

# Automotive Industry: Process Focus Only Insufficient?

Dr. Hans Sassenburg  
SE-CURE AG  
Lenk, Switzerland  
[hsassenburg@se-cure.ch](mailto:hsassenburg@se-cure.ch)

Dr. Lucian Voinea  
SolidSource BV  
Eindhoven, Netherlands  
[lucian.voinea@solidsource.nl](mailto:lucian.voinea@solidsource.nl)

## ABSTRACT

Software is a major worldwide industry. Software pervades a multitude of products and is an important corporate asset, with demand still increasing, especially in the automotive industry. Without software, car manufacturers could not survive in the current marketplace, and both the impact of software on everybody's life and our dependence on software is rapidly increasing. It cannot, however, be denied that software engineering is still a discipline with much potential for improvement. Software projects are characterized by schedule and budget overruns, and the delivery of unreliable, and difficult to maintain, software products. Existing improvement strategies mainly focus on improving the operational efficiency. There is however no indication that any improvement strategy can result in the performance improvements needed, matching the exponential growth of the variety and size of software products. One can still speak of a software crisis, in the sense that the turning point has probably not yet been reached. Typical car manufacturers are likely to become increasingly less predictable in terms of cost, quality and time-to-market. Without adopting a paradigm shift towards the adoption of formal methods, the most important challenge for this industry in the future will no longer be the satisfaction of new needs, but the reparation of damage by the software of today.

## Keywords

Automotive industry, software crisis, reliability, formal methods.

## 1. INTRODUCTION

Software is a major worldwide industry, its demand increasing exponentially [4] [13]. Software pervades a multitude of products, in social, business and military human-machine systems. It includes information (technology) systems, developed for gathering, processing, storing, retrieval and manipulation of information, to meet organizational needs, and commercially-developed software products, sold to one, or more, customers or end-users.

The increasing demand for application software brings with it an increase in our dependence on software [23]:

- Software-based systems replace older technologies in safety or mission-critical applications.
- Software moves from an auxiliary to a primary role in providing critical services.
- Software becomes the only way of performing some function; not perceived as critical but whose failure would deeply affect individuals or groups.

- Software-provided services increasingly become an accepted part of everyday life, without any special scrutiny.
- Software-based systems increasingly integrate and interact, often without effective human control.

Carr criticizes however today's worship of software [9]. Although he recognizes that software has deeply transformed today's society, he claims that it no longer offers a competitive advantage to organizations. Using examples of electric power production and trains he shows that software, as a proprietary technology, will no longer generate a competitive advantage. Instead, it can be expected that the usage of software will become standardized and no organization will benefit from purchasing and maintaining its own software solutions. Although this paradigm shift might be true, it does not imply that the production and consumption of software solutions will decrease. Taking the example of electric power production, used by Carr, it is obvious that, even after standardization instead of proprietary solutions, production and consumption of electric power has increased exponentially. The same holds for software, nowadays spreading from computers to, for example, the engines of automobiles to robots in factories to X-ray machines in hospitals.

## 2. SOFTWARE SIZE

It is often reported that the size of software products is growing at an exponential rate following *Moore's Law* [1] [7] [11] [37]. The observation, made in 1965 by Gordon Moore, co-founder of Intel, noted that the number of transistors per square inch on integrated circuits had doubled every year since the integrated circuit was invented. Moore predicted this trend would continue for the foreseeable future. In subsequent years, the pace slowed a bit, but data density has doubled approximately every 18 months, and this is the current definition of Moore's Law. Expressed as 'a doubling every 18 months' or 'ten times every 5 years', Moore's law is applied in other technology-related disciplines suggesting the phenomenal progress of technology development in recent years. Expressed on a shorter timescale, Moore's law equates to an average growth of over 1% a week.

Software size can be described using different attributes. Fenton and Pfleeger use three attributes [10]:

- *Length*; the physical size of the product. In general, the code length is the easiest to measure and is normally expressed in terms of lines of code.
- *Functionality*; amount of functionality delivered to end-users [in function or object points].
- *Complexity*. This attribute is interpreted in different ways: problem complexity (complexity of the underlying problem); algorithmic complexity (complexity of the algorithm

implemented to solve the problem); structural complexity (structure of the software); and cognitive complexity (ease of understanding the software).

It is assumed that increasing the length, or functionality, of software will, in general, also lead to an increase of complexity, whatever perspective is taken. According to Broy [7], modern automobiles contain up to 10 million lines of software, distributed across 80 separate controllers and 5 bus systems. According to Yang [37], the size of a typical embedded system is estimated to be 32 million lines of software in the year 2010.

### 3. DEFECT DENSITIES

In software, the narrowest sense of product quality is commonly recognized as a lack of defects or ‘bugs’ in the product. Using this viewpoint, or scope, three important measures of software quality are:

- *Defect potential*, defined as the number of injected defects in software systems, per size attribute.
- *Defect removal efficiency*, defined as the percentage of injected defects found and removed before releasing the software to intended users.
- *Defect density*, defined as the number of released defects in the software, per size attribute.

A study by Jones [17] reveals that the defect potential, the defect removal efficiency, and the defect density at release time depend on the software size. From Table 1 it follows that as software grows, defect potential increases and defect removal efficiencies decrease. The defect density at release time increases and more defects are released to the end-user(s) of the software product. Larger software size increases the complexity of software and thereby the likelihood that more defects (both absolute and relative) will be injected. For testing, a larger software size has two consequences [12]:

- The number of tests required achieving a given level of test coverage increases exponentially with software size.
- The time to find and remove a defect first increases linearly and then grows exponentially with software size.

As software size grows, to just maintain existing levels of released defect densities, software manufacturers would have to exponentially improve their defect potentials and removal efficiencies. The situation could be even worse. When software grows, more functionality is offered to the end-users. Assuming they use the enhanced functionality, end-users are exposed to even more defects. Maintaining the existing level of released defect density is not enough and this should be further decreased.

Combining these results leads to the following two conclusions:

1. The size and complexity of software and the amount and variety of software products are growing exponentially.
2. The increased dependence of society on software means end-users will be exposed to more defects when software manufacturers are unable to exponentially improve their defect potentials and removal efficiencies.

**Table 1. Software Size versus Defect Potential, Removal Efficiency, and Density [17].**

Size in function points	Defect Potential (development)	Defect Removal Efficiency	Defect Density (released)
	Defect potential and density expressed in terms of defects per function point		
1	1.85	95%	0.09
10	2.45	92%	0.20
100	3.68	90%	0.37
1'000	5.00	85%	0.75
10'000	7.60	78%	1.67
10'0000	9.55	75%	2.39
<b>Average</b>	<b>5.02</b>	<b>86%</b>	<b>0.91</b>

There is an increasing volume of evidence illustrating these conclusions, and an increasing number of software-related accidents are being reported. Gage and McGormick have published an overview of some known fatal software-related accidents [11].

Leveson published a collection of well-researched accidents along with brief descriptions of industry-specific approaches to safety [22]. Accidents are described in the fields of medical devices, aerospace, the chemical industry and nuclear power. Some of the most widely cited software related accidents in safety-critical systems involved a computerized radiation therapy machine called the Therac-25. Between June 1985 and January 1987, six known accidents involved massive overdoses by the Therac-25 with resultant deaths and serious injuries. They have been described as the worst series of radiation accidents in the 35-year history of medical accelerators [21].

### 4. IMPROVEMENT STRATEGIES

Do the conclusions in the previous section mean there is still a software crisis? One could claim the software crisis is dead, in the sense that the term ‘crisis’ refers to a turning point and we have passed this point. In this case, a valid question is whether there has indeed been a turning point. To answer this question, the gains in software productivity and the most important improvement strategies of the last decades are first discussed.

#### 4.1 Software Productivity

Software productivity is often expressed as a ratio of the amount of source code statements produced for some unit of time, such as lines of code per hour. Scacchi claims however that due to the number and diversity of variables influencing software productivity this is an oversimplification [31]. In the same survey, he further concludes existing software productivity measurement studies are fundamentally inadequate, and potentially misleading. Combining different studies, Scacchi identifies a list of productivity drivers related to the development environment, the product and the project staff. Putnam and Myers follow this approach and define the so-called *Productivity Index*, combining different factors and making the argument that it is objective, measurable and capable of being compared on a numeric scale

[27]. It is a macro-measure of the total development environment. Putnam studies the trend in this Productivity Index and concludes that it has increased linearly over the last decade [28]. Reifer confirms this, reporting an average linear productivity increase of 8 to 12 percent a year [29]. In his study, productivity is viewed from a quality point of view by normalizing it to the quality of a software product when released to its intended customers or end-users.

**Table 2. Software Engineering Effort by Task [17].**

Activities	Workdays	Percent
Testing and defect repairs	120	61%
Time on cancelled projects	30	15%
Productive time on projects	47	24%
<b>Total</b>	<b>197</b>	<b>100%</b>

For software productivity, another relevant issue is the efficiency of developing software products. In general, effort is being wasted for two main reasons: cancelled projects (the software product is never released) and software repair (testing and defect repairs). Jones finds that 15% of the global software workforce is involved in projects that will never be deployed [17]. Further, that 61% of software developers' time is spent on software repair, including testing. This confirms the observation of Boehm and Basili that some 40-50% of the effort on current software projects is spent on avoidable rework, excluding testing [5].

In Figure Table 2, the work pattern for a typical software engineer, after vacations, training and sick days, is given, with all activities other than time spent actually working on software projects factored out.

In addition to developing projects that never see the light of day, software personnel are involved in defect removal from projects eventually completed, and defect repairs during routine maintenance of software products already released.

Assuming software productivity is growing linearly, and not exponentially, a possible solution is to increase the amount of software produced per unit of time, by allowing more people to work in parallel. However, this solution provides limited results. It is only in the later project phases that work can be distributed effectively among more software engineers. Increasing the project team size will lead to higher overheads, thus likely to reduce the average productivity level [6].

It is concluded that, until now, software productivity has mostly been increasing linearly, rather than exponentially, with most development effort still taken up by defect repair, and projects that will be prematurely terminated. This linear productivity increase is insufficient to cope with the exponentially increasing software size. The result is a capability gap between the demand for ever larger and more complex (embedded) software systems and the average development organization's productivity. This gap is the source of enormous tension in the market.

## 4.2 Traditional Improvements

Nearly two decades ago, Boehm identified a number of strategies for improving software productivity: get the best from people, make development steps more efficient, eliminate development

steps, eliminate rework, build simpler products and re-use components [3]. Re-use especially has been regarded as a high-potential solution [25], but overall results have been disappointing [32]. In 2001, Boehm and Basili published a list of the most important factors in reducing defect injection and removal rates [5].

During the last decades, many software development organizations initiated software process improvement programs. The intention of these initiatives is to improve the software manufacturer's performance by reaching, for example, the higher levels of process maturity models. A widely-accepted reference model is the *Capability Maturity Model Integration* or *CMMI* [32]. A well-known standard is *ISO/IEC 15504* describing the requirements for conducting assessments and making process capability profiles [15]. The process descriptions can be found in the *ISO/IEC 12207* standard [16] and in 2005, the derived automotive-specific standard *Automotive SPICE* was released [34].

The Software Engineering Institute twice yearly publishes a *Maturity Profile Update*. These profiles list the percentage of officially assessed software manufacturers performing at each maturity level using CMMI as the reference model. Over the last decade, improvements have been reported. The situation at the end of 2006 revealed that most organizations involved in the study are performing at the lowest and second-lowest maturity level (35.0%), with basic project management practices in place [34]. However, it is assumed the assessment of all software manufacturers worldwide would show a more dramatic picture, as the results reported to the Software Engineering Institute only include officially assessed software development organizations, namely organizations that are willing, or forced, to be assessed.

## 4.3 Improvement Potential

Jones compares the best software manufacturer organizations with the worst ones, to illustrate the scope for improvement [17].

**Table 3. Defect Potential and Removal Efficiency.**

Task	Leading	Average	Lagging
	(defects per function point)		
Requirements	0.55	1.00	1.45
Design	0.75	1.25	1.90
Coding	1.00	1.75	2.35
User manuals	0.40	0.60	0.75
Bad fixes	0.10	0.40	0.85
<b>Total</b>	<b>2.80</b>	<b>5.00</b>	<b>7.30</b>
Removal %	95%	85%	75%
Delivered	0.14	0.75	1.83

In Table 3, a range of results for lagging, average and leading software projects is presented. These results suggest there is ample room for improvement. The defect injection rate in average organizations is almost twice that found in the best organizations. Further, by improvements to both the defect potentials and the defect removal efficiencies, there is room for a reduction by a

factor of five in the number of defects actually delivered by average organizations.

#### 4.4 Reducing Maintenance Cost

Maintenance is the most expensive stage in the life time of a software product. Industry surveys estimate that around 80% of the total development costs of software are incurred after the first release of the product for covering its maintenance (e.g., bug removal and feature addition). Nowadays the situation is aggravated by two sustained trends in the industry: the decreasing time-to-market and the increasing software size and complexity. Both trends cause a decrease in the quality of the product. This translates into more bugs and, consequently, higher maintenance costs. Additionally, the increase in software size and complexity exacerbates another problem of the industry: “the software legacy crisis”. Increasingly more software is maintained by other developers than the initial ones and with little or no access to the initial documentation. These developers are forced to spend valuable time on trying to understand the software, before being able to actually maintain it. This drives costs significantly higher. One of the most promising approaches for addressing the above problems is automatic software analysis. IDC forecasts strong continuing growth in the Automated Software Quality segment. Expected revenue approaches the magic \$2 billion mark around 2010 [14]. This approach offers a number of important advantages. Firstly, it can improve software quality by monitoring/enforcing best practices on a product base - as opposed to process. Secondly, it facilitates software understanding by enabling source code based overviews at varying levels of detail. Thirdly, it provides managers with factual (i.e., source code based) information enabling them to take informed decisions. Finally, being automatic, it is very fast, accurate and reduces the personnel costs.

### 5. SOFTWARE CRISIS?

The term ‘software crisis’ has been used since the late 1960s to describe those recurring system development problems in which software development cause the entire system to be late, over budget, not responsive to the user and/or customer requirements, and difficult to use, maintain and enhance. Royce emphasized this situation [30]: *“The construction of new software that is both pleasing to the buyer/user and without latent errors is an unexpectedly hard problem. It is perhaps the most difficult problem in engineering today, and has been recognized as such for more than 15 years. It is often referred to as the ‘software crisis’. It has become the longest continuing crisis in the engineering world, and it continues unabated.”* It could be argued that the software crisis is dead in the sense that the software industry has passed the turning point, as under the right conditions software development can be managed. The positive side is that software productivity is increasing, some software manufacturers have succeeded in improving their maturity levels, and there is room for improvement, when comparing leading and average software manufacturer organizations. It could also be argued that one can still speak of a software crisis, in the sense that the turning point has probably not yet been reached. There is no indication that any improvement strategy can result in the performance improvements needed. It is questionable whether the effects of any improvement strategy can match the exponential

growth of the variety and size of software products. It is likely that, in the future, more software products will be released with higher defect densities, with the most likely consequence that end-users will be confronted with more defects. Put differently, the chronic character of software development problems is unlikely to be resolved in the future. The question that may arise is whether a possible solution to reduce or even eliminate the chronic software development problems is developing less and simpler software products. The study from Jones, discussed in section 3, reveals that as software size shrinks, defect potential decreases and defect removal efficiencies increase [18]. In practice, it will be difficult to limit the development of new software products, however each software manufacturer will undoubtedly benefit from trying to minimize product size by implementing only those requirements for which a clear customer/user demand exists.

If the software industry is unable to find easy-to-implement improvement strategies, the typical software manufacturer organization is likely to become increasingly less predictable in terms of cost, quality and time-to-market, a trend confirmed by studies of the Standish Group. The latest Chaos report of the Standish Group reveals the software industry is losing ground [36]. Only 28% of software projects succeed these days, down from 34% a year or two ago. Outright failures (projects cancelled before completion) are up from 15% to 18%. The remaining 51% of software projects are seriously late, over budget and lacking features previously expected. In markets with increasing competition and smaller market windows, software manufacturers might experience an increasing pressure to release software products prematurely, disregarding the total life-cycle effects. In this case, uncertainties are:

- *Unknown product behaviour.* It is difficult, if not impossible, to guarantee customers/end-users the exact functional and non-functional requirements of the software product. This may lead to dissatisfied customers/end-users and to unforeseen, even potentially dangerous, situations. Apart from the fact that people’s lives may be at risk, such situations can have an enormous financial impact on the software manufacturer.
- *Unknown operational maintenance cost.* The post-release or maintenance cost of the software may become unexpectedly high. If the exact status of the software with its documentation is unknown, a software manufacturer may be confronted with high maintenance costs for correcting failures. Future adaptive and perfective maintenance activities may be severely hampered.
- *Reputation.* Time-to-market in the automotive industry is extremely important. Missing a model launch date is hardly a viable or economic option, with models being specified years in advance and associated enormous marketing budgets. On the other hand, introducing a car with safety-critical problems is extremely risky. Automotive manufacturers are in most countries by law liable for the failure of their products. Further, if it should become necessary to recall a model due to a software failure, the associated costs are enormous. This market entry trade-off between time-to-market and reliability is expected to become more complex in the (near) future. Carchia summarizes a classic example where ethical aspects were involved, as in the release of the Ford Pinto [8]. Design flaws involving the fuel system were not properly corrected. Ford used a cost-benefit analysis, placing a price on human life, and decided that it was not economically justifiable to fix the known problem

immediately. The result was 27 people killed and Ford ended up paying millions of dollars in legal settlements to accident victims, recalling cars to install a part to fix the problem, and dealing with a tarnished reputation.

## 6. AUTOMOTIVE INDUSTRY

Car manufacturers are reshaping their products from primarily mechanical devices into digitally controlled systems. The number of electronic control systems in every car, from low end to luxury models, is increasing rapidly. Car manufacturers are moving towards a systems oriented approach in which a limited number of suppliers supply fully assembled and tested modular systems. Specific areas of concern are:

- *Efficiency.* Both pre-release development cost and post-release operational cost are under ongoing pressure. The potential to improve operational efficiency, developing the product right the first time, remains an attractive proposition. Especially for suppliers, faced with delivering more functionality for less money, improving operational efficiency has a high priority.
- *Effectiveness.* Embedded architectures in cars, combining hardware and software, are characterized by increasing network complexity. Also, an increasing tendency towards parallelism and distribution in embedded automotive systems can be observed. This highly contributes to increased complexity, leading to possibilities for deadlocks, live-locks, race conditions, and other behavioral problems.

The presence of these areas of concern may have a dramatic impact on a car manufacturer's market position. Launching a new model too late might severely undermine its market position, releasing a new model prematurely might lead to recalls and warranty, or even liability, problems. This is illustrated by figures from this industry; transformed from a mechanically-oriented industry to an electronic and software-oriented industry. In Figure 1, an overview is given of the number of officially reported recalls in the German and UK automotive industry [2] [19], showing a constant increase with an increasing percentage of recalls from the hardware/software area.

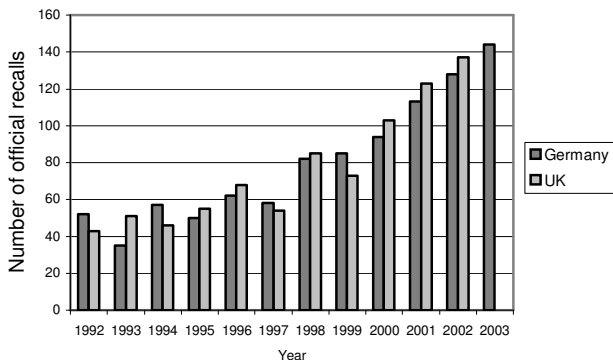


Figure 1. Recalls in Automotive Industry [2] [18].

Including 'silent' recalls, the ones not officially reported, would probably reveal a more dramatic picture.

Another study in the automotive industry, conducted by McKinsey, concerns the financial impact of quality problems on profit margins [25]. As in Figure 2, the economic significance of these problems is high.

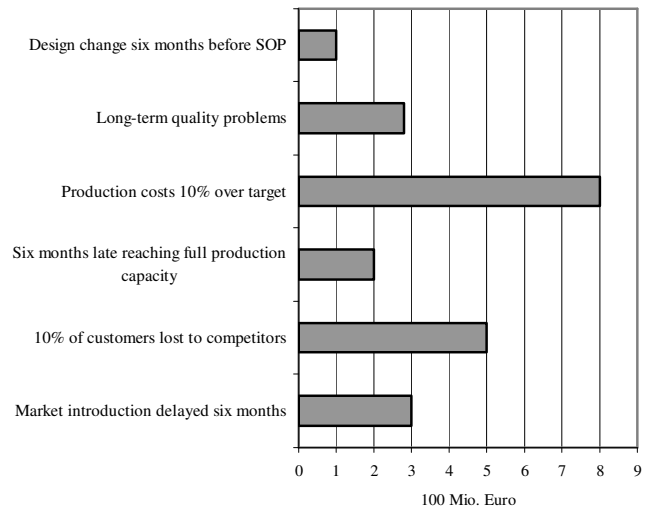


Figure 2. Quality Problems in Automotive Industry [25].

Finally, when looking into the near future, it is expected that the worldwide volume of automotive software and hardware (production costs) will grow significantly [20] [24], as illustrated in Figure 3.

From these figures, it is concluded that automotive manufacturers are facing uncertain times due to the unpredictability of their development process and resultant products, while software is expected to play an increasingly important role in new innovations, production costs, lead-time of product developments and released product quality.

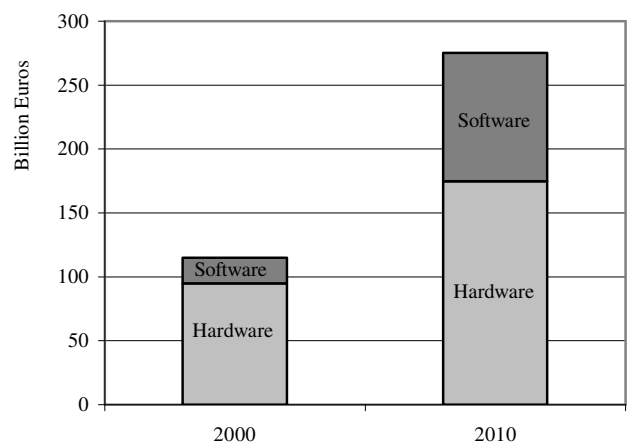


Figure 3. Expected Growth in Market Volume [20] [24].

## 7. PARADIGM SHIFT?

### 7.1 Software Engineering and Mathematics

What distinguishes engineering from craftsmanship? One major characteristic is predictability of outcome. All branches of engineering, except software engineering, routinely employ mathematics to verify specifications and designs before starting implementation. Most software is not developed this way; except in safety critical domains where it may be mandated, mathematics are not routinely employed when specifying or designing software systems. As a consequence, we have only informal, review based methods to examine software designs and specifications for completion and correctness before investing in programming. We have no way of verifying designs for correctness in any formal and complete way. Software development methods rely almost exclusively on testing the implementation in order to determine the correctness of specifications and designs. When testing software, we must detect and remove specification errors, design errors and implementation errors. As a branch of engineering, software engineering is unique in this approach. Over the past 30 years or so, a number of formal methods have been developed in order to apply mathematical techniques to software development. Yet, formal methods are seldom found in the industrial software development organizations. Furthermore, many of those who have tried formal methods are not keen to repeat the experience. The two principle reasons for this are reviewed.

#### *Informal requirements versus formal specifications*

A formal method must start with a formal specification; however, conventional practice in industry starts with compiling an informal requirements specification. This is typically a substantial document, running to hundreds or thousands of pages describing the required behavior and characteristics of the software. It is a key part of the business case supporting the development and is written in business and domain specific terms. Furthermore, this document is not static; managing changes in requirements during software development is simply an industrial necessity. So the requirements specification is a living document that evolves over time. To apply a formal method we must start by developing a formal specification from the informal requirements specification and during development we must be able to keep the formal specification synchronized with changes to the informal specification. Having done this, how do we verify that the formal specification describes the same system as the informal requirements specification? Formal specifications are typically constructed in a specialized mathematical language and require the use of specialists with an extensive mathematical background and expertise in the method. In practice, it is rarely if ever the case that the business analysts and domain experts have that expertise or experience. In short, the only people with the domain knowledge necessary to validate the formal specification cannot do so because they do not understand it.

#### *Disunity between abstract model and software system*

Formal verification techniques are typically applied to reason about some abstract representation or model of parts or all of the actual system under development. Abstract models are, by definition, an abstraction of the actual system being developed; as a result, the verification draws conclusions regarding the 'correctness' (however that is defined) of the abstraction as opposed to the actual system's design or implementation.

Regardless of how effective a formal verification technique is, if we cannot define a clear mapping strategy between the abstract formal models being analyzed and the actual system being developed, the benefits of the formal verification will remain limited. The lack of such a clearly defined relationship between the two poses a fundamental barrier to introducing formal methods in industry. By the very nature of the applications that need a more formal approach, the systems being developed are too complex to reason about confidently by hand (hence the need for formal models). The task of verifying that the formal models truly reflect the actual system is similarly complex. By the very nature of the problem domain we are considering (complex automotive software), the specialist knowledge required by the domain experts is extensive and application specific; to propose to retrain them at the start of a project to reach the necessary level of expertise in the chosen method is not practical. It is necessary to reconcile the gap between the formal analysis being done on the abstract models and the software development process, such that the necessary feedback can flow accurately between the two in both directions.

## 8. CONCLUSIONS

Despite the increasing demand and complexity of software, existing improvement strategies mainly focus on improving the operational efficiency. There is however no indication that any improvement strategy can result in the performance improvements needed, matching the exponential growth of the variety and size of software products as in the automotive industry. One can still speak of a software crisis, in the sense that the turning point has probably not yet been reached. The typical car manufacturer is likely to become increasingly less predictable in terms of cost, quality and time-to-market. Traditional testing-centered software development, with its emphasis on defect detection and removal, is failing in practice to deliver correctly functioning business-critical and untestable software on time and with required quality. Automatic source code analysis during development and maintenance is a very fact and accurate solution to support software manufacturers improving software quality and facilitating software understanding. In addition, it provides managers with source code based information enabling them to take informed decisions. On top of that, there is the need to focus on defect prevention by adopting an alternative and more formal approach embodying the following two principles: (i) business-critical and untestable software must be based on designs that are verifiably correct before a single line of code is written; and (ii) software architects and designers must limit themselves to those designs and patterns that can be verified correct using the currently available tools. Without this paradigm shift, the most important challenge for software development in the automotive industry in the future will no longer be the satisfaction of new needs, but the repair of damage by the software of today.

## 9. REFERENCES

- [1] Asseldonk, W. van, Predictability in Consumer Products. *Proceedings of B&C Conference "Making Software Work"*, September 2004.
- [2] Bates, H., et al., Motor Vehicle Recalls: Trends, Patterns, and Emerging Issues. *ESRC Centre for Business Research, University of Cambridge, working paper no. 295*, 2004.

- [3] Boehm, B.W., Improving Software Productivity. *IEEE Computer*, 20, 8, 1987, 43-58.
- [4] Boehm, B.W., Sullivan, K.J., *Software Economics: A Roadmap*. ACM Press, 2000.
- [5] Boehm, B.W., Basili, V.R., Software Defect Reduction Top-10 List. *IEEE Computer*, 34, 1, 2001, 135-137.
- [6] Brooks, F.R., *The Mythical Man Month*. Addison-Wesley, 1975.
- [7] Broy, M., Challenges in Automotive Software Engineering: From Demands to Solutions. *Proceedings of the EmSys Summer School at the University of Salzburg*, June 30 - July 2, 2003.
- [8] Carchia, M., Profits and Business Models. *Carnegie Mellon University, 18-849b, Dependable Embedded Systems*, Spring 1999.
- [9] Carr, N.G., *IT doesn't matter*. Harvard Business Review, May 2003.
- [10] Fenton, N.E., Pfleeger, S.L., *Software Metrics: A Rigorous & Practical Approach*. PWS Publishing Company, 1997.
- [11] Gage, D., McCormick, J., *Why Software Quality Matters*. Baseline, the Project Management Center (March 4<sup>th</sup>), 2004.
- [12] Humphrey, W.S., The Future of Software Engineering: I. *Column SEI*, 4, 1, 2001.
- [13] Humphrey, W.S., *Winning with Software: An Executive Strategy*. Addison-Wesley, 2002.
- [14] IDC, *Worldwide Automated Software Quality Tools, 2006-2010 Forecast*.
- [15] ISO, *ISO/IEC 15504-2:2003 Information Technology - Process Assessment - Part 2: Performing an assessment*. International Organization for Standardization, 2003.
- [16] ISO, *ISO/IEC 12207:1995/Amd 2:2004*. International Organization for Standardization, 2004.
- [17] Jones, C., *The Impact of Poor Quality and Canceled Projects on the Software Labor Shortage*. Technical Report, Software Productivity Research, Inc., 1998.
- [18] Jones, C., *Software Quality In 2002: A Survey Of The State Of The Art*. Technical Report Software Productivity Research, Inc., 2002.
- [19] Lederer, D., et al., Automotive Systems Engineering - The Solution for Complex Technical Challenges. 5. *Internationales Stuttgarter Symposium "Kraftfahrwesen und Verbrennungsmotoren"*, 2003.
- [20] Leveson, N.G., Turner, C.S., An Investigation of the Therac-25 Accidents. *IEEE Computer*, 26, 7, 993, 18-41.
- [21] Leveson, N.G., *Safeware: System Safety and Computers*. Addison-Wesley, 1994.
- [22] KBA, *Jahresbericht 2003/2004*. Kraftfahrt-Bundesamt, Flensburg (in German), 2003.
- [23] Littlewood, B., Strigini, L., Software Reliability and Dependability: a Roadmap. *Proceedings of the 22<sup>nd</sup> International Conference on Software Engineering*, 2000.
- [24] Mercer Management Consulting, *Automobiltechnologie 2010*. Study from HypoVereinsbank and Mercer Management Consulting (in German), 2002.
- [25] McKinsey, *Quality Gates: Successfully Shaping the Product Development Process*. 2001.
- [26] Poulin, J.S., et al., The business case for software reuse. *IBM Systems Journal*, 32, 4, 1993.
- [27] Putnam, L.H., Myers, W., *Measures for Excellence: Reliable Software On Time Within Budget*. Yourdon Press Computing Series, 1992.
- [28] Putnam, L.H., *Linking the QSM Productivity Index to the SEI Maturity Level*. Quantitative Software Management, Inc., 2000.
- [29] Reifer, D., Industry Software Cost, Quality and Productivity Benchmarks. *The DoD Software Tech News*, 7, 2, 2004.
- [30] Royce, W., Current Problems. In "Aerospace Software Engineering", Anderson, C., Dorman, M., (Eds), Washington, D.C.: American Institute of Aeronautics and Astronautics, 1991, 5-15.
- [31] Scacchi, W., Understanding Software Productivity. In "Advances in Software Engineering and Knowledge Engineering", Hurley, D., (Ed.), 4, 1995, 37-70.
- [32] Schmidt, D.C., *Why Software Reuse has Failed and How to Make It Work for You*. C++ Report Magazine (January), 1999.
- [33] Software Engineering Institute, CMMI for Software Engineering (Staged Representation). *Technical Report CMU/SEI-2002-TR-029*, 2002.
- [34] Software Engineering Institute, *Process Maturity Profile, CMMI<sup>®</sup> v1.1: 2007 Mid-Year Update (September 2007)*, [www.sei.cmu.edu/appraisal-program/profile/profile.html](http://www.sei.cmu.edu/appraisal-program/profile/profile.html), 2007.
- [35] SPICE User Group, Automotive SPICE™ Process Reference Model (PAM) RELEASE v4.3, 2007.
- [36] Standish Group, *CHAOS: A Recipe for Success*. Standish Group International, Inc., 2004.
- [37] Yang, F., ESP: A 10-Year Retrospective. *Proceedings of the Embedded Systems Programming Conference*, November, 1998.