BSSE System and Software Engineering

Cleared Direct To Target – Approaching the Target System at a Touch via Simulation

Dr. Rainer Gerlich, Dr. Ralf Gerlich

BSSE System and Software Engineering, Auf dem Ruhbuehl 181,
88090 Immenstaad, Germany, Phone +49/7545/91.12.58, Mobile +49/171/80.20.659, +49/178/76.06.129
Fax +49/7545/91.12.40, e-mail:Rainer.Gerlich@bsse.biz, Ralf.Gerlich@bsse.biz URL:http://www.bsse.biz

ABSTRACT:

This paper describes an improved process for development of software for critical systems fully bridging the gap between a specification and an executable target version by an automaton. The process covers the broad class of distributed and/or fault-tolerant and/or real-time systems, and meets the needs of critical systems. It has its roots in the space domain.

In the context of this process the role of simulation is different from how it is usually understood. Instead of establishing different model- and code-bases for simulation and the final target version, the process supports a smooth transition from simulation to the target version, continuously evolving the simulation version to the final version, based on a single model

The purpose of this paper is to discuss the process' relevance to avionics systems in terms of efficiency and flexibility of development, and reliability, availability, safety and dependability issues.

The automaton completely covers the transformation of a specification into executable code without any human intervention while putting emphasis on verification and validation. The short turn-around time allows an iterative and incremental approach starting with simulation and possibly emulation of hardware, then proceeding smoothly via refinement towards the final version executing on real hardware.

The process has been applied successfully to large and complex applications, amongst which is an experiment in use on-board of ISS.

As a result there is not only a significant impact on the efficiency in approaching the final version for the target, there are more benefits regarding fault identification and risk minimisation based on a number of means provided in the simulation environment which can be added or removed just by configuration switches.

1 THE PRINCIPAL APPROACH

The fully automated process (called ISG, "Instantaneous System and Software Generation") is based on a modelling language allowing and forcing to express all properties of a distributed real-time system including fault tolerance and fault handling. All relevant functional and non-functional aspects are expressed in one notation. The executable is automatically stimulated with valid and invalid inputs. The degree of time jitter and probability of loss of messages and data can be specified.

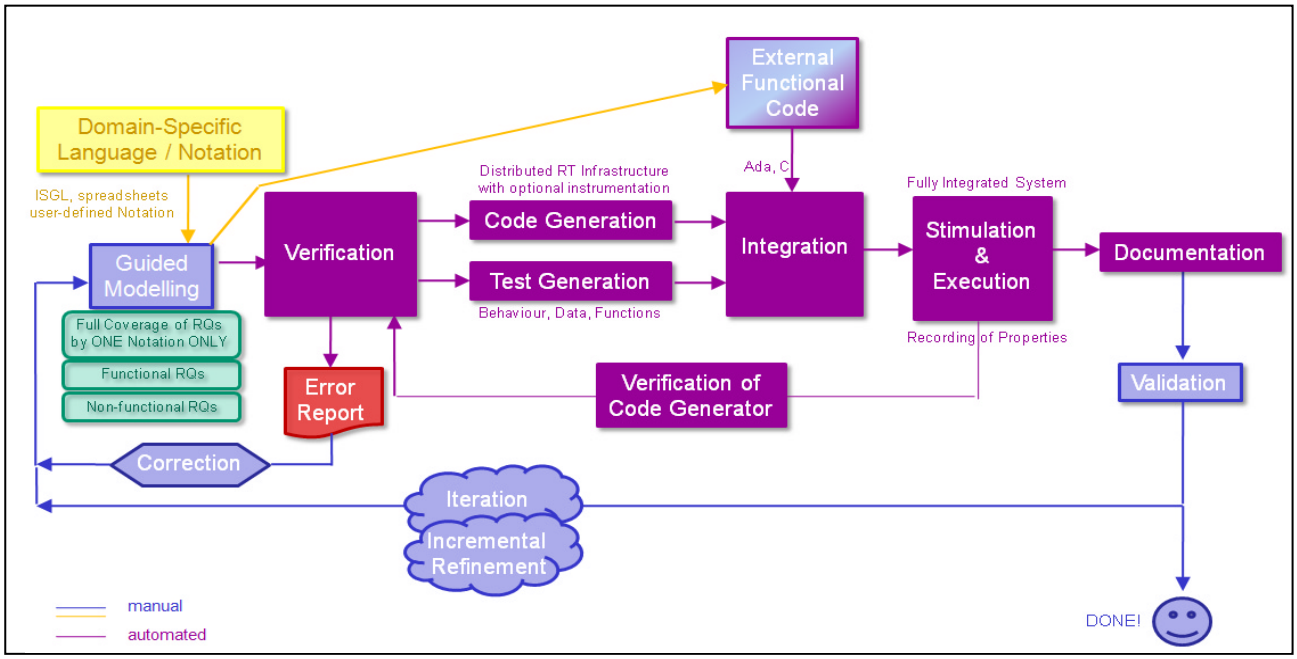

Fig. 1-1: The ISG Process

Due to stimulation the real-time, behavioural, performance and robustness properties can be continuously observed on the development or target platform, supporting a continuous transition between simulation and the target system version.

The modelling approach supports the smooth transition from simulation to the final version. In this context a “simulation version” is considered as a version different from the final version. Consequently, there are many simulation versions – on host and target environment – and one final version. The code generator transforming the specification – the model – into an executable version plays an important role in this process. It does not only generate the code from the model, but adds all the means needed for stimulation, observation of properties , substitution of missing components by intelligent stubs and configuring the code for host or target. So it is just a matter of setting configuration options of the code generator to traverse the different aspects of verification and representative validation in the simulation environment, until the options for the final target version – a lean version – are set.

1.1 Relevance to Avionics

Two main aspects shall be addressed here: the support for validation of real-time operations, focusing on the synchronous approach frequently applied in avionics, and fault identification as an important aspect of verification and validation of a system’s properties.

1.1.1 Synchronous Systems

Currently, modelling and auto-coding in the context of avionics is focusing on critical control systems which are purely synchronous systems. However, in future systems the event-driven part may not be easily integrated into the synchronous concept, e.g. due to a large variation of execution times and required low latencies. In addition, distributed system architectures enforce acknowledgement of the asynchronous nature of real world.

The GALS approach – Globally Asynchronous, Locally Synchronous, historically introduced in the context of “system-on-chip” – raises the need for better support of integration for both types of real-time operations, synchronous and asynchronous ones.

GALS acknowledges the fact that a globally synchronous approach is impossible. Following this conclusion the verification of a synchronous implementation by pure synchronous means is questionable. The cost barrier impeding a more general approach in an asynchronous environment can be reduced considerably by full automation as supported by ISG.

1.1.2 Fault Identification

Fault identification in a simulation environment requires conditions which foster occurrence of faults and their detection..

As it is described in Chapter 2.3.2 the way a system is stimulated may compromise or prevent fault occurrence and identification. Through automated code and test generation a system can be stimulated in the context of its normal operations and under representative timing conditions. Similarly, the detection mechanisms can be built in without compromising fault activation and detection.

1.2 Main Features

1.2.1 Openness

The process covers asynchronous / event-driven and synchronous features. It supports auto-integration of algorithms, e.g. control algorithms, into a distributed real-time infrastructure on the basis of source code (C, Ada) – possibly generated by other code generators such as Scade or Matlab/Simulink.

1.2.2 Dependability Aspects

Verification and validation are major aspects of dependability. The underlying meta-model of ISG allows checking of correctness of a specification. Due to its domain-specific nature, the structure of the specification strictly requires the definition of all relevant properties of the system required for full automation at all stages of the development and maintenance cycle.

Most notably, the modelling environment requires an integrated view on timing and logical behaviour, , thereby providing a wholesale picture of the system to be built and constantly reminding the modeller of the interdependence of functional and non-functional aspects.

So an ISG model unambiguously defines the behavioural und performance properties on modelling level. The real properties are recorded during execution for feedback.

This way a modeller is guided towards a correct model. As the meta-model captures the rules of the application domain more strictly than generic approaches, the system generator can detect many more logical errors before entering the

generation phase than universal approaches, such as UML.

Experience shows that already by such a feedback on modelling errors a modeller significantly can improve the envisaged approach, not only fixing the reported bugs, but being forced to reconsider the current approach, in general. The correctness of the generated code as an important issue is discussed in Chapter 1.2.4.

Validation of the specification is supported by automated evaluation and visualisation of the recorded properties. Due to auto-stimulation the state space can be explored under practical conditions.

Further, automated stimulation with a huge amount of stimuli significantly increases the chance for the detection of sporadic and non-anticipated faults.

The extended verification and validation capabilities provided in the simulation environment are a significant contribution to improve the reliability, availability, safety and dependability properties of a system under development.

1.2.3 Stimulation and Fault Injection

Stimulation extended through fault injection is a typical part of simulation. Simulation opens the door for systematic and representative exploitation of the properties of a system.

Stimulation of a system is derived from its specification. It is straight-forward to define non-nominal values from the nominal values as defined in the specification – provided the specification is expressed in a proper notation, e.g. the ISG modelling language.

In a simulation environment more means are available to systematically exploit the properties of a system – provided representative conditions are guaranteed regarding behaviour, performance and resource properties.

At the end, the model as used for simulation directly can be transformed into a target system version.

1.2.4 Verification of the Code Generator

Today, code generators undergo certification, which – although costly – is unsuited to exclude faults or to even inspire confidence in the absence of faults.

However, the fully automated process of ISG allows automatic verification of every generated application by automatically and independently comparing the specification against the properties observed during automatic stimulation.

Apart from this capability faults in the code generator are easier to detect due to reuse of its parts, because a certain feature of the code generator contributes to several parts of the generated application. For example, the features for generating the behaviour of process or message exchange are needed for every process instance of an application. Through the higher number of occurrences in an application the chances to detect a fault increase accordingly.

The maintenance of the code generator supports convergent removal of detected faults, while in manual case there are a lot chances of introducing faults.

1.2.5 Practicality and Adaptability

The ISG process as it is today evolved from the manual development of distributed real-time systems by continuously finding repeatedly executed tasks and automating them until the full development cycle was completely automated. Formal consolidation of required inputs led to the meta-model and tool landscape at BSSE as it is today.

Automation in this manner avoids unnecessary complexity at model level and not only reduces the effort spent for design and development, but also provides representativity, repeatability and reproducibility of results. This is not possible when manual intervention is part of the generation process, and difficult when introducing automation in a top-down approach.

It can therefore be said that the process originated from bottom-up experience and thus is proven in practice, not only in practicality and product quality, but also in process efficiency.

Although the details of implementation – most notably the hardware, operating systems and communication infrastructure – may vary, the process is flexible enough to adapt to such different requirements.

It is even possible to generate a heterogeneous system regarding the operating system and the processor type. An instance of a process type may run under VxWorks on an Intel architecture, while another instance runs under Linux on a Sparc in the same application. The generation process creates individual code for each of the platforms and takes care of the conversions necessary (e.g. endianness).

In addition, an automated process provides means for representative exploitation of a system's properties through stimulation and fault injection, starting with simulation and ending with the final version.

2 MAJOR RESULTS

The process has been applied successfully to large and complex applications of which four examples are briefly described below.

Due to the fast transition from an idea to the corresponding feedback from an executable a number of aspects are supported:

- Generation of a software product, e.g. the real-time and communication infrastructure of a distributed embedded system, possibly extended through capabilities for telecommanding, telemetry handling and data processing and monitoring, and fault management.
- Verification and validation in the context of simulation by approaching the final version through incremental refinement, iterations and emulation of components.
- architecture evaluation – inherently covered by an iterative approach.
- risk analysis – a consequence of systematic verification and validation support together with full automation – either based on representative simulation or the final version, allowing to get an opinion on a specification from real feedback rather fast.

2.1 From specification to target system at a touch

For a complex material-science experiment on-board ISS the complete distributed real-time infrastructure, the whole chain from (tele-) commanding to telemetry frame generation was automatically generated from a specification including data acquisition with firmware, calibration and limit monitoring and support to fault management.

During hardware-software integration the hardware configuration frequently switched between 1 and 2 processors. Due to the generation process

- adaptation to the different hardware configurations could be done without changing manually any source code. Changing the names of the available processors was sufficient.
- the 2-processor system could continue test and integration as a virtual 2-processor system on one physical processor only – in a representative manner.

Characteristics: 2 processors (Sparc@14 MHz, 6 MB), ~40 processes, 500 data end items, 500 telecommands.

2.2 Architecture Evaluation

For a backup emergency power supply of a nuclear power plant the impact of spatial distribution on a synchronous system with high redundancy (up to 3) and voting was evaluated. The distributed architecture of the system was expressed as specification from which automatically executable code was generated. The algorithms deciding whether the power supply shall be started or not were modelled in Scade, automatically translated into C code and automatically integrated by ISG.

As a result of combining the results from both code generators, ISG and Scade, on C level a virtually distributed, but representative system was established at short hand and its properties were observed in practice. A high number of discrepancies at the final voter stage was detected (~15%) already at low time jitter of ~0.5%.

This contradicted a previously conducted, thorough theoretical analysis suggesting robustness against time jitter. The results as observed in practice were supported by another theoretical analysis which – however – only concluded that a proof on robustness is impossible, without being able to explicitly conclude on insufficient robustness. Therefore robustness was still trusted until the feedback from practice was available.

Characteristics: 16 processes, 16 processors, simple FSMs

2.3 Risk Analysis

Due to the fast transition from a specification to the corresponding product the fully automated approach may be applied to generate a second version of an already existing system to benefit from verification and validation capabilities not available in the native system environment.

Such an approach requires three steps: reverse engineering of the existing code to derive an input which can be fed into the automated generation chain, a bridge to the ISG modelling language ISGL, and the automated generation with ISG itself.

This approach was already successfully applied to the following applications from space area.

2.3.1 Large Flight Application

A large application of 400+ KLOC of Ada code was controlled by 38 Finite State Machines (FSM)**.** The aim was to investigate the properties of the FSMs with ISG. As an extension of ISG, a bridge was added which extracted the relevant information from Ada and transformed it into a model specification, from which subsequently executable code was automatically generated. Nominal and non-nominal stimuli were automatically derived from the specification of the FSMs. Due to auto-stimulation the properties of the FSM could be observed and analysed, though the state space was rather large, even for one FSM (see Fig. 2-1).

Just by looking for red colour in the graphics it was detected that the FSMs are consist of a number of subnets of states with no transition possible between different subnets

There was an unchecked mechanism foreseen at lower level to command transitions from any state to any other state, but it was impossible to track the relevant information – neither manually nor automatically – due to type casting and a complex data flow in Ada in combination with roots in the database. The conclusion was that the reachability of states was still an open issue to be tackled.

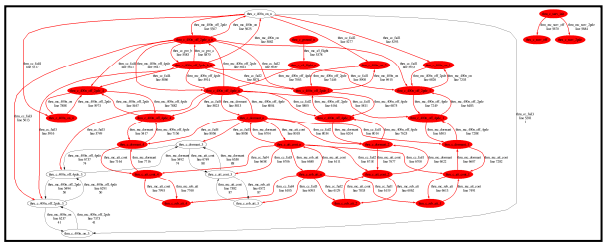

Fig. 2-1: Unexpected Unreachable Subnets of States

Characteristics: 38 processes, 1 processor, 380 states, 1500 transitions, 10000 actions in the transitions, overall state space ~10^{34}.

2.3.2 Fault-tolerant Processing

For demonstration of capabilities of model-based software engineering (MBSE) a model of a command processing was established which should tolerate loss of commands during command verification, queuing and execution.

The model was established in UML2 and was subject of verification based on the supported means of the chosen tool. The verification result as obtained in the UML2 environment was: perfect.

To get a second opinion on verification the UML2 model was automatically transformed to ISGL. The obtained report on system properties showed some red colour for coverage of states and state transitions (see Fig. 2-2).

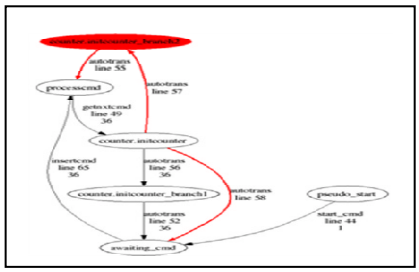

Fig. 2-2: Red Colour as Marker for a Non-anticipated Fault

A brief analysis yielded that the missing coverage occurred in a fault handling branch, only entered when a command is lost. Therefore the fault injection capability for loss of signals was activated, expecting now full coverage.

However, the situation became worse: more states were not covered instead. A further analysis showed that the fault-tolerant algorithm was faulty and caused a deadlock.

This could not detected before, because the UML2 tool did not support automated loss of commands. Verification could only be achieved by stepping manually through the model. This may have suppressed a condition– unintentionally – which otherwise would have enforced entering the fault-handling branch. Therefore the real consequences of a loss never were observed before.

Also, it was very difficult to detect the deadlock by manual inspection, because the involved logic was spread over two processes.

In the automated case the loss was induced automatically in the real system context, not having any chance of modifying the execution sequence in a way compromising the verification result.

This fault would have been immediately detected in practice when a loss of a command would have been occurred, because the deadlock would have stopped execution. This is the good point. However, the big question is, whether the loss would have been initiated on the real system before putting it into operation, or whether engineers would have completely relied on the results of verification as obtained in the modelling environment.

Characteristics: 4 processes, 1 processor, states, state transitions, actions.

3 BEYOND SYNCHRONOUS PROCESSING

The synchronous paradigm eases verification of a system's timing properties by considering periodic and fix-phased processing. Essential safety properties can be easier proven when supposing synchronicity.

However, the world is dominated by asynchronicity, and synchronous approaches attempt to artificially consider asynchronicity in a synchronous context.

While the lower hierarchy levels of a system – which are close to the physical environment – are of synchronous nature, the asynchronous properties are more dominant, the higher the level in the system hierarchy is. E.g. at higher levels command execution, exception handling or reconfiguration has to be done, which is heavily event-driven and nearly can be harmonised with a synchronous approach.

Clock synchronisation is difficult to achieve in a perfect manner within a distributed system. This led to the GALS architecture, originally introduced for "System-on-Chip": Globally Asynchronous, Locally Synchronous.

As outlined in Chapter 2.2 the results of safety analyses based on the synchronous paradigm may not always be true, because hidden facts compromise the basic assumptions of a proof.

From this point of view it must be doubted whether the result of a single analysis can be considered as the definitive answer on safety properties. This question is even more essential for critical systems, and also addresses non-anticipated faults e.g. in the proofer algorithms or their implementation.

Therefore, a second opinion may be very valuable to confirm an analysis result in an environment representative for the real world, i.e. an environment supporting asynchronicity. Fault injection, e.g. time jitter, causes asynchronicity regarding the synchronous paradigm and focuses on the robustness in real world context.

An automated transition to an executable version – as described in Chapter 2.2 – providing sufficient capabilities for observation of properties is a cost-efficient approach when needing a second branch on verification under representative conditions of real world.

4 IMPACT ON FAULT IDENTIFICATION

Through execution of a number of exercises three major improvements for fault identification were recognised:

- the enhanced support for identification of sporadic and non-anticipated faults by raising the activation probabilities,
- the identification of faults though not knowing the desired properties, and
- raising fault identification probability through diversification, i.e. through an independent opinion.

4.1 Increased Identification Probability

Through automation much more stimuli can be injected without disturbing the timing conditions for fault activation and identification. In addition, fault injection can be seamlessly integrated with nominal stimulation. The same is true for instrumentation of the code to record the observations.

This allows to achieve a significantly increased coverage of system states up to full coverage and visualisation of recorded properties in different shapes of detail and from different points of view, needed to fully understand the observed properties.

4.2 Capturing of Non-Anticipated Faults

4.2.1 Identification by Symptoms

Fault identification based on symptoms is not really new. However, it was recognised that it may systematically be extended once the mechanisms were understood as very valuable for identification of faults, especially of non-anticipated ones.

A program abort is a classical mechanism for detecting a fault without knowing in detail what a system ought to do. Observation of such a symptom is the starting point for identification of the source of the fault. In fact, a symptom like an abort is also a property of a system – but an undesired one, which is covered by rather generic requirements like "A system should never abort" – if such a requirement is given at all.

However, from such a requirement there is no direct way to symptoms flagging a fault. Moreover, several ways usually exist leading to the same or different faults. It is likely that a symptom may help to detect a number fault types manifesting in a fault.

Having identified the capabilities of symptom-based fault identification it is straight-forward to extend the known scheme.

E.g. coverage of states, state transitions or code (on model or source code level) is a valuable concept for identification of non-anticipated faults based on a requirement like “Full coverage shall be reached”. However, it is evident that symptoms may be observed without the need for an explicit requirement when providing adequate reporting capabilities. Automated code generation perfectly supports such means for recording and visualisation of symptoms.

Sufficient visualisation is very important. This does not only mean just to print or draw something, it is essential to present the information comprehensively, e.g. by filtering of information, because usually there is a huge amount of information in which some small, but important pieces could be hidden.

The red colour in Fig. 2-1 and Fig. 2-2 is a simple measure to mark a fault symptom in a way easily understood.

Finally, it should be mentioned, that symptom-based fault identification is much more expensive than rule-based identification (violation of a rule). So an optimum strategy must try rule-based identification to the largest extend possible to be efficient. However, symptom-based identification is the only way for identification of non-anticipated faults, and should be applied later.

4.2.2 Diversification

The usual cost-driven approach suggests to test as close as possible to the target system. This is reasonable when the nominal properties shall be demonstrated, because a deviation on a non-representative environment does not mean anything. This understanding is extrapolated to fault identification, thereby restricting conditions to the ones occurring on the target system.

However, experience gained in the context of automated generation – just by turning a switch of the code generator into another position – suggest that it could be more efficient to take a non-representative platform for fault identification. Of course, then rule-based identification is out of scope. Symptom-based identification is the only goal. Especially, dormant faults can be identified easily this way.

4.2.2.1 Platform Diversification

A dormant fault may be hidden on the native platform, but may immediately raise an activation condition and flag a symptom on another platform. An example is detection of access of NULL pointers which may dynamically occur and therefore cannot be detected by analysis. On the target platform such an access may be tolerated, e.g. under VxWorks on an Intel x86, on another platform not, e.g. under VxWorks on a Sparc.

In this context a platform is understood as the triplet of operating system, compiler and processor type / architecture. There are multiple reasons for that: an OS cannot support a symptom’s identification feature because it is not supported by hardware, or it is not implemented in the OS or the compiler, be it intentionally or unintentionally.

Another example is based on differences in processor architecture, e.g. RISC and CISC. It was recognised that an overflow in byte operations was masked by the RISC 32-bit architecture, while on Intel x86 with byte-instructions it was detected. Though on Sparc this is a dormant fault – as the result was correct, at least in this case – its (future) impact remains unpredictable in general, because the behaviour is undefined and depends on the actual capabilities of processor hardware and compiler.

Remark: From a rigorous point of view it is reasonable to spend a considerable amount of effort to detect dormant faults even if they do not compromise a result in the tested cases. However, a valid conclusion “result not compromised” can only be done in general when a dormant fault has been detected and classified. As such an analysis is required after every change in the software – as long as the dormant fault is left in the software, the more efficient way is to fix it.

4.2.2.2 Method and Tool Diversification

Apart from platform diversification another option was recognised for diversification as already mentioned in Chapter 2.2: complementing analysis by testing. Of course, the other way around is also valuable.

It has been observed that certain methods do not support detection of certain fault types from a principal point of view. Similarly, the related tools may not support it – either intentionally or unintentionally, though the method does it. In consequence, it is not sufficient, just to apply one tool or method, when a system is considered as critical.

Exact knowledge on method and tool capabilities is required to have a chance to maximise coverage of fault types. Extended, non-expensive stimulation based on automation is a valuable approach for complementing analyses or legacy simulation

environments raising the fault identification probability by independent verification means.

4.2.2.3 Summary on Diversification

Automation – which does not require any manual intervention to reach the intended goal – significantly increases the chances of detecting a fault, either by opening the door towards different platforms at low costs, or by complementing the primary methods and tools by extended and independent fault identification capabilities.

5 PRESENT AND FUTURE WORK

The ISG tool for automated generation of distributed real-time systems from a specification without any manual intervention was the first in a series of generators followed later, e.g. for test generation from source code, interface adaptation, integration of source code, generation of SQL and the related GUIs, extension of legacy code through additional functionality, code and model transformation.

Another interesting task is the automated porting of legacy code to a modern platform when the previous one disappears.

Currently, timing analysis is based on stimulation with fault injection and observations in context of ISG, i.e. on test and dynamic analysis. In future, capabilities for static analysis of timing properties shall be made available through a bridge to a sophisticated analysis tool, considering more representative conditions like a task's inner states and margins instead of exact performance values. This will enhance the validity of the overall result due to diversification considering test and analysis.

6 CONCLUSIONS

The automated approach for generation of executables from a specification implies the following major benefits:

Firstly, a fast transition to the currently specified version of a system through iterations and refinements is supported, which provides higher flexibility at low costs to obtain a high quality product. Automated generation requires excellent verification capabilities, otherwise garbage fed in at top will cause much more garbage in the form of generated code.

Secondly, the verification and validation capabilities significantly increase the chances of fault identification, either by stimulation (incl. fault injection) at high stimuli rates, which never can be achieved manually, or by stimulation at realistic conditions not implying a risk to compromise fault activation or fault identification.

Finally, due to the availability of a second version at low costs diversification can be applied easily, thereby raising the probability of fault identification.

Regarding risk reduction the direct consequence of experience gained in the past is to focus on a systematic fault identification strategy based on diversification where the alternate versions can be obtained immediately – more or less – and at low costs.

This leads to a "software bus" based on automated transformation of representations opening the door for complementary or identical verification capabilities. If complementary, more properties can be exploited, verified and validated, if identical, results can be cross-checked.

Moreover, automatic test – though not perfect – turned out as a powerful means to capture faults, especially regarding non-anticipated faults, which cannot be identified by analysis from a principal point of view.

However, whenever a non-anticipated fault is identified through testing, automatic analysis should be extended by adding a rule making the fault an anticipated fault.

Another remarkable finding is that it is possible to identify faults based on certain symptoms without requiring any detailed knowledge about an application. This is not really new as we know from crashes flagging a non-anticipated fault. What is new, however, is to enforce this identification capability in a systematic manner and to extend it from run-time anomalies to more sophisticated anomalies where automation makes it feasible in practice.

The means described above significantly enhance the quality of a product through simulation capabilities while reducing the costs for development and maintenance.