of the module. HM is an operating system component that monitors hardware, operating system and application faults and failures. Its main task is to isolate faults and prevent failure propagation. For example, the HM can restart a partition when detects application fault.

By assumption, the applications (or partitions) may be developed by different providers. Therefore an integrator of the IMA system development process is necessary. This person collects data regarding resources, timing constraints, communication ports and exceptions defined in each partition. The collected data are transferred into configuration records. The configuration record for each module is an XML document interpreted during compilation and consolidation of software.

2.3. APEX Interface

APEX (APplication/EXecutive) interface definition is the main part of ARINC 653. The APEX specifies how to create platform-independent software that fulfills ARINC 653 requirements. Main components of the interface are:

- partition management,
- process management,
- time management,
- memory management,
- interpartition communication,
- intrapartition communication,

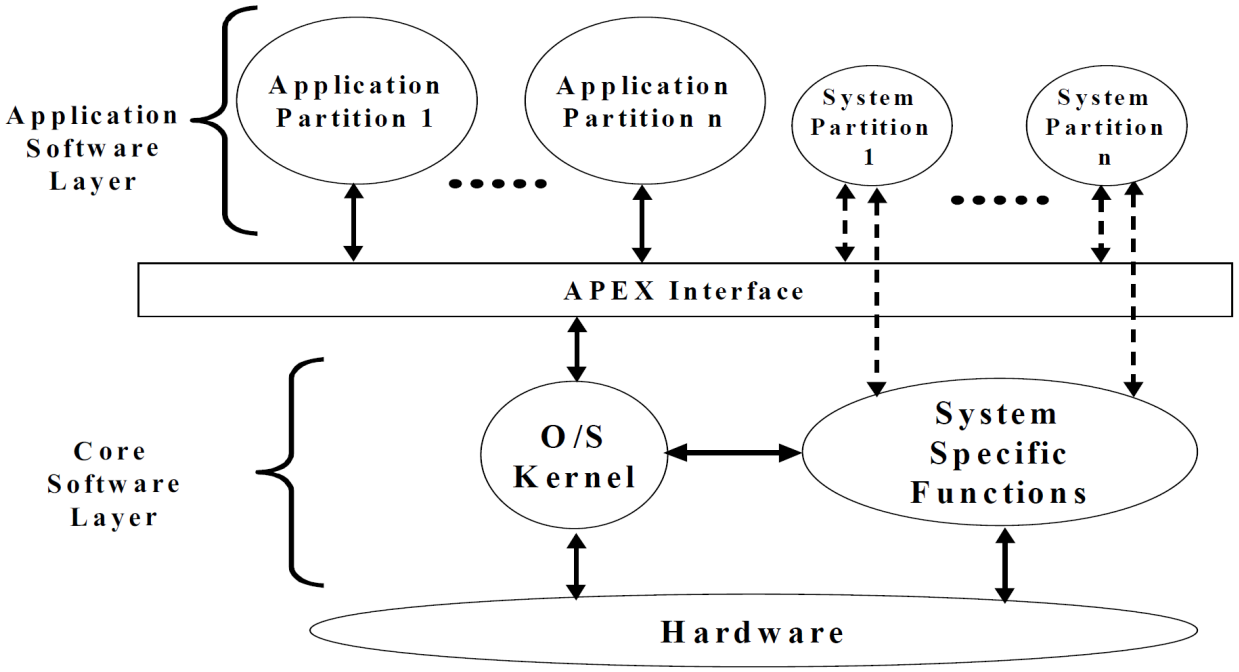


Figure 1. Logical real-time operating system structure created according to ARINC specification 653 ([17], pp. 4).

– health monitoring.

The APEX interface provides separate set of functions enabling the user to determine actual partition mode and change it. The application may start the partition after creation of all application components. It is also able to obtain the current partition execution status. Interpartition health monitoring procedures can stop or restart the partition.

The application may be constructed as a set of (soft or hard) real-time processes, scheduled according to priorities. APEX process management services can:

– create process and collect process status or ID,
– start, stop, suspend or resume process,
– prevent from process preemption,
– change the process priority.

The APEX manages both aperiodic and periodic processes. Periodic processes are activated regularly. Additionally, a separate parameter called "time capacity" is attached to each of them. It defines time frame (deadline) within which single instance of task must finish computations. When a process is started the deadline is set to current time plus time capacity. The operating system periodically checks whether the process completes processing within the allotted time. Each process has a priority. During any rescheduling event the O/S always selects for the execution the highest priority process in "ready" state.

There are no memory management services in APEX because partitions and associated memory spaces are defined during system configuration. The ARINC 653 assumes, for safety reasons, that the whole memory is statically allocated to partitions and processes before the partition or application starts. It is expected that memory space is checked either at build time or before running the first application.

Interpartition (inter-application) communication is based on queuing port and sampling port communication units. The queuing port provides interpartition message queue, whereas the sampling port shares variables between the ports. During system integration, the ports are connected by means of channels defined in system configuration tables. The ports communicate with other partitions or device drivers within the

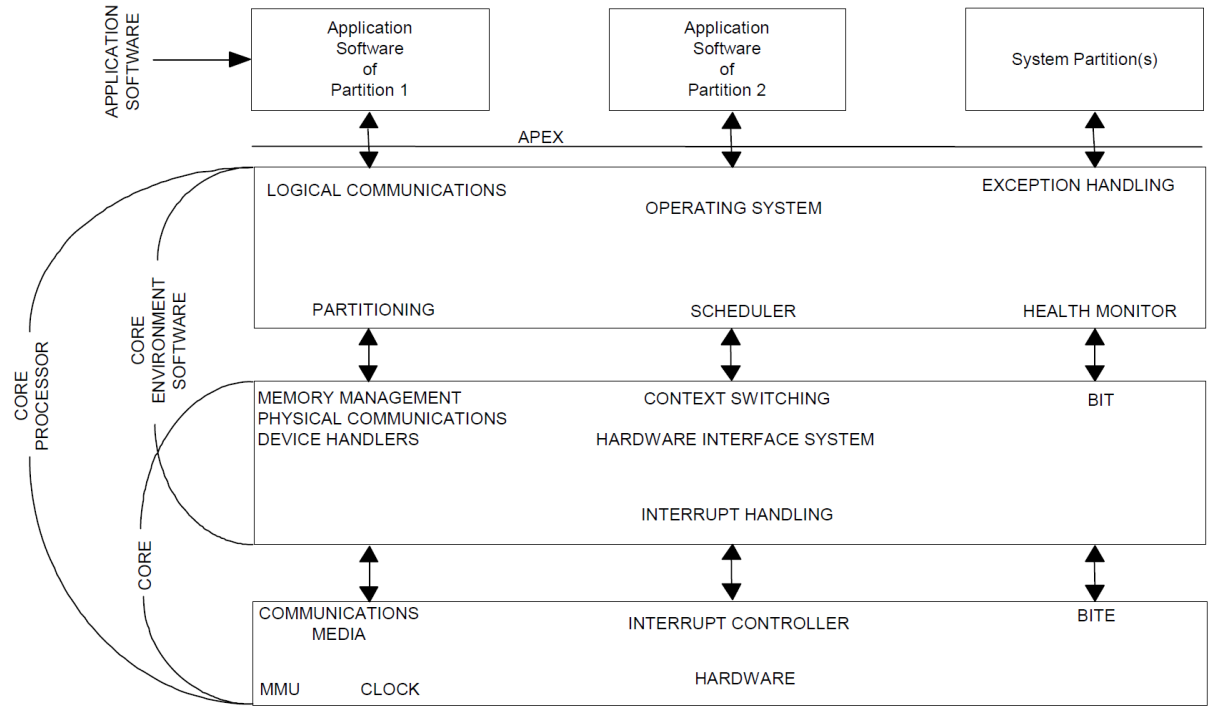


Figure 2. Hardware-software architecture of typical module according to ARINC 653 specification ([17], pp. 11).

module, or exchange data between modules (by means of AFDX network interfaces).

The synchronization of processes belonging to one partition may be achieved by appropriate application of counting semaphores and events. The inter-process communication within the partition (intrapartition communication) is implemented by means of APEX buffers (shared message queues) and APEX blackboards (shared variables). It is possible to define a time-out within which process waits for the data, if not available immediately. The process may be blocked for the specified time only.

The ARINC 653 Health Monitor is an advanced exception handing engine. Three error levels are defined:

- Process Level which affects one or more processes in the partition,
- Partition Level with only one partition affected,
- Module Level which affects all partitions within the module.

Both Partition Level and Module Level errors are handled by a set of procedures installed by the system integrator. The Process Level errors can be handled by the programmer. Separate sporadic task called "error handler" can be registered for each of the partitions. When the Health Monitor detects an error at the process level it calls the error handler. The handler recognizes the error and, depending on the error, can:

- log it,
- stop or restart the failed process,
- stop or restart the entire partition,
- invoke the registered error handler process for the specific error code.

To the author's knowledge four real-time operating systems meet ARINC 653 requirements, i.e. Wind River VxWorks 653 [1], Sysgo PikeOS [2], LynuxWorks LynxOS-178 RTOS, and LynuxWorks LynxOS-SE RTOS [3].

3. ARINC 653 based Pitch Control System Development

This section describes an ARINC 653 based sample avionics subsystem development. The objec-

tive is to create a distributed hard real-time application to control a pitch angle of typical airliner (Pitch Control Application – PCA). Subsequent development steps are: control system definition, control procedures allocation to hardware units, timing requirements assessment, application structure design, system programming, testing. Preliminary version of the PCA has been proposed in [26]. Papers [27] and [28] report on stages of PCA development and some extensions to detect control system malfunctions.

3.1. Control System Definition

The system controls two actuators (brushless motors) connected to a load (elevator). Each actuator is controlled by separate cascade of controllers shown in figure 3. The single actuator control system includes internal current control loop, velocity control loop (PID2, PID4), and position control loop (PID1, PID3). The Flight Control Algorithm (FCA) is a supervisory module that generates position demand signal for both actuator control subsystems. It collects signals from the pilot, aircraft, and actuators. The reference signal is a force or shift of control side-stick moved by the pilot. The FCA module corrects the reference signal using the actual aircraft pitch angle. The PCA includes also Error Estimator that collects some control signals and estimates quality of control system during runtime. It also produces separate output for system operator. The entire Pitch Control Application has been modeled in Matlab-Simulink [14] software toolkit. All control procedures and settings have been designed following control engineering rules.

3.2. Distribution of Control Procedures

The Pitch Control Application has been developed according to a set of restrictions formulated by the system integrator. One of the restrictions requires selected control procedures to be allocated on different hardware modules. Figure 4 illustrates the desired PCA control modules allocation. Hardware Module 1 (HM1) includes the FCA, Error Estimator and position controllers (PID1 and PID3, compare fig. 3). Hardware Modules 2 and 3 (HM2 and HM3) include velocity controllers (PID2 and PID4, compare fig. 3). The next restriction is that the FCA, Error Estimator and all the PID controllers should be software modules whereas the current controllers must be included into motor drives hardware. The hardware modules are connected via AFDX network. The network structure has also been imposed by the system integrator.

According to the requirements the hardware-software environment follows the IMA philosophy. Therefore the FCA and Error Estimator control blocks belong to one ARINC 653 based application partition, as two separate real-time tasks. All PID controllers are separate real-time tasks each of which belongs to separate application partition. The partition structure physically and temporally isolates the main control application subsystems. It also enables reallocation of PID control procedures between hardware modules, what is another application requirement.

3.3. Timing Requirements

The Pitch Control Application has been developed to fulfil the ARINC 653 real-time parameters shown in figure 5 and table 1. The time capacity (deadlines) and task periods have been chosen taking into account both actuator dynamics and computing power of hardware units. The partition including the FCA and Error Estimator periodic real-time task acquires 6 [ms] time slot and 20 [ms] deadline. It is executed in two 3-millisecond time frames (fig. 5). The partitions with PID1 and PID3 control procedures are activated every 5 [ms] and executed within 1 [ms] time frame. Similarly, PID2 and PID4 real-time tasks are activated every 5 [ms], but their deadlines are extended to 2[ms] since Hardware Modules 2 and 3 provide lower computing power than Hardware Module 1. The major time frames in figure 5 include some “System” slots. These slots may be used by other software modules loaded on hardware. The HARD attribute attached to each of the real-time tasks instructs the Health Monitor (built into the operating system) that if any task misses its deadline, the core operat-

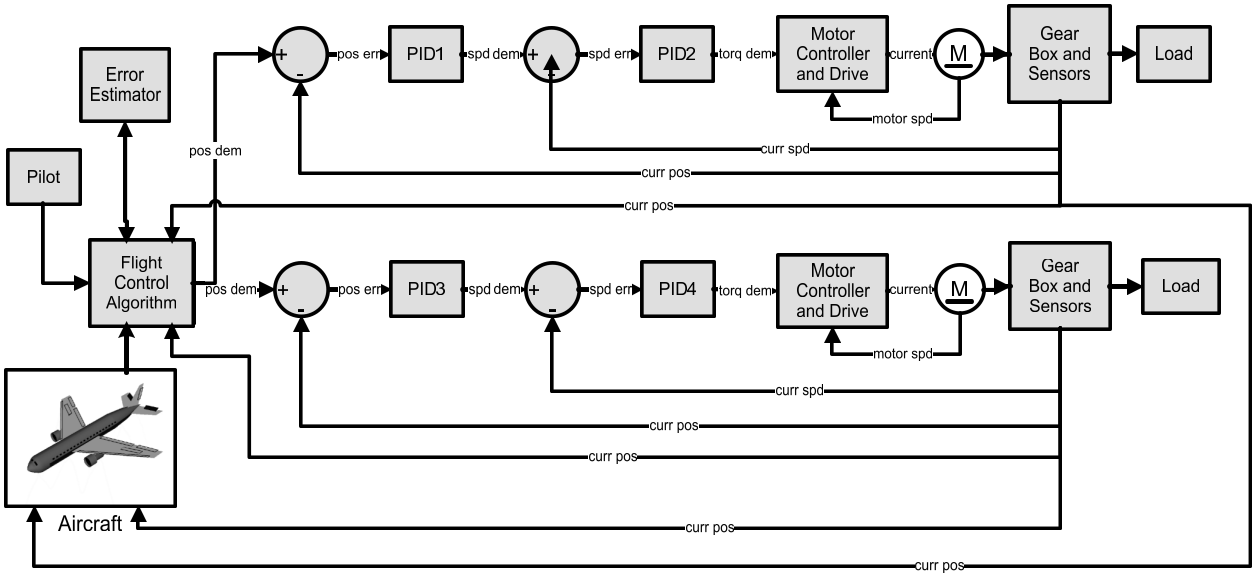


Figure 3. Pitch control system architecture

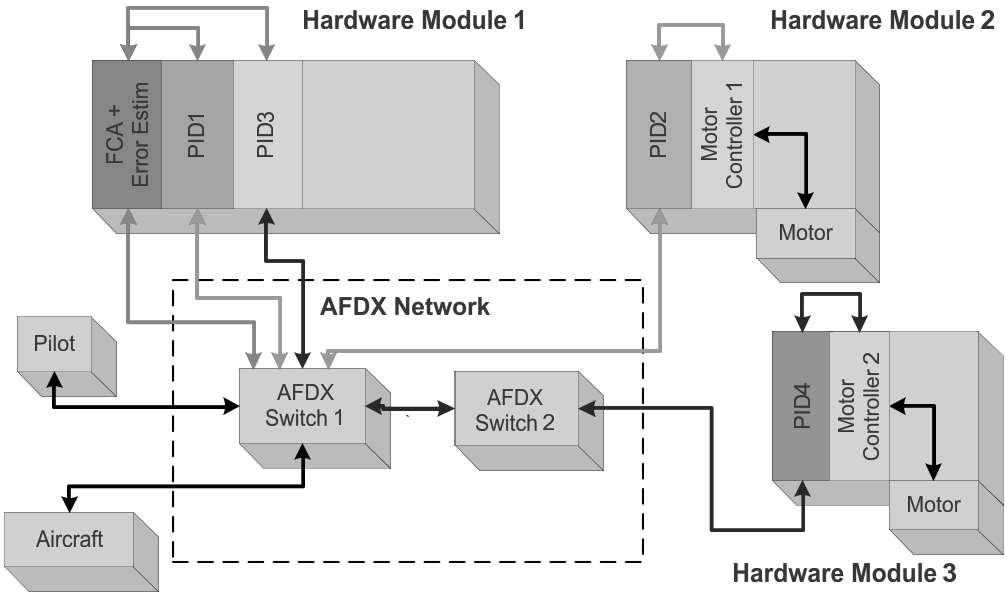


Figure 4. Allocation of Pitch Control System procedures on the Hardware Units

ing system must be informed. In consequence, this forces the operating system to take appropriate action. The Health Monitor procedures may even reload the whole partition that misses timing constraints. It has been assumed that the maximum communication delay in the AFDX network should not exceed 7 [ms].

All of the algorithms applied in the PCA are either controllers or simplified numerical procedures which solve some differential equations. During the development the worst case computing time for each of the algorithm has been evaluated. Experimental checks have proved that the algorithms meet their timing constraints.

As mentioned before, the PCA timing restrictions have been provided by control engineers who specified the system. P2 and P3 partitions acquire 1 [ms] time frames for computations and are activated every 5 [ms]. This guarantees sufficient frequency (200Hz) of PID algorithm repetition, so sufficient quality of control. P1 partition is 4 times "slower" than others without adverse effect

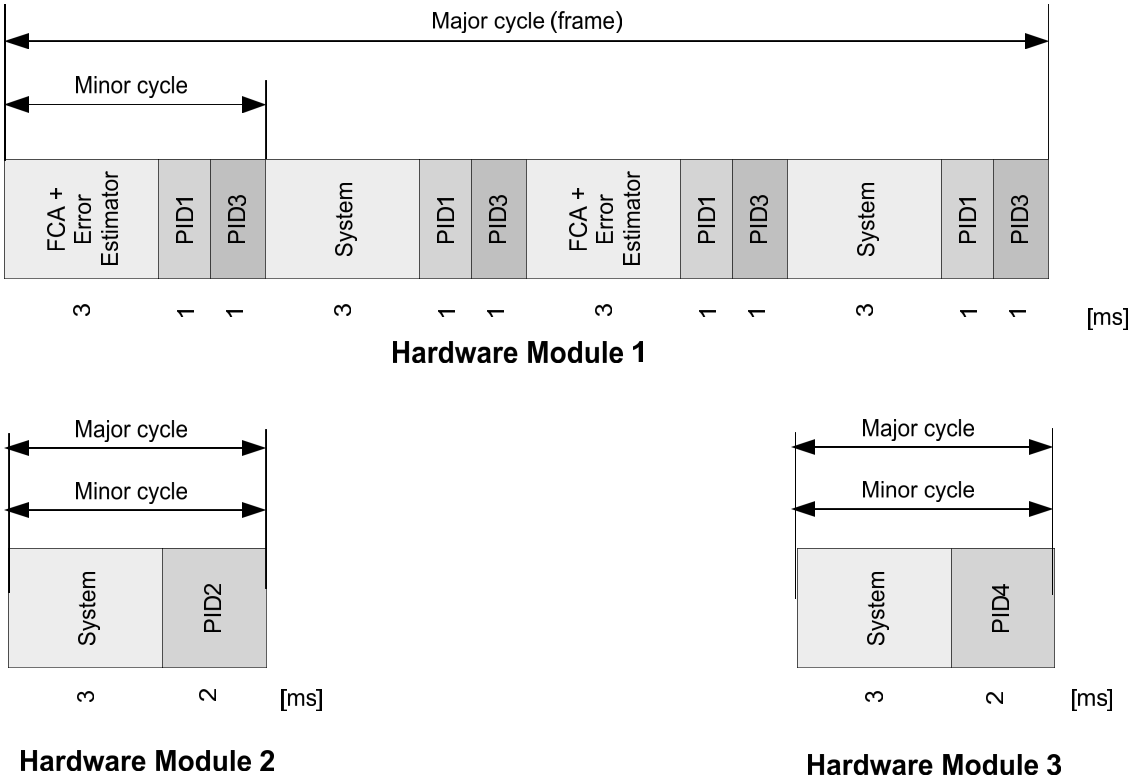


Figure 5. PCA timing requirements

Table 1. PCA real-time tasks parameters

Control Procedure	Stack Size	Base Priority	Period	Time Capacity	Deadline
FCA	4096	18	20	20	HARD
Error Estimator	4096	17	20	20	HARD
PID1	4096	20	5	1	HARD
PID3	4096	20	5	1	HARD
PID2	4096	20	5	2	HARD
PID4	4096	20	5	2	HARD

on quality of control. This preserves some computational time for other applications installed on the same hardware module.

3.4. Application Structure

The Pith Control Application structure is shown in figure 6. Apart from control procedures allocation strategy of figure 4, figure 6 shows the partitions and the intra- and interpartition communication structure. The first (P1) partition loaded into the Hardware Module 1 includes two real-time tasks, the Flight Control Algorithm (FCA) and Error Estimator. The FCA task collects signals from Pilot, Aircraft and actuator modules and produces the desired pitch angle signals for position controllers (PID1, PID3). The Error Estimator monitors both communication channels and quality of control during runtime. The algorithms applied in the Error Estimator have been presented in [27, 28]. The second (P2) partition includes the first position controller algorithm (PID1), running as separate real-time task. Identically, the third (P3) partition includes the second position control algorithm (PID3). The fourth (System Partition) collects all signals exchanged between hardware modules and transfers them to the AFDX network. The Hardware modules 2 and 3 have the same hardware-software structure.

They include one system partition and one application partition. The application partitions (P4 and P5) involve single real-time tasks with speed control PID algorithm. The system partitions provide inter-hardware module communication.

For the intrapartition communication, ARINC 653 blackboards have been applied, whereas the interpartition communication is based on ARINC 653 sampling ports and channels. Blackboards and sampling ports seem most suitable communication units for the system, since they are in fact shared and protected data regions. They always provide the latest acquired data. It is possible to set their properties in such a manner that they do not block the real-time tasks, even if they are empty. It is assumed that some of data packages produced by "fast" control blocks may be lost due to the "slow" ones. From real-time software engineering point of view all communication mechanisms applied in the PCA are shared variables and monitors [9]. This solves the mutual exclusion problem. The shared variables are accessed (at the operating system level) according to Priority Inheritance Protocol [8, 9]. One can check that the PCA communication structure does not include deadlocks.

Due to "external" partition scheduling and simple application structure, the priorities of local task are used mostly for the precedence constraints definition. The tasks priorities (defined within the partitions – compare tab. 1) reflect the order of the computations, the tasks should follow. This approach is essential especially in P1 partition. It is expected that the FCA real-time task is terminated before the Error Estimator task begins.

4. System Programming and Testing

The PCA has been finally implemented in C language for VxWorks 653 [29, 30, 31] and PikeOS [32, 33, 34] operating systems, with APEX interface applied for PCA structure generation. From the programmer's point of view, each partition defined during the application development is a separate program. The program consists of a set of real-time tasks. The tasks exchange data by means of APEX blackboards or send and receive data via sampling ports. Timing constraints are attached to the tasks. Each task involves main function which collects data from the input communication objects (blackboards or sampling ports), calls appropriate control block algorithm, sends computed data to output objects, and finally suspends execution. The function is periodically activated by core operating system.

The PCA is a distributed real-time control application, so implementation tests have involved three main areas, i.e. communication, timing constraints and control algorithms. Firstly, the communication structure of the application has been assessed and all channels and data structures checked. Secondly, Worst Case Execution Time (WCET) for each of the real-time task has been measured and the meeting of timing constraints both at the process and the partition level evaluated. The measured time has been compared with partition time slots defined at the beginning. Thirdly, the control procedures applied in the application have been tested, both as separate function blocks and as complete set of cooperating software modules.

To the author's experience, good practice for the ARINC 653 programmer is to prepare working document that explains:

- ports introduced in the application,
- channels' description (how to connect the ports with other ports),
- time budget (partition window) allocated to the partition,
- time period within which the application (partition) should run again,
- memory requirements.

Some extra records with internal structure of the application will also help. Typically the application developer should provide:

- parameters of real-time tasks (IDs, stacks, priorities, deadlines, periods, criticalness <HARD/SOFT>),
- internal communication structure (blackboards and buffers, their IDs, capacities, tasks attached),

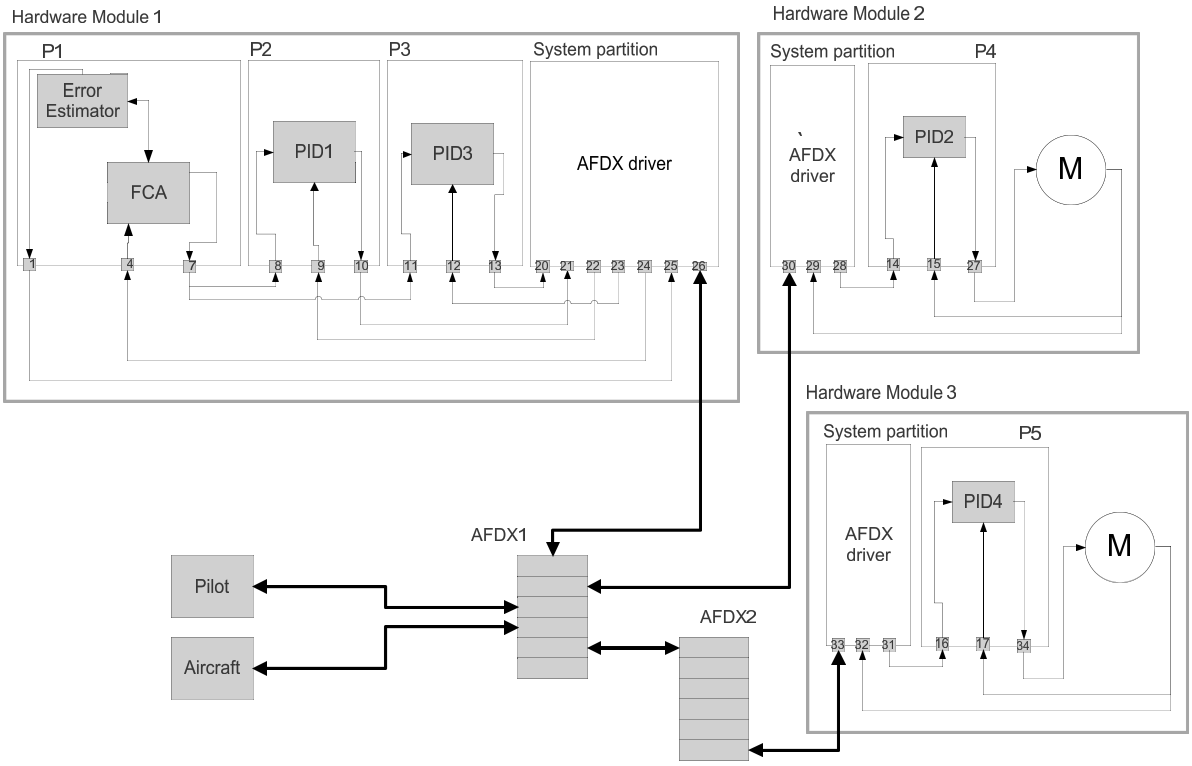


Figure 6. PCA structure

– communication timeouts (attached to APEX function calls to read data from blackboards or buffers),
– internal error handler procedure actions.

According to IMA philosophy the prepared applications (partitions) in a form of binary files equipped with such documents are sent to the system integrator.

5. Conclusions and Future Research

The paper reports emerging trends and design patterns in real-time systems development. It is focused on ARINC 653, where a new standard for real-time systems design is introduced. Successive steps of a typical ARINC 653 based application (Pitch Control Application PCA) development are described. The application has been designed as contribution of Rzeszów University of Technology to European SCARLETT [22] project. In the current state of the PCA development, the application is being integrated with other hardware and software modules delivered by SCARLETT partners. It is expected that the final PCA application test will reveal whether the current hardware-software environment conforming with ARINC 653 is able to execute some distributed hard real-time applications successfully.

6. Acknowledgments

Research reported in the paper is funded by SCARLETT 7th European Framework Project, Grant Agreement No. FP7-AAT-2007-RTD-1-211439. Some of hardware components used in the research published within this paper were financed by the European Union Operational Program – Development of Eastern Poland, Project No. POPW.01.03.00-18-012/09.

References

[1] Wind River WWW Site. [Online]. Available: http://www.windriver.com/
[2] SYSGO WWW Site. [Online]. Available: http://www.sysgo.com/

[3] Lynux Works WWW Site. [Online]. Available: http://www.lynuxworks.com/
[4] WindowsCE WWW Site. [Online]. Available: http://www.microsoft.com/
[5] Linux RTAI WWW Site. [Online]. Available: https://www.rtai.org/
[6] J. Barnes, *Programming in Ada 2005*. Addison-Wesley, 2006.
[7] ——, *High Integrity Software, The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.
[8] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, 1997.
[9] A. Burns and A. Wellings, *Real-Time Systems and Programming Languages*. Pearson Education Limited, 2001.
[10] "IEEE Std 1003.1, 2004 Edition," IEEE, Tech. Rep., 2004.
[11] J.-P. Rosen, *HOOD an Industial Approach for Software Design*. Elsevier, 1997.
[12] A. Burns and A. Wellings, *HRT-HOOD: A structured design Method for hard Real-Time Ada Systems*. Elsevier, 1995.
[13] A. Burns, B. Dobbing, and T. Vardanega, "Guide for the use of the ada ravenscar profile in high integrity systems," University of York, Technical Report YCS-2003-348, Jan 2003.
[14] Matlab-Simulink WWW Site. [Online]. Available: http://www.mathworks.com/
[15] Scade Suite WWW Site. [Online]. Available: http://www.esterel-technologies.com/products/scade-suite/
[16] IBM Rational Rose RealTime WWW Site. [Online]. Available: http://www-01.ibm.com/software/awdtools/developer/technical/
[17] *Avionics Application Software Standard Interface Part 1-2, ARINC Specification 653P1-2*, 2005.
[18] *SAE AS5506 Standard: Architecture Analysis and Design Language (AADL)*, 2006.
[19] P. Bieber, E. Noulard, C. Pagetti, T. Planche, and F. Vialard, "Preliminary design of future reconfigurable ima platforms," in *ACM SIGBED Review - Special Issue on the 2nd International Workshop on Adaptive and Reconfigurable Embedded Systems*. ACM, Oct 2009.
[20] P. Parkinson and L. Kinnan, "Safety-critical software development for integrated modular avionics," Wind River, Wind River White Paper, 2007.
[21] J. W. Ramsey, "Integrated Modular Avionics: Less is More Approaches to IMA will save weight, improve reliability of A380 and B787 avionics," *Avionics Magazine*, 2007. [Online]. Available: http://www.aviationtoday.com/av/categories/commercial/8420.html
[22] SCARLETT Project WWW Site. [Online]. Available: http://www.scarlettproject.eu
[23] "AFDX: The Next Generation Interconnect for Avionics Subsystems," Avionics Magazine Tech. Report, Tech. Rep., 2008.
[24] *Aircraft Data Network Part 7 - Avionics Full Duplex Switched Ethernet (AFDX) Network, ARINC Specification 664 P7*, 2005.
[25] *ARINC 429: Mark 33 Digital Information Transfer Systems (DITS)*, 1996.
[26] S. Samolej, A. Tomczyk, J. Pieniążek, G. Kopecki, T.Rogalski, and T. Rolka, "VxWorks 653 based Pitch Control System Prototype," in *Development Methods and Applications of Real-Time Systems*, L. Trybus and S. Samolej, Eds. WKL, 2010, ch. Chapter 36, pp. 411–420, (in Polish).
[27] T. Rogalski, S. Samolej, and A. Tomczyk, "ARINC 653 Based Time-Critical Application for European SCARLETT Project," Aug 2011, accepted for presentation at the AIAA Guidance, Navigation, and Control Conference, 8-11 Aug 2011 Portland, Oregon, USA.
[28] S. Samolej, A. Tomczyk, and T. Rogalski, "Fault Detection in a ARINC 653 and ARINC 644 Pitch Control Prototype System," Sep 2011, accepted for publication in: Development, Analysis and Implementation of Real-Time Systems, L. Trybus and S. Samolej eds., WKL, 2011, (in Polish).
[29] *VxWorks 653 Configuration and Build Guide 2.2*, Wind River, 2007.
[30] *VxWorks 653 Configuration and Build Reference, 2.2*, Wind River, 2007.
[31] *VxWorks 653 Progmer's Guide 2.2*, Wind River, 2007.
[32] *PikeOS Fundamentals*, Sysgo AG, 2009.
[33] *PikeOS Tutorials*, Sysgo AG, 2009.
[34] *PikeOS Personality Manual: APEX*, Sysgo AG, 2009.

e-Informatica Software Engineering Journal, Volume 5, Issue 1, 2011, pages: 51–63, DOI 10.2478/v10233-011-0030-4
VERSITA

Experience with instantiating an automated testing process in the context of incremental and evolutionary software development

Janusz Górski*, Michał Witkowicz*
*Departament of Software Engineering, Gdansk University of Technology
jango@pg.gda.pl, miwi@eti.pg.gda.pl

Abstract

The purpose of this article is to present experiences from testing a complex AJAX-based Internet-system which is under development for more than five years. The development process follows incremental and evolutionary lifecycle model and the system is delivered in subsequent releases. Delivering a new release involves both, the new tests (related to the new and/or modified functionalities) and the regression tests (after their possible refactoring). The article positions the testing process within the context of change management and describes the applied testing environment. Details related to documenting the test cases are given. The problem of automation of tests is discussed in more detail and a gradual transition from manual to automated tests is described. Experimental data related to the invested effort and the benefits resulting from tests automation are given. Plans for further development of the described approach are also presented.

1. Introduction

Testing can serve both, verification and validation purposes. It can generate considerable costs (according to [1] it is not unlikely that testing consumes 40% or more of the total development effort) and therefore defining testing objectives and strategy belong to the key decisions to be made in a software development project. This includes not only the answer to how much we want to spend on testing but also what, when and how to test to maximize benefits while keeping the costs in reasonable limits. It is well known that testing cannot prove absolute correctness of a program [2] and can consume (practically) unlimited resources. On the other hand, testing can prove (and does it in practice) that the program includes faults. Therefore, an important criterion to be used in practice is to drive the testing process in a way that increases the likelihood of fault detection. In practical situations it means that we are interested in such test cases selection criteria which result in tests that have the highest potential for detecting significant faults within given time and budgetary constraints. Significant faults are these which have highly negative impact on the program behavior in its target environment. For instance, a fault which is never or very rarely activated within a given operational context of a program is less significant than a fault which is activated in (almost) every usage scenario, unless the former fault is considered to be 'catastrophic' in which case it should be eliminated even for a very high price.

Current tendency is that the number of application programs developed for use in Internet increases rapidly. The complexity of such applications grows. As they are often used to support businesses, commerce and other services, the expectations related to their dependability increase. This requires employment of more sophisticated assurance processes. To maintain their usabil-

ity, such programs have to follow the evolution of their requirements and target environments. Therefore, in addition to the common corrective maintenance practices, they have to be subjected to the perfective and adaptive maintenance processes [3]. A program, instead of being considered as the final result of its development process, is better understood as an object which appears in time in subsequent incarnations, following an evolutionary process. An important question then is not only how to ensure the expected dependability of the program, but in addition how to maintain the assumed level of guarantees throughout the program evolution. Without such change in the attitude we can end up with a product which quickly disintegrates in time and in consequence, the users lose their interest in it.

Changes of a program undermine the confidence in its reliability. Even a very small change (for instance one single bit) can result in a dramatic loss of reliability (for instance, the program stops to work entirely). A widely adopted solution is to have *regression testing* in place, meaning that the program is subjected to a designated set of test cases each time it has been modified. With careful selection of these test cases, a positive result of all tests supports the claim that the reliability of the modified program remains unchanged. Depending on the scope of the changes involved, the regression test suite is being modified to reflect the program evolution.

In present business environments, time is considered a very valuable asset and shortening time-to-market of new products and services is among the highest priorities. This is reflected in the growing popularity of incremental delivery of software products, where the product is delivered as a series of subsequent increments, each increment representing some added value for the users. From the software assurance viewpoint we are facing here the same situation as in software evolution. The product is growing and each subsequent increment is expected not to decrease its dependability comparing to its predecessor.

Combining incremental development with software evolution is the common situation which calls for strengthening software assurance practices and building them into the software process from the very beginning. This has been reflected in the agile approaches to software development where testing is brought to the front of the process by integrating it with the requirements specification (specifying requirements by test cases) and running tests as soon as the first increments are coded.

The purpose of this paper is to present a case study involving instantiation of an automated testing process during development of a substantial Internet application TCT [4, 5] based on AJAX technologies. The application follows the evolutionary and incremental development process model. The data presented in this paper were collected during the period of more than five years of development and evolution of TCT.

The application is based on AJAX technologies [6, 7]. AJAX radically changes the protocol of interaction between the Internet browser and the server. The granularity of exchanged messages drops down from a full page to a page element and such elements are exchanged asynchronously in a way which is highly transparent to the user. The result is that the user has a feeling of working interactively without delays caused by page reloading.

Moving to the AJAX technologies has significant impact on program testing. Numerous techniques, e.g. these described in [8, 9] become non-applicable or can be applied only partially. In Table 1 we assess some of them following [8].

Table 1 demonstrates that only selected techniques (in particular, these based on the so called test recording and replaying) and some tools of the xUnit type (e.g. squishWeb or Selenium) are suitable for testing AJAX-based software. Other techniques are not applicable or require significant modifications.

The TCT application considered in this paper is following the incremental and evolutionary development process. The objective is to deliver subsequent increments while maintaining a satisfactory level of reliability and keeping the development effort in reasonable limits. To achieve this we had to invest in automation of tests which proven to be particularly beneficial.

The paper first sets the scene, explaining the object of testing, its architecture and the pro-

Table 1. Web testing techniques applied to AJAX-based applications [8]

Testing	Adequate	Problems	Tools
Model-based	no	Web models extracted are partial; existing Web crawlers are not able to download site pages	research
Mutation-based	no	Mutant operators are never being applied to client Web code; the application of mutant operators is difficult	not-existing
Code Coverage	partially	It is difficult to cover dynamic events and DOM changes; coverage tools managing a mix of languages are not available	*Javascript*: Coverage validator *Java*: Cobertura, Emma, Clover, etc. *Languages mix*: not available
Session-based	no	It is impossible to reconstruct the state of the Web pages using only log-files	research
Capture/Replay and xUnit	yes	Javascript, asynchronous HTTP requests and DOM analysis are not always supported	*not ok*: Maxq, HTTPUnit, InforMatric, etc. *partially ok*: Badboy, HTMLUnit, etc. *ok*: squishWeb, Selenium, etc.

cess of its development. Then we describe the testing process and explain how it developed in time. Next, we present subsequent steps towards automation of the testing process together with the collected data which characterize the performance of the automated tests. We also highlight the main factors which, in our opinion, had the most significant influence on these results. In conclusion we also present the plans for further improvement of the process as the application grows and its usage context becomes richer.

2. Object of testing

The object under test was an Internet application called TCT, being a part of the Trust-IT framework [4, 5]. The objective of Trust-IT is to provide methodological and tool support for representation, maintenance and assessment of evidence-based arguments. From the user's perspective, TCT implements a set of services. They cover eighteen groups of key system functionalities accessible by a user (listed in Table 6). Multiple instantiations of the services are deployed for different groups of users (presently we run some sixteen such instantiations). Each instantiation supports different 'projects' which are used by different users. The users work independently and in parallel, accessing the services by an Internet browser.

Trust-IT together with the TCT tool has been developed in a series of projects supported by EU 5^{th} and 6^{th} Framework Programmes[1]. The tool was already applied to analyze safety, security and privacy of IT systems and services from different domains, including healthcare, automobile and business. Presently TCT is being used to support processes of achieving and assessing conformance to norms and standards, in particular in the medical and business domains (more information can be found in [10]). Further application domains are being investigated, including monitoring of critical infrastructures.

[1] 5^{th} FR UE Project DRIVE, IST-12040, 6^{th} FR UE Integrated Project PIPS IST-507019, 6^{th} FR UE STREP Project ANGEL IST-033506

2.1. Application architecture

The architecture of TCT follows the *rich client-server model* which is illustrated in Figure 1 [11]. The model includes three layers: database server PostgreSQL [12], application server JBoss [13] and a client written in JavaScript [14] in accordance with AJAX (Asynchronous JavaScript and XML) [6]. The client is automatically uploaded to the browser of the end user.

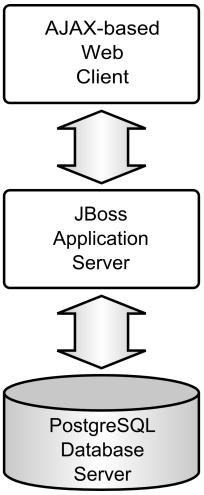


Figure 1. The architecture of TCT

The lowest layer (the database) implements the business logic as a set of stored procedures. The intermediate layer is based on J2EE [15] and links the database with the client. Communication between these layers is based on web services and SOAP (Simple Object Access Protocol) [16].

Each layer is a complex program: presently the database stores some 135 procedures, the intermediate layer and the client layer have 141 classes and 139 classes correspondingly. The two higher layers include additional structure (internal layers) to provide for better understandability and maintainability.

2.2. Increments and evolution

Following earlier prototypes, the development of the present TCT tool was initiated in December 2005. At the beginning, the objectives and the scope of the required functionalities were identified and the delivery of the functionalities was planned as a series of increments. The intention was to follow the *incremental development model* [17, 18] by producing deliverables in subsequent iterations, where each iteration involves requirements gathering and analysis, design, implementation and testing. In effect, each iteration results in the release of an executable subset of the final product, which grows incrementally from iteration to iteration to become the final system.

However, it has been soon realized that following the incremental model in a strict sense is not relevant. As TCT was being developed in the context of on-going research and the research objectives (and consequently the results) were shaped and scoped by the results of the experiments and the feedback received from the participants of these experiments, the requirements for TCT were changing, following a learning curve. Therefore we had to switch to a more complex, *incremental and evolutionary development model*, which can be characterized as follows [19]: it implies that the requirements, plans, estimates, and solutions evolve and are being redefined over the course of the iterations, rather than being fully defined and 'frozen' during the major up-front specification step before the development iterations begin; evolutionary model is consistent with the pattern of unpredictable discovery and change in new product development.

From the testing perspective the incremental and evolutionary development process poses important challenges: (1) to maintain reliability of the subsequent increments it was necessary to have the regression testing in place from the very beginning, and (2) to follow the evolution of the application, the set of regression tests could not be treated as monotonic (i.e. extended by the new test cases reflecting the current increment while preserving the already used test cases); instead it had to evolve following the changes introduced to the already existing functionalities.

So far, TCT has been delivered in eight releases: four of them were related to the *major changes* and the remaining four were related to the *localized changes* of the application. To reflect this scope of change, the former are also called *main releases* and the latter *intermediate releases*. The difference between the two is based on the assessment of the impact of the introduced changes

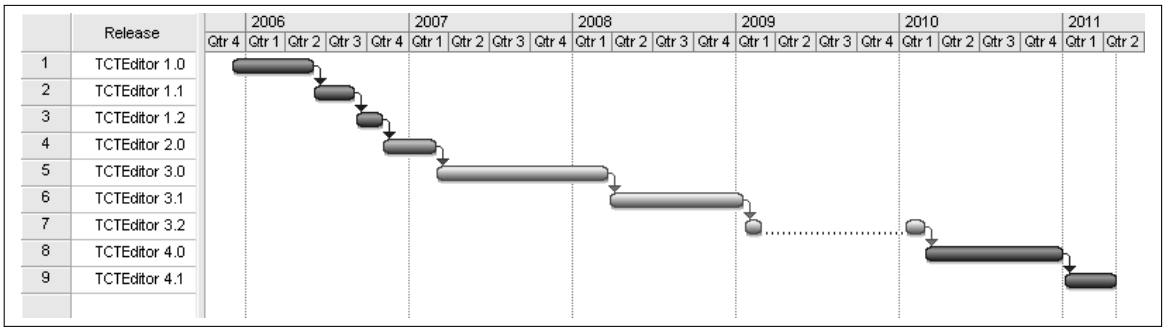


Figure 2. The history of development of TCT

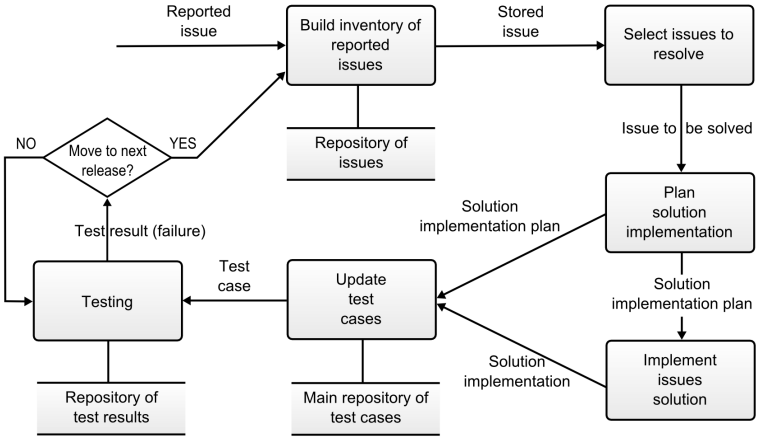


Figure 3. Testing in the context of change management process

and the resulting need for the thoroughness of regression tests. The history of TCT releases is illustrated in Figure 2. The releases are numbered by two digits, where the first one denotes the number of the main release, whereas the second denotes the intermediate release related to the main one.

Throughout the whole evolution, the architecture of the application remains stable. The changes were mainly related to functionality (adding new functions, changing existing functions, removing obsolete functions) and to the applied technologies.

3. Testing in the context of change management

The testing process is embedded in the broader process of change management. The model of the change management process is shown in Figure 3.

The process implements mechanisms for reporting the issues related to the application and for maintaining a related repository of issues. The repository is based on the MantisBT platform [20]. The repository is being periodically reviewed, which results in selecting the issues to be resolved. The selected issues are assigned priorities and then are grouped together in packages, each package containing the issues which can be solved by a common maintenance action. Not all issues require changes in software, for instance some are related to the way a user interacts with the system and can be solved by improving user's documentation and training. The issues which call for software changes are dealt with in the next steps of the process. First, the necessary changes are subjected to planning and then an explicit decision about implementing the plan is bring made, after assessing the resources needed for such implementation and the resulting impact. The plan for the change and the change implementation process provide input to the step of updating the test cases. The new and updated test cases are then stored in

the tests repository forming the new suite of the tests to be performed. The repository of test cases is implemented using Subversion (SVN) [21]. After implementing the change, the tests are being applied.

For a given release of the application, depending on if it is related to a main release or to a intermediate release, the scope of related testing differs. In the former case, the tests include both, the tests covering the new functionality and the full set of regression tests. In the latter case, the tests cover the new/changed functionality and only a subset of the regression tests is included. Limiting the scope of regression tests for a localized change is based on the assumption that the impact of the change is limited and is unlikely to affect the whole scope of the functions.

The testing process is illustrated in Figure 4. It starts with building the *current repository of test cases* which is a subset of the tests maintained in the main repository of test cases. The selection criteria depend on if we are testing a main or an intermediate release of the system. In the former case the current repository is simply a copy of the main repository. In the latter case it is a proper subset of the main repository. Then, the selected test cases are run and the results are collected in the *repository of test results*. The results of the tests are then analyzed and assessed. In case of failed tests, there are two possibilities: (1) inserting new issues to the *repository of issues* (for further processing) or (2) updating the current repository of tests. The former possibility takes place if removing the cause of the test failure involves a significant change; then introducing this change is left to the next releases of the system. The latter possibility is in place if (due to a negative test result) an immediate and localized change is introduced to the software which affects the corresponding tests kept in the current repository. The decision on which alternative is chosen is taken by the tests manager and involves consultation with the representatives of key groups of the users.

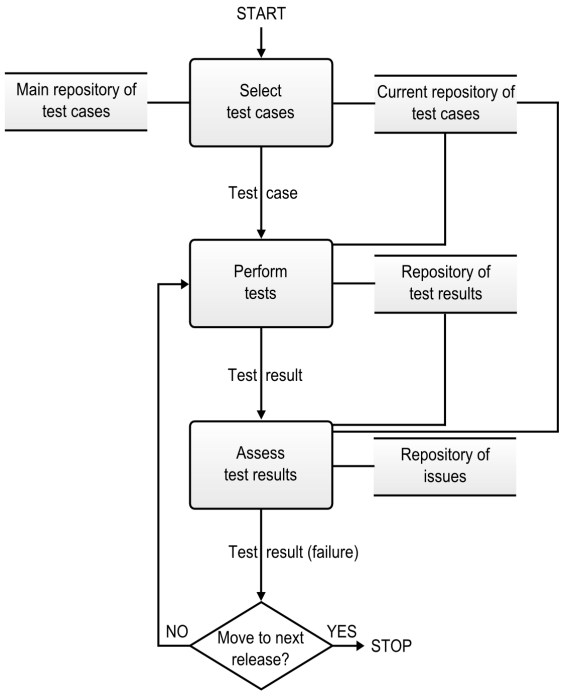


Figure 4. Two phase testing process

After assessing the test results, the decision is being made concerning the continuation of the testing process. If all the issues detected during the previous phase are deposited to the repository of issues, the testing process stops and the next release is delivered to the users. If however, some immediate changes were introduced to the software, the tests kept in the current repository are executed again.

3.1. Specification of test cases

Each test case is represented in accordance with a predefined structure. It includes the following elements:

- **Identifier** - a unique name of the test case. The name suggests the tested functionality;
- **Author** - identification of the person responsible for this test case;
- **Feature** - brief, intuitive description of the tested feature;
- **Scenario** - description of the related testing scenario including:
  - summary of the scenario,
  - description of the initialization phase,

  - the list of actions necessary to complete the test case,
  - description of the closing phase of the test case;
- **Success criterion** - specification of how to assess that the test case completed successfully.

An example test case specification is given in Table 2. The objective of this test case is to check if authentication of the users assigned to different roles works correctly.

3.2. Manual execution of test cases

At the beginning, all test cases were executed manually. The testers were following the test scenarios specified for each test case. If a test case does not pass the success criterion, the tester immediately reports this as an issue to be solved. The issue specification includes details of the sequence of actions which led to the failure.

If the currently reported issue is similar to an issue already reported, then the existing issue description is being updated instead of inserting the new one. Assessment of this 'similarity' was left to the tester. And this appeared to be a weak point in issues reporting. Because analyzing the existing descriptions was boring, the testers often did not go deeply into the details and just concluded that the issue has been already reported and its description did not need any update. The result was that sometimes an important information was not included in the issue description, which adversely impacted fault diagnostics and correction.

4. Automation of test case execution

Manual execution of the process illustrated in Figure 4 consumed significant resources for both, complete regressions tests (full suite of regression tests performed for each main release of the system) and partial regression tests (for the releases related to the localized software changes). This had negative impact on the delivery time of subsequent releases and slowed down the development process.

To deal with the above problems we decided to invest in automation of test case execution. A separate testing system was developed based on the TestNG library [22] and the server and the library offered by Selenium Remote Control [23]. The testing system maintains the TCT system metaphor which is being updated in parallel to the subsequent releases of TCT. The metaphor is used to activate these functions of TCT which are callable from an Internet browser. In consequence, these functions are being activated automatically.

Each result of an automated test case is structured in the XML format (Extensible Markup Language) [24] and inserted to a file. Such reports are periodically reviewed and the detected issues are inserted to the issues repository.

Figure 5 illustrates an example fragment of the code implementing the testing scenario shown in Table 2. Method 1 attempts to log a user in, assuming the role 'viewer'. Method 3 implements the logging sequence. It includes a check if the required element has been visualized. Method 2 finalizes the test scenario and logs the user out.

5. Experiences

So far, there were eight system releases (see Figure 2) including four main releases and four intermediate ones. Testing of the three first releases (two main and one intermediate) was performed by a team of five testers. The process was manual and consumed significant resources. Testing of the second main release was performed by a team of four testers and with partial automation of test cases (automated testing did not play an important role yet and was just experimented with). The third main release was tested with 30% test cases already automated. The testing process involved two testers.

The results achieved were so encouraging that the effort in test case automation was increased which resulted in that for the fourth main release (Release 4.0 in Figure 2) the number of automated test cases reached 95%. This resulted in radical decrease of the effort to execute test

Table 2. An example of test case specification

Identifier	SystemFunctions_Login
Author	Michał Witkowicz
Feature	Access to system functions for different user roles - system login
Scenario	*Summary:* Make sure that all possible roles have accounts in the system (viewer, developer, assessor and admin). Then login to the system as a user assuming different roles. *Initialization:* Check if login screen is properly displayed (find DOM element with id="tct_login_data_form"). If not, the user is likely to be logged in; log the user out (SystemFunctions_Logout). Then, if the login screen is properly displayed - do nothing. *Actions:* 1. Input user login and password. 2. Press "log in" button. 3. Check if the root node named "Projects" of the projects tree is displayed (max. waiting time = 10 sec.). 4. Log out the user. 5. Check if the login screen is correctly displayed (find DOM element with id="tct_login_data_form"; max. waiting time = 10 sec.). 6. Repeat the steps 1-5 for every possible user role: admin, developer, assessor and viewer. *Finalization:* Repeat the Initialization phase again.
Success criterion	Users assigned to all different roles are able to log in to the system

Method 1. test

```
@Test (groups = {"TCT","viewer"})
public void test() throws InterruptedException {
    cmdContainer.loginPage.logIn(viewerUser.getLogin(), viewerUser.getPasswd());
}
```

Method 2. tearDownTest

```
@AfterMethod
public void tearDownTest() throws InterruptedException {
    cmdContainer.mainMenuBar.logout();
}
```

Method 3. logIn

```
public void logIn(String userName, String password)
throws InterruptedException, SeleniumException {
    selenium.type("login_username", userName);
    selenium.type("login_password", password);
    selenium.click("button_logIn");
    cmdContainer.waitForElementPresent("dom=document.
getElementById('projects_root').parentNode.childNodes[3].firstChild",
cmdContainer.loadPageDelay);
}
```

Figure 5. An example test case code

cases and in significant shortening of the delivery time for this release.

The progress in test cases automation is illustrated in Figure 6.

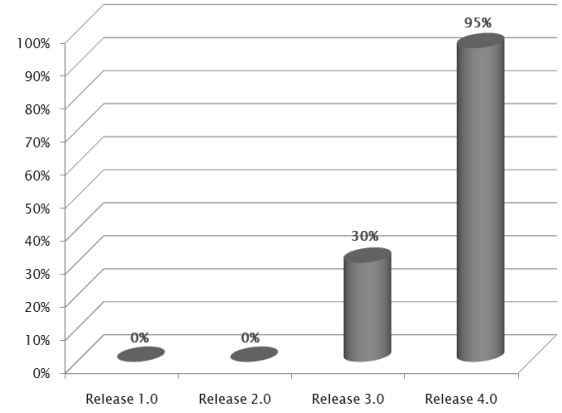


Figure 6. The progress of the test cases automation

Automation of test cases is not free, however. In our experience, one man-day resulted in automation of approximately five test cases (to automate 67 test cases we needed 14 man-days). However, comparing to the effort needed for manual execution of test cases during testing of subsequent system releases, this effort was acceptable (more details are given in Table 4 and 5).

When it comes to the cost of maintaining automated test cases, it depends on a scope of changes in the application in the next release. Obviously, the maintenance of test cases was more expensive while preparing a main release of the system. For example, for the fourth main release it has been observed that one man-day resulted in approximately three updated test cases and total 16 man-days were needed for tests maintenance (see Table 4 and 5). For intermediate releases, usually only few test cases needed to be updated, and the total effort was significantly less.

The total numbers of test cases for the subsequent main releases of the system are given in Figure 7.

Tables 3, 4 and 5 summarize the effort needed for testing the four main releases and illustrate the gain (in terms of effort) resulting from test cases automation.

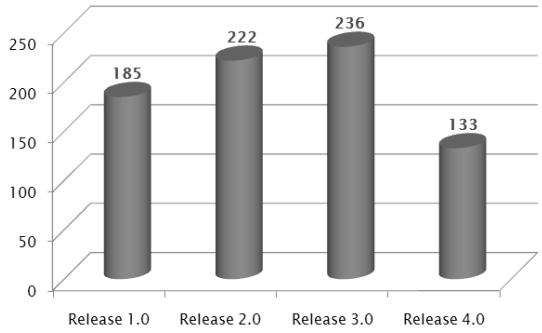


Figure 7. The numbers of all test cases for the main releases of the system

Test cases were following system development and evolution. This was not only because new test cases were being introduced but also because some test cases became obsolete and some other were merged together or modified. As the result, periodic refactoring [25] of test cases was necessary, to maintain test cases integrity and understandability. In particular, such refactoring was performed before testing the release 4.0 which resulted in reducing the number of test cases from 236 (in release 3.0) to 133.

The coverage by test cases of the key system functionalities of the release 4.0 is shown in Table 6.

Tables 7 and 8 illustrate the numbers of issues reported during testing of subsequent main releases (Table 7) and during exploitation of these releases (Table 8).

During the exploitation phase, the reported problems were classified in accordance with the different categories of the maintenance objectives [26]: adaptive, corrective, preventive and perfective. On the other hand, all issues detected during testing were classified as corrective.

Figure 8 compares numbers of different issues detected during exploitation and Figure 9 compares the issues of corrective category between testing and exploitation phases.

From figures 8 and 9 we can see that for release 2.0, testing detected some 30% of corrective issues whereas the remaining 70% were detected in the exploitation phase. For the release 3.0 this proportion looks better and may indicate the positive influence of tests automation. For the release 4.0 this proportion looks much better: automated tests detected nearly 100% of corrective issues. However, it should be noted that the

Table 3. Number of test cases and number of testers involved in testing of main releases

Release #	# of test cases	# of manual test cases	# of automated test cases	# of testers
1.0	185	185	0	5
2.0	222	222	0	4
3.0	236	169	67	2
4.0	133	6	127	2

Table 4. Distribution of testing effort for main releases

Release #	Application design	Specification/ Implementation	Maintenance	Execution	Total effort
1.0	-	5 man-days	0	15 man-days	20 man-days
2.0	-	1 man-day	6 man-days	18 man-days	25 man-days
3.0	10 man-days	15 man-days	5 man-days	20 man-days	50 man-days
4.0	0	14 man-days	16 man-days	3 man-days	33 man-days

Table 5. Distribution of testing effort for automated tests

Release #	Application design	Specification/ Implementation	Maintenance	Execution	Total effort
3.0	10 man-days	14 man-days	0	2 man-days	26 man-days
4.0	0	14 man-days	16 man-days	2 man-days	32 man-days

Table 6. Test case coverage

Functionality	Number of test cases
Copy, cut and paste functions	27
Accessibility of basic system functions	14
Tree of projects and versions	11
Management window for administrators	10
Tree of trust cases	9
Access rights management	9
Appraisal mechanism	8
Management on links	7
Tree element expand and collapse	6
Refresh function	5
Import and export	5
Login and logout	5
Management of user settings	4
Behaviour of tree icons	4
Management of repositories	4
Reference nodes	3
Report generator	1
Traversal tool	1

Table 7. The number of reported issues during testing of main releases

Release #	Issues
1.0	15
2.0	56
3.0	20
4.0	63

Table 8. The number of reported issues during exploitation of main releases

Release #	Adaptive	Corrective	Preventive	Perfective	Sum of issues
1.0	1	47	0	38	86
2.0	4	173	18	75	270
3.0	5	39	1	32	77
4.0	0	1	0	2	3

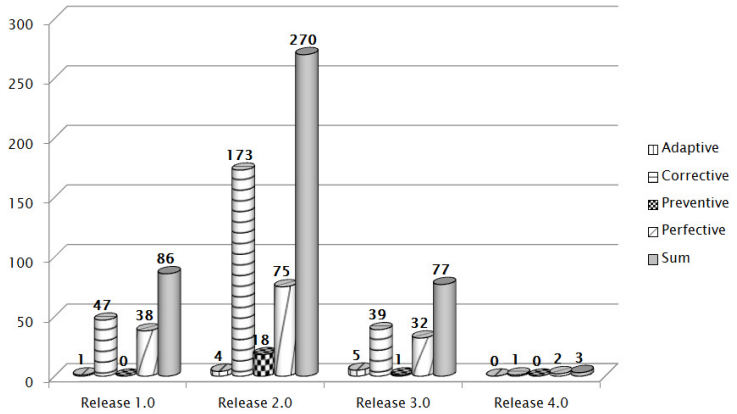


Figure 8. Issues distribution during maintenance of the TCT system

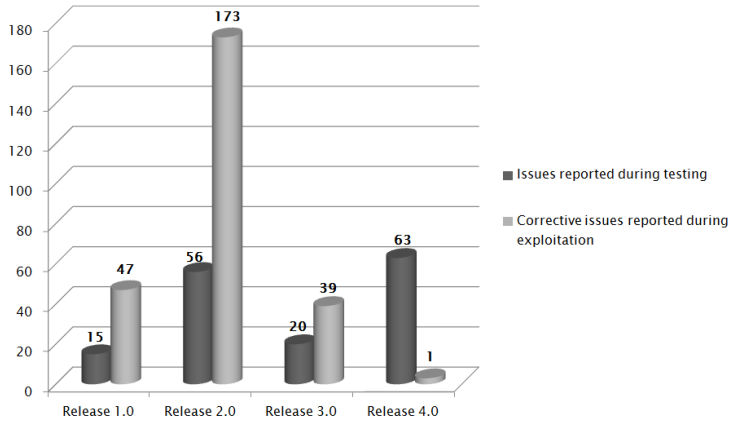


Figure 9. Corrective issues detected during testing and exploitation

exploitation period of release 4.0 amounts for just three months (whereas for release 2.0–12 months and for release 3.0–10 months). To make these data more comparable we calculated the 'issues density' metrics (dividing the number of detected issues by the system exploitation period). The result is shown in Table 9.

The above numbers should not be over interpreted, however. To assess the influence of the testing process on the reliability of the TCT system we would have to take into account other factors for which we have no quantifiable data yet. This involves for instance the influence of the 'size' system change or the operational profile during system exploitation. Nevertheless, at least one relationship can be clearly observed: as the number of different users of the subsequent

Table 9. Corrective issues density for the last three main releases

Release #	Issues per month
2.0	14,41
3.0	3,9
4.0	0,33

system releases increases, in the light of the data presented in Table 9, the claim of increasing reliability of the system gains more credibility.

The presented results suggest that there is still a considerable opportunity to improve the effectiveness of the testing process. However, we cannot ignore the fact that the defects detected during testing are usually the 'big' ones (i.e. such that disable or significantly disturb the usage of the system), whereas the defects detected during exploitation are usually 'small' and rarely prevent the users from using the system. However, it is also worth to note that the difference between 'big' and 'small' defect is context dependent and what is 'small' from one user's perspective (not noticeable at all or slightly disturbing) can be considered 'big' from the perspective of another user with different operational profile. Although we did not yet collected enough data to differentiate between the different impact of the defects, we are well aware of this problem and intend to exploit it while planning for the next steps of test process improvement.

6. Conclusions and plans for the future

The decision about automation of test cases, in particular with respect to the regression tests, has been positively verified in practice as it resulted in considerable reduction of the tests execution effort and contributed to removing subjectivity from execution and interpretation of tests and their results. Despite relatively high cost of the implementation and maintenance of automated tests, the total testing effort decreased and a significant gain in system reliability has been observed.

In our particular case we could observe that perfective maintenance had a considerable share in system changes. This is because the system is being developed in the context of a research process which generates new ideas and discovers new ways of system usage. It can be expected that for systems developed in a business context the influence of this type of changes would be less significant.

In the near future we plan for delivering the next (intermediate) release of the system. This involves the ongoing effort of designing new test cases (checking the new functionalities) and refactoring the existing test cases. Nevertheless, the goal of having 100% regression tests fully automated seems to be not realistic due to the present limitations of the Selenium platform [23].

To exploit the potential of improving the effectiveness of the testing process (illustrated in Figure 9) and to take into account the differences between 'big' and 'small' faults we plan for introducing to our testing process the risk based selection of test cases [27, 28, 29, 30]. This will take into account different usage scenarios and the consequences related to system failure within these scenarios. This information will be then traced back to the system functionalities and reflected in 'weighting' of the related test cases. These weights will be taken into account while planning for the test coverage of critical functions.

The next main release of the system will involve a significant change of technology, especially in relation to the client layer (see Figure 1). To deal with this change we also plan for extending the scope of unit testing of system components. For better control of tests coverage, mutation testing [31] is also considered.

References

[1] I. Sommerville, *Software Engineering*, eighth edition ed. England: Pearson Education, 2007.

[2] R. Patton, *Software Testing*, second edition ed. United States of America: Sams Publishing, 2006.

[3] P. Grubb and A. A. Takang, *Software Maintenance Concepts and Practice*. Singapore: World Scientific Printers, 2003.

[4] J. Górski, "Trust-it - a framework for trust cases," in *Proc. Workshop on Assurance Cases for Security - The Metrics Challenge*. Edinburgh, UK: The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks DSN, 2007, pp. 204–209.

[5] J. Górski *et al.*, "Trust-it research project," information Assurance Group, Gdansk University of Technology (18.10.2011). [Online]. Available: http://iag.pg.gda.pl/iag/?s=research&p=trust_cases

[6] D. Crane, E. Pascarello, and D. James, *Ajax in Action*. Manning Publications Co., 2006.

[7] J. Eichorn, *Understanding AJAX: Using JavaScript to Create Rich Internet Applications*. Prentice Hall, 2006.

[8] A. Marchetto, P. Tonella, and F. Ricca, "Testing techniques applied to ajax web applications," 2007, workshop on Web Quality, Verification and Validation (WQVV), at the International Conference on Web Engineering.

[9] A. Mesbah, "Analysis and testing of ajax-based single-page web applications," Ph.D. dissertation, Delft University of Technology, 2009.

[10] NOR-STA, "Support for achieving and assessing conformance to norms and standards," (04.11.2011). [Online]. Available: http://www.nor-sta.eu/

[11] L. Cyra, J. Miler, M. Witkowicz, and M. Olszewski, "Advanced design solutions of a rich internet application," in *Zwinność i dyscyplina w inżynierii oprogramowania*, A. Jaszkiewicz, B. Walter, and A. Wojciechowski, Eds., Politechnika Poznańska. Poznań: Nakom, 2007, pp. 35–47, (In Polish).

[12] PostgreSQL. (10.06.2010). [Online]. Available: http://www.postgresql.org/

[13] JBoss. (10.06.2010). [Online]. Available: http://www.jboss.org/

[14] D. Flanagan, *JavaScript: The Definitive Guide*. O'Reilly, 2001.

[15] SunMicrosystems, "Developer resources for java technology," (10.06.2010). [Online]. Available: http://java.sun.com/

[16] W3C, "Simple object access protocol," (10.06.2010). [Online]. Available: http://www.w3.org/TR/soap/

[17] C. Larman and V. R. Basili, "Iterative and incremental development: A brief history," *Computer*, vol. Volume 36, no. 6, pp. 47–56, June 2003.

[18] PCMagazine-Encyclopedia, "Iterative development," (06.04.2011). [Online]. Available: http://www.pcmag.com/encyclopedia/

[19] C. Larman, *Agile and Iterative Development: A Manager's Guide*. Addison-Wesley Professional, 2003.

[20] Mantis. (10.06.2010). [Online]. Available: http://www.mantisbt.org/

[21] Subversion. (10.06.2010). [Online]. Available: http://subversion.tigris.org/

[22] Testng. (10.06.2010). [Online]. Available: http://testng.org/

[23] Selenium, "Selenium web application testing system." [Online]. Available: http://seleniumhq.org/projects/remote-control/

[24] W3C, "Extensible markup language," (10.06.2010). [Online]. Available: http://www.w3.org/XML/

[25] A. V. Deursen, L. Moonen, A. Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering*. Sardinia, Italy: XP2001, 2001, pp. 92–95.

[26] E. B. Swanson, "The dimensions of maintenance," in *Proceedings of the 2nd international conference on software engineering*, San Francisco, 1976, pp. 492–497.

[27] R. Black, *Advanced Software Testing - Vol. 2: Guide to the Istqb Advanced Certification as an Advanced Test Manager*. USA: Rock Nook Inc., 2009, vol. 2.

[28] R. Black and N. Atsushi, "Advanced risk based test results reporting: putting residual quality risk measurement in motion," *Software Test & Quality Assurance*, vol. Volume 7, no. issue 8, pp. 28–33, 2010.

[29] R. Black, K. Young, and P. Nash, "A case study in successful risk-based testing at ca," (06.04.2011). [Online]. Available: http://www.softed.com/resources/

[30] F. Redmill, "Theory and practice of risk-based testing," *Software Testing, Verification and Reliability*, vol. Volume 15, pp. 3–20, 2005.

[31] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," CREST Centre, King's College London, Technical Report TR-09-06, 2009.

e-Informatica Software Engineering Journal, Volume 5, Issue 1, 2011, pages: 65–76, DOI 10.2478/v10233-011-0031-3

VERSITA

Conversion of ST Control Programs to ANSI C for Verification Purposes

Jan Sadolewski*

*Department of Computer and Control Engineering, Rzeszow University of Technology

js@prz-rzeszow.pl

Abstract

The paper presents a Behavioral Interface Specification Language for control programs written in ST language of IEC 61131-3 standard. The specification annotations are stored as special comments in ST code. The code and comments are then converted into ANSI C form for further transformation with Caduceus and Why tools. Verification of compliance between specification and code is performed in Coq.

1. Introduction

In some safety oriented applications control programs should be formally proved before deployment in the controllers. Control systems are usually programmed in languages of IEC 61131-3 standard, however ANSI C is typically used for prototype systems. The IEC standard defines five programming languages, i.e. LD, IL, FBD, ST and SFC, allowing the user to choose the one suitable for particular application. Instruction list (IL) and Structured Text (ST) are text languages, whereas Ladder Diagram (LD), Function Block Diagram (FBD) and Sequential Function Chart (SFC) are graphical ones.

Recently developed compiler called MatPLC [1] converts the code from ST, IL, FBD and LD languages into ANSI C form. It seems that the main purpose of MatPLC developers was to provide equivalent ANSI C code for small hardware platforms and prototypes, where IEC languages are not available.

This paper presents somewhat different approach to code conversion, focusing instead on extension of ST language towards formal verification of compliance between specification and implementation. The conversion can also be used for *design by contract* method [2] in which clauses describe specification. The approach employs open source software Caduceus [3], Why [4] and Coq [5], whose connection can be used for formal verification of ANSI C programs. The specification is based on adaptation of JML language [6] for ST. Special annotations stored as comments express Dijkstra Weakest Preconditions [7] for programs, functions and function blocks (Program Organization Units in ST). The method presented here starts from ST source code with annotations and uses automated tools to obtain lemmas whereas approach described in chapter [8] starts from function blocks models written in Why language by hand. The annotation extending ST language was proposed in [9] and currently developed features are presented here.

The paper is organised as follows. Current state of verification of C programs, and corresponding concept of verification of ST programs are presented in Section 2. Next section briefly describes assertions and useful constructs of JML language adapted to ST. Section 4 describes translation of ST code with specification annotations to ANSI C with corresponding annotations. The translation is made automatically by program STVCGen developed for the purpose of

[0] The research has been supported by MNiSzW under the grant N N516 415638 (2010–2011).

this paper. Code translation takes into account three aspects: (1) translation of POU interfaces into C language functions, (2) conversion of POU ST code into equivalent C form, (3) translation of specification annotations into C form for Caduceus. Example of conversion of TON standard function block (timer), supplemented with specification annotations is presented in Section 5. Section 6 describes verification process of C code of the TON block (it becomes a function). The verification is processed half-automatically with standard tactics from Coq prover. For one of the lemmas the whole proof tree is presented, which can be of some help for similar examples.

2. Verification Concept

Freely available software such as Caduceus, Why and Coq can be used to verify correctness of programs written in ANSI C language. These tools may prove compliance between specification and implementation, or help to find mistakes and side effects. Specifications of programs are stored in annotations placed in special comments as BISL code (Behavioral Interface Specification Language). The Caduceus program converts the annotated C code to Why language (Fig. 1, second and third blocks). In the following step, Why generator produces verification lemmas based on Dijkstra Weakest Preconditions. Such lemmas are stored in Coq format, for further proving with tactics. If all the lemmas are proved, then correctness of the code is confirmed.

Control programs are typically written in ST language, so as to use such approach it is necessary to convert ST to C at the beginning, as shown in Fig. 1. A prototype tool called STVC-Gen described in Section 4 converts ST language code supplemented with specification annotations into C code with corresponding annotations. This is further converted by Caduceus into Why program. After applying Why generator we obtain a collection of lemmas to be proved by Coq.

3. Behavioral Interface Specification Language for ST

The main purpose for introducing the BISL languages was to define behaviour of components of developed code. Such languages are used in *design by contract* programming methods. Generally speaking the BISL languages are based on assertions, examined at run-time. Some languages like Eiffel and Why use build-in clauses for storing such assertions, but popular languages like Java and C use special kind of comments beginning with '`@`' character.

An assertion is a part of code composed of conditional Boolean expression, which should be satisfied when evaluated at specific place of the executed program (i.e. it returns true). Typical assertions from popular languages are shown in Tab. 1. They are used solely for testing purposes, and their code is not compiled into the final distribution. Assertion failure may be represented by message box, with exception or interruption of program execution. The message may involve current call stack, place in source code, etc.

Design by contract uses two special assertions, i.e. `requires` to denote preconditions, and `ensures` for postconditions. They must be kept near developed code, in the form of special comments beginning with '`@`' mentioned above. The assertions express conditions, which must be satisfied when given subroutine is called, and conditions guaranteed at its termination.

Java Modelling Language (JML) is an example of a BISL language, which uses comments to store annotations. This feature allows for code migration between compilers of different providers, which do not support annotations. Program Organization Units (POUs) from IEC standard are similar to lightweight Java objects, so JML can be adapted as a base of BISL language for ST. Naturally, only a subset of JML will suffice for verification problem considered here.

Adaptation of JML for ST language is presented at Tab. 2. The clauses are grouped according to their types. Each clause has its own range. Range *instruction* means that corresponding clause can be placed where instruction or expression is expected. Ranges *local* and *global* re-

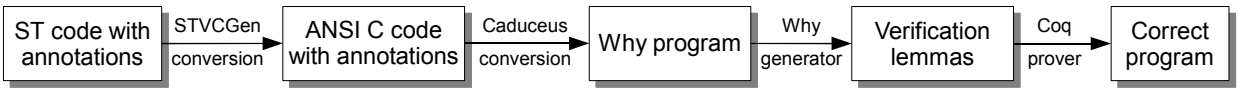


Figure 1. Method for verification of ST programs

Table 1. Assertions in popular languages

ANSI C, POSIX	Delphi	C#

`#include <assert.h>`
`void additem(struct`
`ITEM *itemptr)`
`{ assert(itemptr`
`!= NULL);`
`... }`
	`procedure ModifyStorage`
	`(AStorage: TStorage;`
	`const s: string); begin`
	`Assert(AStorage <> nil,`
	`''); AStorage.Data := s;`
	`... end;`
		`{ int index;`
		`...`
		`System.Diagnostics`
		`.Debug.Assert`
		`(index > -1);`
		`... }`

Table 2. Adaptation of JML in ST language

Type	Standard JML	ST adaptation	Range
Assertions	`assert`	`assert`	instruction
	`ensures`	`ensures:`	local
	`requires`	`requires:`	local
Localise modifiers	`\at`	`\at` or `at`	instruction
	`\old`	`\old`	instruction
Quantifiers	`\exists`	`\exists`	mixed
	`\forall`	`\forall`	mixed
Invariant	`invariant`	`invariant:`	instruction
Declarations	`label`	`label:`	instruction
	`logic`	`logic:`	global
	`ghost`	`ghost:`	local
	`predicate`	`predicate:`	global
	`axiom`	`axiom:`	global
Function return value	`\result`	`\result` or *function_name*	local
Operations	`set`	`set:`	instruction
	`assigns`	`assigns:`	local
W-F iteration	`variant`	`variant:`	instruction

fer to POU or whole project, respectively. Clause whose use depends on the context has the range *mixed*.

Verification clauses are located inside corresponding program unit. For example, annotation clause of function block is written after identifier with the name of the block. The clause must contain at least `ensures` section, but it often involves `requires` and `assigns`, especially when annotated POU is a program which modifies global variables. There are two ways to access return value of function, i.e. `\result` or function name, which is specific for ST language. Verification is based on memory states [10], which contain variable values at specified moment of execution. Modifier `\old` represents variable value at the beginning of execution, obtained in previous cycle. Similarly, modifier `\at` denotes variable value at specified location in the code, declared with `label`.

Sometimes additional function that does not appear in the original code may help in construction of specification. The function can be reached by global `logic` clause. Additional local variables can be used to express the specification. Such variables are defined by `ghost` clause and operated on by `set` clause. The `predicate` declares additional logic function, which returns Boolean value. The `axiom` generates new axiom which can be used by the prover. Quantifiers appear in

declarations of loop invariants. They may also examine if the loops are well founded.

More details on adaptation of JML for ST are given in [9].

4. Conversion of ST to ANSI C

As indicated in Section 2, conversion of POUs from ST language into ANSI C code is needed to use open source tools for program verification. The STVCGen tool based on ST compiler from CPDev package [11] executes the conversion. Components of the compiler are classes in C# language, so they can be reused with typical mechanisms like inheritance and overriding. Main goal while developing the STVCGen has been to get a compiler quickly from existing code of CPDev. The parser is built according to top-down scheme with syntax-directed translation [12]. It recognises meaning of ST code and produces corresponding ANSI C code. In addition to translating ST, the parser collects annotations and generates code for Caduceus or Frama C tools[1].

Code translation is performed in three aspects, i.e. concerning POUs, instructions, and annotations, respectively. The first one is to translate POUs into C language functions. If POU is a function, then translation proceeds directly. Return value must be declared only to conform with the code. Translation of function block or program is more complicated. Function block is translated into C function in the following way:

- block inputs are converted into function parameters,
- block outputs become function parameters, however declared as pointers,
- local variables are also declared as pointer parameters,
- all pointer parameters produce extra `requires` expression with different base addresses.

An ST program is translated into C as follows:

- global variables remain global in C,
- local variables become function parameters, declared as pointers,
- local function block instances are ignored, but their pointer parameters are also declared as additional pointers.

The conversion cases are illustrated in Fig. 2. ST variable types are converted into corresponding C types, with equivalents presented in Table 3.

The second aspect is to convert instruction code into valid C form. Generally speaking, code shape in both languages is similar, so examples presented at Fig. 3a where *OP* is arithmetic or logic operator are natural. Most of ST operators have equivalents in C, so C code construction involves operator replacements and parentheses in case of different priorities. The problem arises when converted variable after conversion is declared as a pointer. In such case each instance must appear in C code with a star and parentheses. If an ST operator does not have C equivalent (like power `**`), STVCGen replaces it with function provided by header file bundled with the tool, as in Fig. 3a (macros). Some ST operators have more equivalents in C code. For example `AND` operator may be logical operator between Boolean expressions $\texttt{bexpr}_1$ and $\texttt{bexpr}_2$ (Fig. 3b), and can be also used for bitwise calculations in digital expression involving $\texttt{dexpr}_1$ and $\texttt{dexpr}_2$. When such operator (`AND`, `OR`, `NOT`) appears in source code, the compiler checks if the expression evaluates to Boolean. If yes, the logical operator is used, otherwise bitwise one. Conversion of `NOT` operator may lead to one of two macros. When the operand is Boolean then the `NOT` operator is converted to '`!`' in C hidden under `_BOOL__NOT__` macro. If the operand is bitwise, `NOT` is converted to '`~`' in `_BIT__NOT__`.

Some ST and C constructs are very similar, as `IF` statement in Fig. 3c. Conditional Boolean expression remains valid after conversion into C. This does not happen however, in case of `FOR` loop whose conversion depends on values in source code. If the constant $\texttt{im}_3$ in Fig. 3d is greater than zero, then the equivalent C statement uses less or equal comparison and increment operator. If the constant is lower than zero, then the C statement uses greater or equal comparison and decrement operator.

[1] Due to different annotations, Caduceus or Frama-C are chosen by compiler settings.

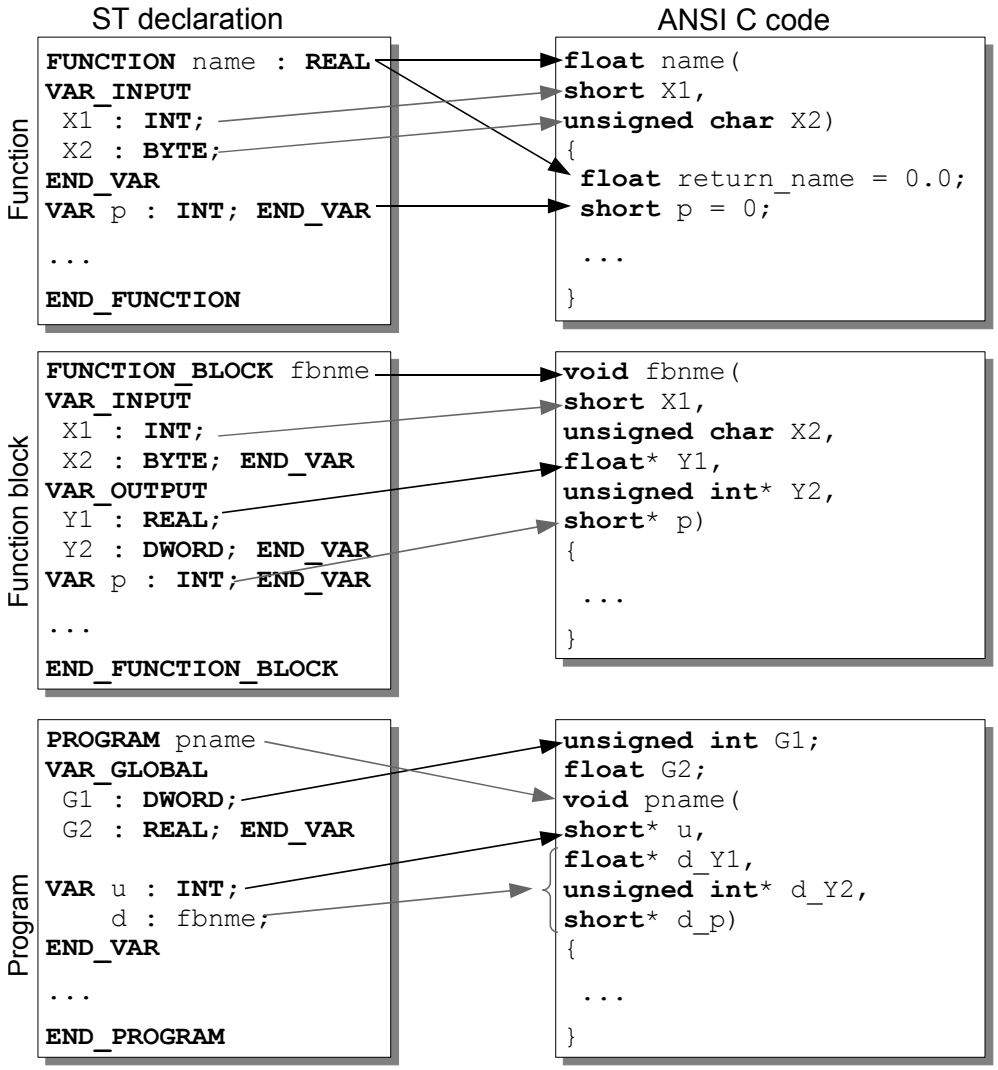


Figure 2. Conversion of three types of POUs into C

Calls of instances of function blocks require more effort, because values of local and output variables from previous execution must be preserved. As shown before in Fig. 2, the program `pname` uses a hypothetical function block `fbname` with the instance called `d`, so additional function inputs (beginning with `d_`) have also been declared. Call of the instance `d` in ST and the translation to ANSI C are presented in Fig. 4. The single variable `d` does not exist here, but is replaced by corresponding arguments of the converted program. Such approach produces less complicated verification lemmas, which can be proved half automatically.

The third aspect of conversion is to change annotations describing a POU in ST language into equivalent form in C with necessary modifications and supplements.

Converted annotations do not differ much from original ones, except operator syntax and removal of some characters not needed by Caduceus (ST assertional extension involves characters that specify range and objective of some clauses). However, the conversion generates additional components in specification, mostly describing pointer properties and arithmetic, including different base addresses for pointer variables and their non-`NULL` values. Since pointers do not exist in ST, therefore each variable, so also a pointer, is allocated at different address. Values different than `NULL` are preserved by task allocator, which can execute programs only with

Table 3. Type conversion

ST type	C type	ST type	C type
BOOL	char	BYTE	unsigned char
SINT	char	INT	short
WORD	unsigned short	DINT	int
DWORD	unsigned int	LINT	long long
REAL	float	LREAL	double
LWORD	unsigned long long	TIME	int
DATE_AND_TIME	unsigned long long	TIME_OF_DAY	unsigned int

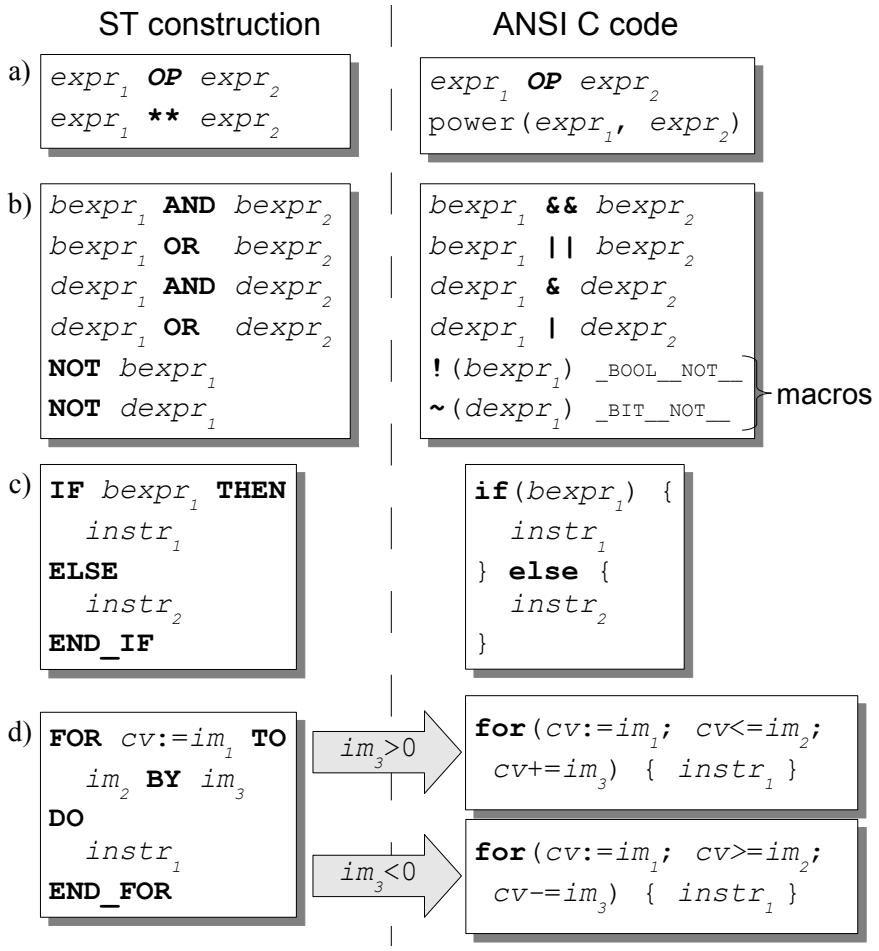


Figure 3. ST code conversion to C form

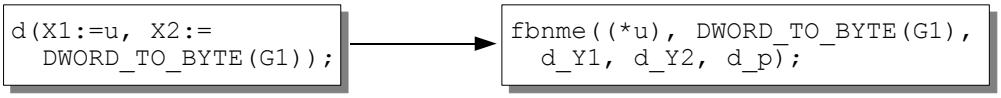


Figure 4. Conversion of function block call

complete set of parameters. The example in Fig. 5 presents an instance of assertional ST extension and converted form in ANSI C for Caduceus. The clause `requires` is directly converted into destination form and supplemented with expression (denoted by circled 1) which by the clause `\valid` indicates non-`NULL` values of pointer variables, and by the clause `\base_addr` assures different addresses pointed by the pointers. The use of pointer variables at C side also requires `assigns` clause (circled 2), which defines variables changed by the function.

Application of the three translation aspects in STVCGen produces coherent ANSI C code, which can be handled by verification tools like Caduceus, Why and Coq.

5. Example of TON function block

As stated in Sec. 2, the sequential verification process consists of source code transformations from ST language with annotations through ANSI C and Why into verification lemmas (Fig. 1). The example considered now involves function block TON (on-delay timer) of Fig. 6a, whose input-output time plots are shown in Fig. 6b. The plots can be split into three parts (states) denoted by the circled digits. The ST source code with specification annotations at the beginning is presented in Fig. 7. Each part of the plot is associated with a single line in `ENSURES` specification clause. In design by contract approach the clause expression and block interface (inputs and outputs declaration) are written by designer. Construct *var*`<>FALSE` implies that Boolean variable *var* equals `TRUE`. It is necessary, because strict Boolean type does not exist in C language. Here it is simulated by integer value zero (`FALSE`) and non-zero (`TRUE`).

The implementation code beginning from `IF` defines instructions to be performed. The `REQUIRES` clause defines constraints. If they are not satisfied, execution of the block may return invalid results. The constraints are also used in verification. The ST code from Fig. 7 is translated by STVCGen to ANSI C form presented in Fig. 8 (in printable version[2]).

According to Sec. 4, function block TON becomes function in C, and `requires` clause is strengthened with `\valid` and `\base_addr` constructs. Block outputs and local variables become pointers in C, so additional clause `assigns` is necessary to deal with pointer arithmetic while proving. Transformation of function block body applies statement conversion and pointer substitution of some variables.

6. ANSI C Verification

The ANSI C code of Fig. 8 is further converted with Caduceus which produces equivalent program in Why code (Fig. 2). In the next step the Why tool generates verification lemmas, which must be proved to confirm program correctness. Details of Caduceus and Why conversion are skipped due to limited space. Here we focus on the lemmas produced by Why generator. In case of TON function (Fig. 8), Why produces 12 lemmas which must be proved with Coq Proof Assistant. First four lemmas refer to correct allocation of variables declared as pointers. One of them is presented in Fig. 9, remaining lemmas have different variable in the goal part (last not indented line). They are easily proved with default tactic `intuition`. Fifth lemma listed in Fig. 10 deals with first possible execution of the program and is more complex. Using `intuition` tactic to prove it leads to undetermined value. This means that `intuition` must be replaced by elementary tactics.

At first `intros` tactic is applied, which introduces local hypothesis into the context. The following `repeat split` splits the goal into five subgoals (denoted as circled numbers in Fig. 11). The first subgoal can be proved by sequential reduction of subsequent memory states (`subst intM_global` with appropriate number), and `caduceus` tactic, when reduction reaches the initial state. The second subgoal invloves contradiction in hypotheses, so one of the opposite hypotheses is passed as argument to the `absurd`

[2] Actually STVCGen produces output with a lot of brackets, so it is not easily readable.

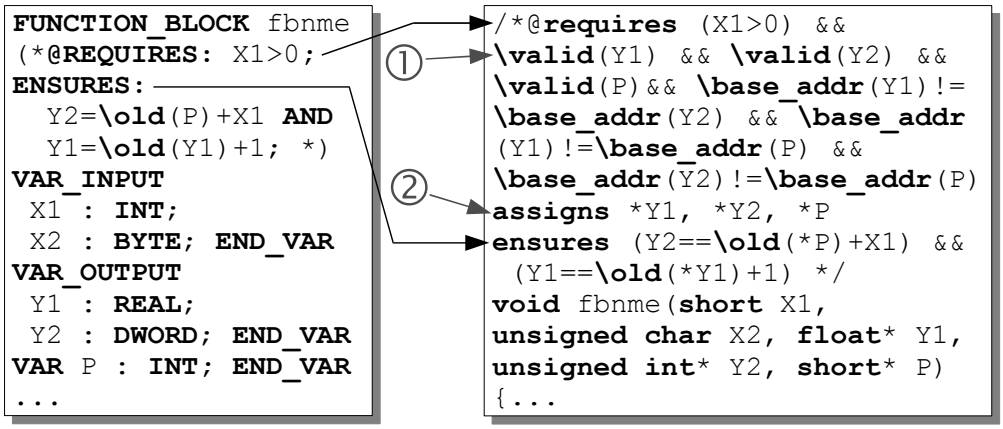


Figure 5. Converting assertional extension with supplements

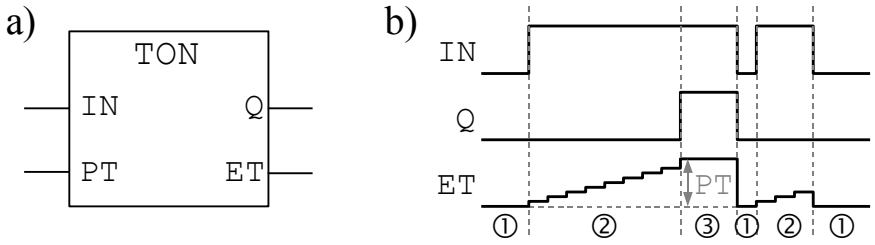


Figure 6. TON function block: a) symbol, b) time plots

```
FUNCTION_BLOCK TON
(*@REQUIRES: (PT > TIME#0ms) AND (ET >= TIME#0ms) AND (ET <= PT);
ENSURES: ((IN = FALSE) ==> ((ET=TIME#0ms) AND (Q=FALSE))) AND
(((ET < PT) AND (\old(Q)=FALSE) AND (IN<>FALSE)) ==> (Q=FALSE)) AND
(((ET = PT) AND (\old(Q)=FALSE) AND (IN<>FALSE)) ==> (Q<>FALSE));  *)
VAR_INPUT  IN : BOOL;  PT : TIME;  END_VAR
VAR_OUTPUT  Q : BOOL;  ET : TIME;  END_VAR
VAR   L_STIME : TIME;  LC : TIME;  END_VAR

IF IN THEN
  IF NOT Q THEN
      LC := CUR_TIME() - L_STIME;
       IF LC >= PT THEN  Q := TRUE;  ET := PT;
       ELSE   ET := LC;
       END_IF
  END_IF
ELSE
     Q := FALSE;  ET := TIME#0ms;  L_STIME := CUR_TIME();
END_IF
END_FUNCTION_BLOCK
```

Figure 7. Source code of TON function block

```
/*@requires (((PT>0x00000000) && (*ET>=0x00000000)) && (*ET<=PT)) &&
\valid(Q) && \valid(ET) && \valid(L_STIME) && \valid(LC) &&
\base_addr(Q)!=\base_addr(ET) && \base_addr(Q)!=\base_addr(L_STIME) &&
\base_addr(Q)!=\base_addr(LC) && \base_addr(ET)!=\base_addr(L_STIME) &&
\base_addr(ET)!=\base_addr(LC) && \base_addr(L_STIME)!=\base_addr(LC)
 assigns *Q, *ET, *L_STIME, *LC
 ensures ((IN==0) => ((*ET==0x00000000) && (*Q==0))) &&
  (((((*ET<PT) && (\old(*Q)==0)) && (IN!=0)) => (*Q==0)) &&
  ((((*ET==PT) && (\old(*Q)==0)) && (IN!=0)) => (*Q!=0))) */
void TON (char IN, int PT, char* Q, int* ET, int* L_STIME, int* LC)
{
 if(IN) {
    if(_BOOL__NOT__(*Q)) {
      *LC = CUR_TIME() - *L_STIME;
      if(*LC >= PT) { *Q = 1; *ET = PT; }
      else { *ET = *LC; }
    }
 } else {
 *Q = 0; *ET = 0x00000000;  *L_STIME = CUR_TIME();
 }
}
```

Figure 8. Result of ST conversion to ANSI C

```
Lemma TON_impl_po_1 :
 forall (IN: Z), forall (PT: Z), forall (Q: (pointer global)),
 forall (ET: (pointer global)), forall (L_STIME: (pointer global)),
 forall (LC: (pointer global)), forall (alloc: alloc_table),
 forall (intM_global: (memory Z global)),
 forall (HW_1: ((((((((((((PT > 0 /\ (acc intM_global ET) >= 0) /\
 (acc intM_global ET) <= PT) /\ (valid alloc Q)) /\ (valid alloc ET)) /\
 (valid alloc L_STIME)) /\ (valid alloc LC)) /\
 ~((base_addr Q) = (base_addr ET))) /\
 ~((base_addr Q) = (base_addr L_STIME))) /\
 ~((base_addr Q) = (base_addr LC))) /\
 ~((base_addr ET) = (base_addr L_STIME))) /\
 ~((base_addr ET) = (base_addr LC))) /\
 ~((base_addr L_STIME) = (base_addr LC)))), forall (HW_2: IN <> 0),
(valid alloc Q).
```

Figure 9. First lemma generated by the Why

tactic. Two generated subgoals are proved with `assumption`. The third subgoal is proved in the same way as the first one. The fourth subgoal, after introduction of additional hypothesis and decomposition of the conjunction in hypothesis `HW_1`, can also be proved like the first subgoal.

The fifth subgoal requires more effort, because it begins with `not_assigns` clause for pointer arithmetic. However, standard scheme described in [3] can be applied to prove corresponding lemmas, extended here to handle four pointer variables (instead of one). At first the `intros A B C`[3] tactic is applied. Then we must duplicate the last lemma `C` so many times, as to match the number of variables declared as pointers, except the first one (so three here). It is done with `generalize C` and `intro D`, or `E`, or `F` tactics (see Fig. 11). Next the `apply`

[3] A sequence of Latin letters is used for brevity. If one of them is already used in the lemma, it can be replaced with another unique name.

```
Lemma TON_impl_po_5 :
 forall (IN: Z), forall (PT: Z), forall (Q: (pointer global)),
 forall (ET: (pointer global)), forall (L_STIME: (pointer global)),
 forall (LC: (pointer global)), forall (alloc: alloc_table),
 forall (intM_global: (memory Z global)),
 forall (HW_1: ((((((((((PT > 0 /\ (acc intM_global ET) >= 0) /\
  (acc intM_global ET) <= PT) /\ (valid alloc Q)) /\ (valid alloc ET)) /\
  (valid alloc L_STIME)) /\ (valid alloc LC)) /\
  ~((base_addr Q) = (base_addr ET))) /\
  ~((base_addr Q) = (base_addr L_STIME))) /\
  ~((base_addr Q) = (base_addr LC))) /\
  ~((base_addr ET) = (base_addr L_STIME))) /\
  ~((base_addr ET) = (base_addr LC))) /\
  ~((base_addr L_STIME) = (base_addr LC)))),
 forall (HW_2: IN <> 0),
 forall (HW_3: (valid alloc Q)), forall (result: Z),
 forall (HW_4: result = (acc intM_global Q)),
 forall (HW_5: result = 0),
 forall (result0: Z), forall (HW_7: (valid alloc L_STIME)),
 forall (result1: Z),
 forall (HW_8: result1 = (acc intM_global L_STIME)),
 forall (HW_9: (valid alloc LC)),
 forall (intM_global0: (memory Z global)),
 forall (HW_10: intM_global0 = (upd intM_global LC (result0 - result1))),
 forall (HW_11: (valid alloc LC)),
 forall (result2: Z),
 forall (HW_12: result2 = (acc intM_global0 LC)),
 forall (HW_13: result2 >= PT),
 forall (HW_14: (valid alloc Q)), forall (intM_global1: (memory Z global)),
 forall (HW_15: intM_global1 = (upd intM_global0 Q 1)),
 forall (HW_16: (valid alloc ET)),
 forall (intM_global2: (memory Z global)),
 forall (HW_17: intM_global2 = (upd intM_global1 ET PT)),
((((IN = 0 -> (acc intM_global2 ET) = 0 /\ (acc intM_global2 Q) = 0)) /\
 ((((acc intM_global2 ET) < PT /\ (acc intM_global Q) = 0) /\ IN <> 0 ->
 (acc intM_global2 Q) = 0))) /\ ((((acc intM_global2 ET) = PT /\
 (acc intM_global Q) = 0) /\ IN <> 0 -> (acc intM_global2 Q) <> 0))) /\
 (not_assigns alloc intM_global intM_global2 (pset_union (pset_singleton
 LC) (pset_union (pset_singleton L_STIME) (pset_union (pset_singleton ET)
 (pset_singleton Q))))))).
```

Figure 10. Fifth lemma generated by Why

`pset_union_elim1 in C` is applied for hypothesis `C` to eliminate first set in the union of sets with pointer variables (`pset`). For the second hypothesis `D` the tactic `elim2` is applied first, followed by `elim1` as before. The remaining `E` and `F` hypotheses require increase of `elim2` applications by one, followed by single `elim1` in `D`, and not in the last `F`. After these steps the tactic `apply pset_singleton_elim in _` applied for all four hypotheses provides pure distinction between addresses of pointer variables. Last steps involve the approach used already to prove the first and third subgoal, i.e. `subst intM_global` tactic to reduce memory states, terminated by final `caduceus` tactic.

The remaining lemmas 6, 8, 9, 10 and 11 are proved automatically with `intuition` tactic. Lemmas 7 and 12 can be proved similarly as lemma 5.

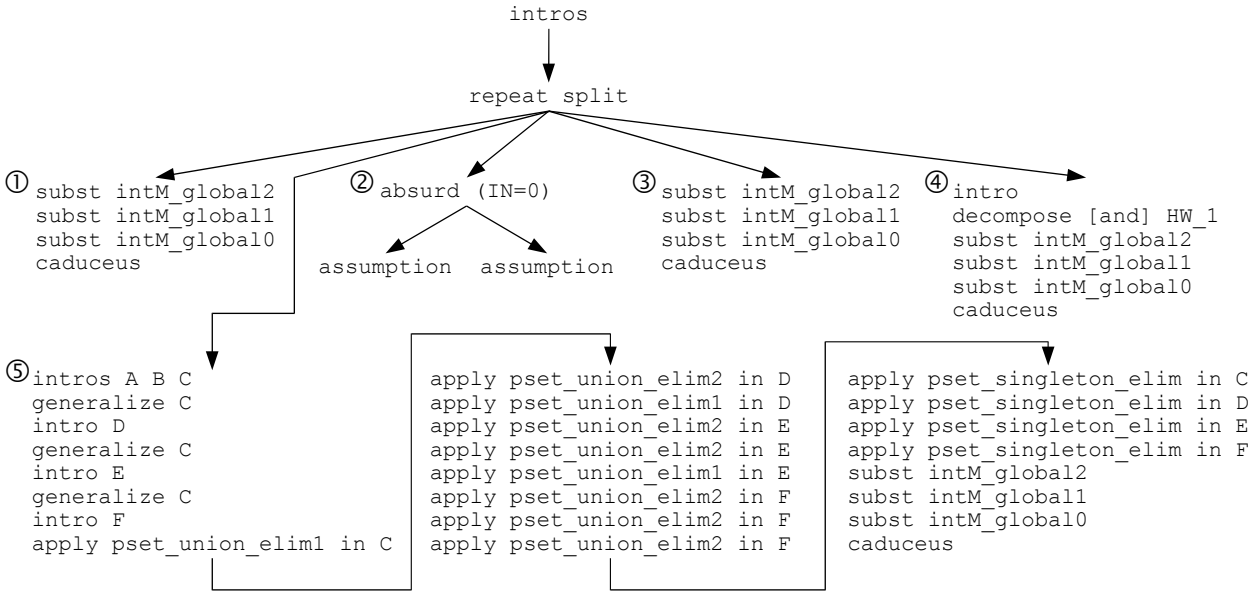


Figure 11. Tactics of proof tree for TON function block lemma

The method presented here can be used for verification of simple functions, function blocks and programs which do not call other function blocks. The limitation is caused by annotation injection arising when a subroutine is called, so some kind of decomposition must be used to deal with it. More information on decomposition can be found in [8].

7. Summary

The application of Behavioral Interface Specification Language for ST language of IEC 61131-3 standard concerning control programs has been presented. The annotations to ST code express specification of function, function block or program, which after conversion to C can be used for formal verification of compliance between specification and implementation. Such approach is typical for design by contract method applied while developing advanced applications. Stepwise conversion by STVCGen, Caduceus and Why tools produce verification lemmas which can be proved by Coq with a set of appropriate tactics. Till now several function blocks and programs have been verified. The examples involve combinatorial logic (binary multiplexer, two-bit sum, heater control), sequential logic (flip-flops, water level control, wood sorting machine), and sequential logic with time constraints (cargo lift).

Specification in the form of annotations is transparent for compilers which do not support such assertions. Therefore for practical reasons, one general purpose IEC 61131-3 compiler may be used for verification, and another one, dedicated to particular hardware, applied for implementation. The presented compiler may be also extended to perform dynamic run-time verification, as provided by JML with some supporting tools.

Future work will concentrate on direct conversion from ST language into Why code, without limitation of available types. The types constrained by Caduceus conversion will be transformed to suit types provided by Coq or by external libraries. Naturally, direct conversion will require some additional algorithms to construct proofs of the lemmas.

References

[1] E. Tisserant, L. Bessard, and M. de Sousa, "An open source IEC 61131-3 integrated development environment," in *5th Int. Conf. Industrial Informatics*. Piscataway, NJ, USA, 2007.

[2] B. Meyer, "Applying "design by contract"," *Computer*, vol. 25, no. 10, p. 40–51, 1992.

[3] J.-C. Filliâtre, T. Hubert, and C. Marché, "The

Caduceus verification tool for C programs," [online] http://caduceus.lri.fr, 2008.

[4] J.-C. Filliâtre, "The Why verification tool. tutorial and reference manual," [online] http://www.lri.fr, 2010.

[5] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development*. Springer-Verlag, Berlin Heidelberg, 2004.

[6] G. T. Leavens, A. L. Baker, and C. Ruby, *JML: a Notation for Detailed Design*, ser. Behavioral Specifications of Businesses and Systems, 1999.

[7] E. W. Dijkstra, *A Discipline of Programming*. Prentice-Hall, Inc., 1976.

[8] J. Sadolewski, *Verification of complex programs for control systems*, ser. Methods of producing and applying real time systems. Wydawnictwa Komunikacji i Łączności, 2010, (in Polish).

[9] ——, "Assertional extension in ST language of IEC 61131-3 standard for control systems dynamic verification," *Pomiary Automatyka Robotyka*, no. 2, pp. 305–314, 2011, (in Polish).

[10] R. Bornat, "Proving pointer programs in Hoare logic," in *Mathematics of Program Construction*. Springer-Verlag. London, 2000, pp. 102–126.

[11] D. Rzońca, J. Sadolewski, A. Stec, Z. Świder, B. Trybus, and L. Trybus, "Open environment for programming small controllers according to IEC 61131-3 standard," *Scalable Computing Practice and Experience*, vol. 10, no. 3, pp. 325–336, 2009.

[12] K. D. Cooper and L. Torczon, *Engineering a Compiler*. Morgan Kaufmann, San Francisco, 2003.

e-Informatica Software Engineering Journal, Volume 5, Issue 1, 2011, pages: 77–85, DOI 10.2478/v10233-011-0032-2

VERSITA

Multiple tasks in FPGA-based programmable controller

Zbigniew Hajduk*, Jan Sadolewski*, Bartosz Trybus*

*Faculty of Electrical and Computer Engineering, Department of Computer and Control Engineering, Rzeszow University of Technology

zhajduk@kia.prz.edu.pl, js@kia.prz.edu.pl, btrybus@kia.prz.edu.pl

Abstract

An FPGA-based execution platform for PLC controllers with capability to run multiple control tasks is presented. The platform, called multi-CPCore, uses hardware virtual machines to execute control tasks defined in CPDev engineering environment. The tasks consist of one or more programs written in IEC 61131-3 languages, such as ST, IL or FBD. They may run with different cycles and communicate via global variables. Parallel programming mechanisms like process image and semaphores are provided to handle potential conflicts when accessing shared resources.

1. Introduction

CPCore (Control Program Core) is an execution platform for programmable logic controllers (PLCs) designed in FPGA technology. Hardware-implemented IEC 61131-3 virtual machine [1, 2] is its main feature. CPCore is programmed in CPDev engineering environment (*Control Program Developer*) [3], which integrates tools for programming, simulation, hardware configuration, on-line testing and running control applications on different platforms. Programs can be written in ST and IL textual languages (Structured Text, Instruction List) and in FBD graphical language (Function Block Diagram). CPDev compiler produces universal executable code called VMASM (Virtual Machine Assembler), interpreted at the target controller by the CPDev virtual machine (runtime). Software virtual machine written in C is available for multiple target platforms with general-purpose CPUs (ARM, AVR or x86). However, interpretation takes time, so portability of the VMASM code is achieved at the price of slower program execution.

On the contrary, CPCore platform involves FPGA-based hardware virtual machine which directly executes VMASM code. This results in much shorter execution time, from several to a few hundred times, when compared with typical microcontrollers [1]. Similar solutions have been presented in [4, 5]. The CPCore hardware machine is actually a 32-bit microcontroller that executes VMASM code generated in CPDev environment. The microcontroller is built according to Harvard architecture with separate data and program buses. A prototype PLC controller implemented in CPCore technology is shown in Fig. 1. The main board (upper left) includes Xilinx FPGA chip, real-time clock, SRAM, NVRAM and Flash memories. Analog and binary inputs and outputs are handled by slave boards (right side of Fig. 1). The operating panel (lower left) involves LCD display, LEDs and push buttons.

Ability to run multiple IEC control tasks at the same time has been introduced to the CPCore platform recently and is described here.

[1] The research has been supported with MNiSzW grant N N514 412736 (2009-2011).

Each task is executed by a single hardware machine core. The cores are independent and run in parallel, creating a *multiprocessor* architecture (multi-CPCore).

The paper is organized as follows. At first, program execution is characterized. Then the multiple task capabilities are described from the programmer's viewpoint, with the process of creating tasks and setting their parameters. Main aspects of resource sharing are covered, i.e. accessing common hardware blocks and using semaphores for mutual exclusion. Finally, hardware structure of the multi-CPCore platform is presented.

2. Program execution

Source programs for the CPCore controller are processed by the CPDev compiler which generates VMASM universal executable code [3]. The VMASM code can be executed by the controller hardware machine (*executor*). Functional side of the machine corresponds to IEC 61131-3 standard [2] and provides the following capabilities:

- Handling IEC data types: Boolean BOOL, integer BYTE, SINT, INT, WORD, DINT, LINT, DWORD, LWORD, real REAL, LREAL, time and date TIME, DATE, TIME_OF_DAY, DATE_AND_TIME.
- Execution of functions: arithmetic ADD, SUB, MUL, DIV, MOD, numerical SQRT, LOG, SIN, ASIN, EXP, Boolean NOT, AND, OR, XOR, bit shift SHL, ROL, comparison GT, GE, LT, EQ and others.
- Program flow control by means of jumps JMP, JZ, JNZ, calls of function blocks CALB, early exit RETURN, memory handling MCD, MEMCP (Move from Code to Data, Memory Copy).
- Hardware function blocks: invoking blocks implemented in FPGA hardware (called also *native blocks*).
- Parallel programming instructions: global variable handling (Sec. 4.1), semaphore operations LOCK, UNLOCK, TRYLOCK (Sec. 4.3).

The IEC standard also defines multi-element variable types, i.e. arrays and structures. The machine handles these two types by means of a few dedicated commands, e.g. AURD/AUWD read/write data from/to indexed array.

Basic logical registers of the hardware machine are listed in Table 1. Since accumulator does not exist in VMASM specification, results of commands are stored in variables. *Task cycle* can be configured and monitored by the machine during program execution. *Actual task cycle* (last value) is particularly useful for on-line testing (commissioning). *Status1* stores exception flags, including cycle overflow, therefore appropriate reaction can be programmed.

3. CPCore programming with multiple tasks

In the multi-CPCore solution, each task contains its own control algorithm compiled into VMASM. The task is executed by separate instance of the hardware machine. This means that multi-CPCore can be viewed as a group of virtual controllers.

According to IEC 61131-3 standard, engineering project of control system is created in hierarchical manner, i.e. by defining controller configuration, implementing algorithms in programs and function blocks, and assigning them to tasks.

3.1. Creating POUs

In the CPDev environment, the user creates a set of so-called *Program Organizational Units* (POUs). The POUs can be written in ST language or two other languages of IEC 61131-3 standard, i.e. IL and FBD. Main window of CPDev in Fig. 2 shows sample configuration with four POUs, whose names are seen in the project tree on the left. The tree also contains global variables used in the project. START, STOP and ALARM represent digital inputs, while the other, MOTOR, PUMP, OUT0...OUT3 are outputs (all are BOOLs). The global variables are followed by two tasks (described later) and libraries with blocks used by POUs. The first

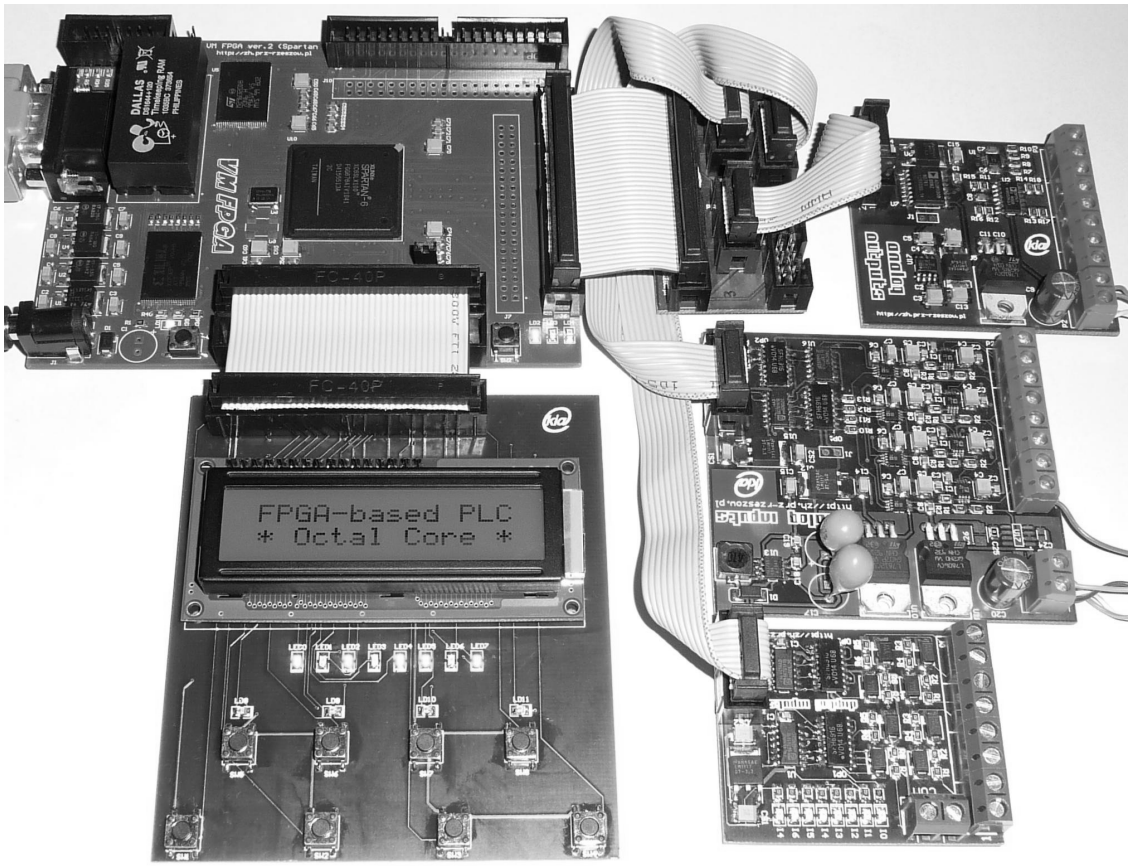


Figure 1. CPCore controller prototype

POU, START_STOP, has been created as an FBD diagram (center part of Fig. 2). It turns MOTOR on if START is pressed, provided that STOP and ALARM are not set. MOTOR continues running after releasing START. PUMP is turned on and off 5 seconds after the MOTOR. Time delay (T#5s) is introduced by two function blocks, TON and TOF. The second POU, MOVE_UNIT, subsequently turns on and off a set of devices in a loop. The algorithm written in ST (right side of Fig. 2) sequentially sets to TRUE one of the global variables OUT0...OUT3 assigned to binary outputs. It is done every 2 seconds (t#2s) by using system clock (CUR_TIME function).

Besides START_STOP and MOVE_UNIT, there are also two other POUs in the project tree, namely LCD_CH and DISPB. The first one is a hardware function block which puts a character onto the CPCore LCD. DISPB is another function block which uses LCD_CH internally to display a string. The block can be invoked by other POUs to print some messages. The two blocks will be described in Sec. 4.2.

3.2. Defining tasks

The user creates a task by selecting appropriate POUs and assigning them to the task. The POUs assigned to the task are executed sequentially in the order defined by the user. The task can group a set of POUs written in different IEC languages. In the sample project of Fig. 2 two task are defined, TASK_SS and TASK_MU. Creation of TASK_SS will be described in more detail.

Task definition is done in CPDev window of Fig. 3. Task name must be entered first, so TASK_SS here. Then task type is selected to indicate execution mode. TASK_SS will be executed cyclically, with cycle time of 1 millisecond. Cyclic task is most common choice, but one can also select “As soon as possible” (endless

Table 1. Logical registers of hardware machine

Register	*Function*
Program counter	Indicates next VMASM command
Data offset	Index to data area being used
Call stack pointer Data stack pointer	Pointers to call stack (POUs) and data stack
Task cycle Actual task cycle	Configured and measured task cycle
Cycle counter	Counts cycles (from reset)
Status1	VM status word (array index faulty, time cycle exceeded, cold start, etc.)
RTC clock	Absolute time

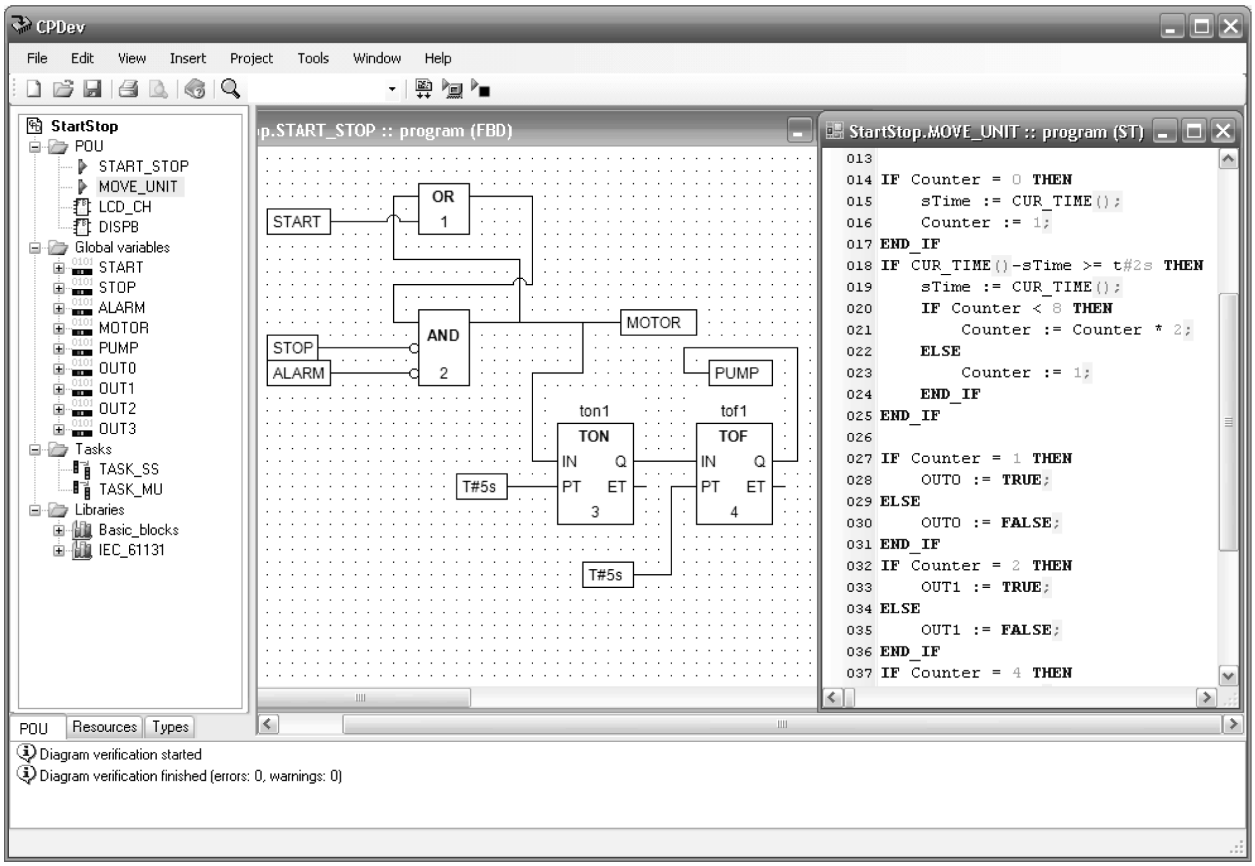


Figure 2. CPDev environment window with two POUs

loop) or single execution (not implemented yet in multi-CPCore prototype). POU assignment is done by moving available programs from the right list to the left (Fig. 3). Two POUs are available here, START_STOP and MOVE_UNIT. The other POUs of the project, LCD_CH and DISPB, do not appear in the window because they are function blocks, not programs. In case of TASK_SS, only START_STOP is selected for execution. The other task, TASK_MU (Fig. 2), involves the MOVE_UNIT program and is also executed cyclically with period of 10 ms.

4. Task communication and resource sharing

As mentioned before, the tasks in multi-CPCore are run independently by their own executors.

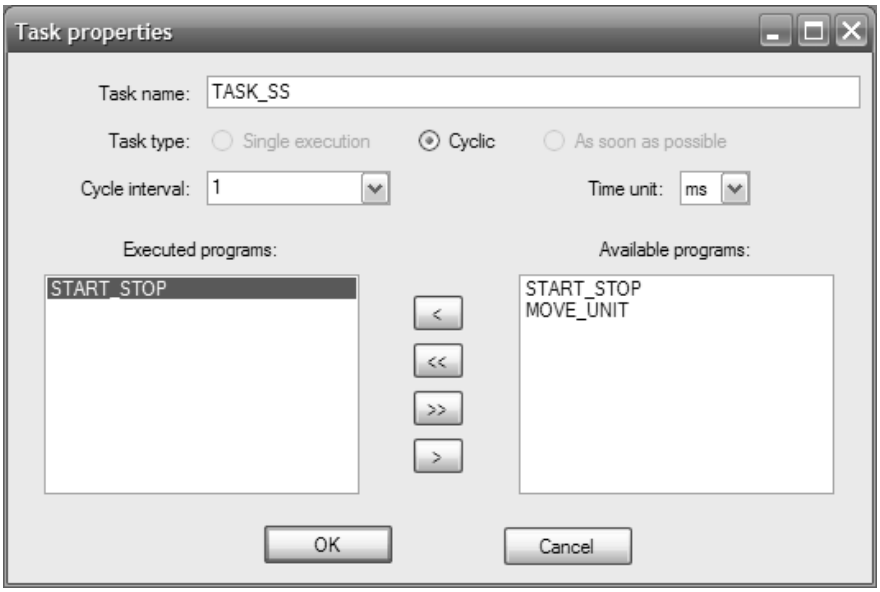


Figure 3. Defining a task for multi-CPCore controller in CPDev

However, as parts of control project, they must communicate and exchange variables. Basic problem in parallel programming is to get access to shared resources. Multi-CPCore can be viewed logically a set of virtual PLCs, which share the same peripherals (inputs and outputs, display, real-time clock, etc.).

4.1. Global variables

Data exchange between CPCore tasks is performed by means of global variables. Such way of task communication is also recommended in IEC standard [2]. Global variables can be accessed by programs and tasks. In the sample project of Fig. 2, START and STOP are used in both tasks (TASK_SS and TASK_MU) to activate corresponding devices (not shown, however, in the part of MOVE_UNIT code).

Upon start of the configuration, multi-CPCore executes special initialization code generated by CPDev compiler, which sets initial values of the global variables (e.g. STOP:=FALSE). This is done before any task is invoked.

To avoid conflicts related to sharing global variables between tasks, CPCore executors operate on so-called *process images*. At the start of the cycle, the task is provided with current copy of the global variables (local shadows). When the task is executed, only the shadows are used, so change of global values caused by other tasks does not affect calculations. When the cycle is about to end, the calculated shadows are stored in the global variables. Synchronization is done only for the variables that have been modified within the cycle.

4.2. Accessing hardware blocks

Two types of function blocks are available in the CPCore platform, i.e. program blocks and hardware blocks. The first ones are created in CPDev environment in one of IEC languages. The platform also supports a set of dedicated hardware blocks used to access low-level functions or to speed up calculations.

As an example, access to the hardware block LCD_CH mentioned above is described now. LCD_CH displays a character on the controller LCD. First, the user creates a new function block (new POU) and chooses ST as implementation language. However, instead of writing an implementation, only the following declaration of the block is entered:

```
FUNCTION_BLOCK LCD_CH
 (*$HARDWARE_BODY_CALL ID:0004; Align:4;
                                Extra:0;*)
```

```
 (*$PLACE_UID_VAR*)
VAR_INPUT
  C:BYTE;
END_VAR
VAR_OUTPUT
END_VAR
END_FUNCTION_BLOCK
```

As seen, single block input is declared (a BYTE value) and no outputs. There is also no body code, however the directive *HARDWARE_BODY_CALL* instructs the compiler to assign the declaration to particular hardware. In CPCore, the LCD_CH block has a unique identifier 4 (ID:0004). After the declaration is entered, an instance of the block can be created and invoked from a program as any other function block.

However, since hardware blocks usually control peripherals, their usage in parallel environment is somewhat limited due to potential conflicts. Typically, such block cannot be concurrently executed by two or more tasks. This is called *mutual exclusion* and can be achieved in two ways.

Some hardware block calls from multiple tasks are queued internally by the CPCore and then executed sequentially. Task execution may be delayed due to queue processing, but the collision does not occur. This mechanism applies mostly to simple blocks executed in one-shot manner. LCD_CH and some flip-flops are examples of queued hardware blocks.

Assignment of hardware block to particular executor (virtual machine) during configuration of CPCore is another way of avoiding conflicts during the calls. In such arrangement, only that executor will be able to call the block. Other tasks cannot call the block directly, however a software solution can be implemented to provide access to the block functionality via a dedicated task. In CPCore this applies to UART and 1-wire interface handling blocks.

Fig. 4 shows CPDev *hardware configurer* window which allows to set up hardware blocks for CPCore controller. The upper area activates blocks related to standard peripheral services (UART, LCD, 1-wire interface) and a type conversion block. The lower part contains a list of IEC standard blocks implemented in hardware. Contrary to software-implemented blocks they are executed extra fast, so the overall performance of the algorithm is increased. In case of CPCore, one can use RS and SR flip-flops, counters, triggers and timers. Here, two instances of TON and TOF blocks are defined, for instance to be used in START_STOP diagram (Fig. 2). According to the settings, the hardware configurer generates appropriate libraries for the CPCore FPGA chip.

4.3. Mutual exclusion with semaphores

Sometimes hardware solutions described above are not sufficient to provide collision-free native block calls. The problem arises especially when a hardware block is called in the code of another function block. For example, DISPB is a conventional function block written in ST, used to print a string on LCD display. DISPB executes actual printing by calling LCD_CH hardware block for every character. Although DISPB can be used by any task, if printing loop is in progress in one task, the other tasks cannot get access to the display. Otherwise consistency of the display would be violated.

Multi-CPCore programmer can use *semaphores* for task synchronization and mutual exclusion. A semaphore is a global integer variable accessed from tasks by LOCK and UNLOCK functions. Unlocking increments semaphore value, while locking decrements it. When the semaphore value is zero, the locking task is suspended until one of other tasks unlocks the semaphore. Semaphores prohibit tasks from running critical part of code, when that part is currently executed.

To provide mutual exclusion upon DISPB call, a semaphore should be created, i.e. VSEM below, with initial value 1. Then the following code protects the printing loop from re-entry.

```
LOCK(VSEM);
FOR i:=1 TO 16 DO       (* 16 chars are kept *)
 LCD_PUTCHAR(c:=str[i]) (* in str array *)
END_FOR
UNLOCK(VSEM);
```

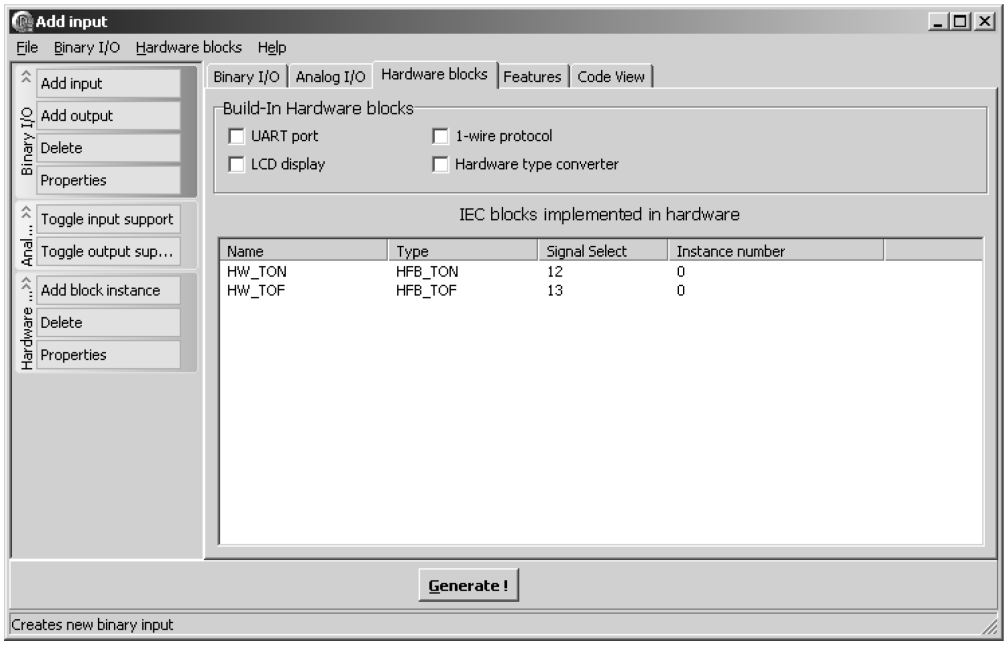


Figure 4. Configuring hardware blocks for the CPCore controller

When a task enters the code, it locks the semaphore by decrementing it. So the printing begins. When another task tries to enter the code while printing, the semaphore value is zero, so that task will be suspended and queued. After the first task unlocks the semaphore, one of the queued tasks is resumed and can execute another printing.

5. Hardware structure

Simplified block diagram of the multi-CPCore controller implemented in FPGA is shown in Fig. 5. The design is based on symmetric multiprocessor architecture [6]. Multiple hardware machine cores in the center of Fig. 5 are hardware machines (executors) which run concurrently. Implementation of the hardware machine core has been described in [1]. In the actual CPCore they are additionally equipped with a floating point unit [7] (useful for continuous control). Each core, being in fact a specialized microprocessor with dedicated program and data memories, is responsible for execution of a single task.

Apart from the machine cores, there is also another unit, called *initiator core*. It is a simple processor responsible for initialization of selected locations in global variable memory. The initiator core is triggered on power up or after a new configuration is loaded into the controller. Only after completing the initialization, other cores begin to work.

As described in Sec. 4.1, communication between machine cores is implemented through common global memory. Collision free access to that memory is provided by the global memory and I/O access arbiter block (Fig. 5). Handshaking protocol is applied for data transfer between the cores and the arbiter. This is a part of the process image mechanism described in Sec. 4.1.

A core, which needs to access the global memory, sets a request signal. The arbiter successively analyses request occurring at its ports and grants access to the global memory. Granting the access to particular core is confirmed by acknowledgment signal. At the end of data transfer, the core releases the request. In response, the arbiter releases the acknowledgment and begins scanning other ports for request signals.

The global memory address space is divided into two ranges. The lower range, starting from address 0 up to a configured value, is reserved for addressing input/output devices (peripherals), such as digital input and output modules. The upper range maps physical synchronous RAM

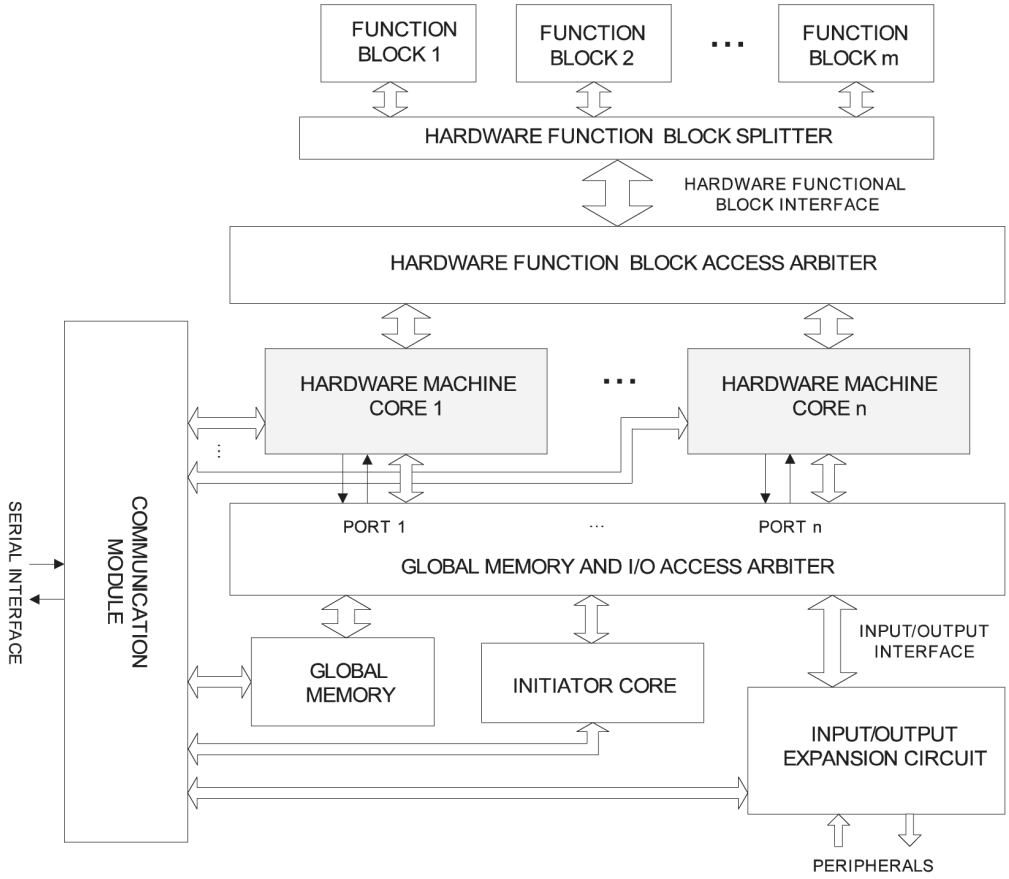


Figure 5. Simplified block diagram of multi-CPCore FPGA implementation

memory, used to hold global variables. Hardware machine cores access the input/output devices in the same way as accessing the global memory (i.e. via the arbiter). Additional expansion circuit is needed to connect peripherals to input/output interface of the arbiter block.

The CPCore FPGA-based controller has been equipped with facility to integrate hardware function blocks. Special mechanism for data transfer between such blocks and executing machines has been designed and implemented. As described in Sec. 4.2, there are two options for connecting hardware blocks. The first one assumes that each core has its own set of hardware blocks. The second one shown in Fig. 5 implements the idea of sharing hardware blocks among executing cores. In this case, each core can invoke any of available blocks. This capability requires the use of arbiter block, which ensures collision-free access to the blocks. The hardware function block access arbiter (Fig. 5) operates similarly to the global memory access arbiter. However, the hardware function block splitter, which consists mainly of a set of multiplexers, is additionally required to connect the arbiter to hardware blocks.

Communication module is an important component of the CPCore platfrom [8]. It provides data transfers with CPDev environment, especially for on-line monitoring and commissioning purposes. The communication module ensures full read and write access to the global variable area, as well as to program and data memories of each hardware machine core. It also performs special functions like in-circuit debugging.

A prototype controller with multi-CPCore technology shown earlier in Fig. 1 consists of eight executing machine cores what allows for execution of up to eight concurrent control tasks. Four hardware function blocks (UART, alphanumeric LCD, 1-wire bus, hardware type conversion) are available. The prototype has been implemented in Xilinx Spartan-6 FPGA

XC6SLX100-3FGG676, with the main circuit board described in [9]. The chip employs 31099 6-input LUTs (49% of all available in FPGA) and 13513 slice registers (10%).

6. Summary

Multi-CPCore FPGA-based PLC execution platform has been described. The platform integrates several hardware machines, each handling one control task. FPGA implementation results in short execution times, if compared to standard microcontroller-based solutions. The CPCore tasks run concurrently and independently, as a set of virtual PLCs. Each task can be set up with its own cycle time. As a result, CPCore controller can handle different applications at the same time. For example, it can execute fast logic control concurrently with continuous control, and also handle HMI operating panel. Such functionalities are available in industrial computers, but CPCore technology can be applied for much smaller devices.

Multi-CPCore is programmed and configured in CPDev engineering environment, compatible with IEC 61131-3 standard. The tasks are composed of programs written in ST, FBD or IL languages. In addition to standard libraries, a library of hardware blocks is available to access peripherals and speed up operations. Task synchronization mechanisms have been developed to eliminate conflicts while accessing hardware resources in parallel environment.

References

[1] Z. Hajduk, B. Trybus, and J. Sadolewski, "Hardware implementation of virtual machine for programmable controllers," in *Metody wytwarzania i zastosowania systemów czasu rzeczywistego*, L. Trybus and S. Samolej, Eds. Warszawa: Wydawnictwa Komunikacji i Łączności, 2010, ch. Chapter 5, pp. 333–342, (in Polish).

[2] *IEC 61131-3 standard: Programmable Controllers – Part 3. Programming Languages*, IEC Std., 2003.

[3] D. Rzońca, J. Sadolewski, A. Stec, Z. Świder, B. Trybus, and L. Trybus, "Open environment for programming small controllers according to IEC 61131-3 standard," *Scalable Computing: Practice and Experience*, vol. Volume 10, no. 3, pp. 325–336, 2009.

[4] D. Gawali and V. Sharma, "FPGA Based Micro-PLC Design Approach," *Advances in Computing, Control, and Telecommunication Technologies, International Conference on*, vol. 0, pp. 660–663, 2009.

[5] M. Adamski and J. L. Monteiro, "From interpreted Petri net specification to reprogrammable logic controller design," in *Proc. IEEE Int. Symp. Industrial Electronics (ISIE 2000)*, vol. 1, 2000, pp. 13–19.

[6] P. Huerta, J. Castillo, C. Pedraza, J. Cano, and J. I. Martinez, "Symmetric multiprocessor systems on FPGA," in *IEEE Int. Conf. on Reconfigurable Computing and FPGAs*. ReConFig '09, 2009, pp. 279–283.

[7] Z. Hajduk, "Floating-point arithmetic unit for a hardware virtual machine," *Pomiary Automatyka Kontrola*, vol. 57, no. 1, pp. 82–85, 2011, (in Polish).

[8] ——, "Communication module for a hardware implemented virtual machine," *Elektronika – konstrukcje, technologie, zastosowania*, no. 5, 2011, (in Polish).

[9] ——, "PLC controller prototype with a hardware virtual machine," *Elektronika – konstrukcje, technologie, zastosowania*, no. 4, pp. 114–118, 2011, (in Polish).

e-Informatica Software Engineering Journal (http://www.e-informatyka.pl/wiki/e-Informatica) is an international journal that concerns theoretical and practical issues pertaining development of software systems, and focuses on experimentation in software engineering.

The purpose of e-Informatica is to publish original and significant results in all areas of software engineering research.

The scope of e-Informatica includes methodologies, practices, architectures, technologies and tools used in processes along the software development lifecycle, but particular stress is laid on empirical evaluation.

Topics of interest include, but are not restricted to:

- Software requirements engineering and modeling
- Software architectures and design
- Software components and reuse
- Software testing, analysis and verification
- Agile software development methodologies and practices
- Model driven development
- Software quality
- Software measurement and metrics
- Reverse engineering and software maintenance
- Empirical and experimental studies in software engineering
- Evidence based software engineering
- Systematic reviews
- Object-oriented software development
- Aspect-oriented software development
- Software tools, containers, frameworks and development environments
- Formal methods in Software Engineering.
- Internet software systems development
- Dependability of software systems
- Human-computer interface
- AI and knowledge based software engineering
- Project management

The submissions will be accepted for publication on the base of positive reviews done by international Editorial Board (http://www.e-informatyka.pl/wiki/e-Informatica_-_Editorial_Board) and external reviewers. English is the only accepted publication language. To submit an article please enter our online paper submission site.

Subsequent issues of the journal will appear continuously according to the reviewed and accepted submissions.

http://www.e-informatyka.pl/wiki/e-Informatica

e-Informatica

ISSN 1897-7979