Towards a Discipline of System Engineering: Validation of Dependable Systems

Andrea Bondavalli*, Alessandro Fantechi**, Diego Latella* and Luca Simoncini***

*CNUCE-CNR, Pisa, Italy
**University of Florence and IEI-CNR, Italy
***University of Pisa and CNUCE-CNR, Italy

Abstract

Complex systems require the use of an integrated and best balanced set of components. The integration and the balanced set are crucial issues, which require some sort of verifiable compositionality property of component parts that contribute structurally, functionally, non-functionally and interactionally to the total quality of the system design. This is even more important when dealing with the design of highly dependable systems. The concept of verifiable compositionality is much more demanding than the usual approach based on composition of building blocks. It implies the preservation of properties and the ability of verifying them, as well as those that are added (which mainly deal with interactions among parts) in the process of designing and building a system made of components. Economic reasons push towards the use of COTS (Commercial Off the Shelf) and towards the re-use of available components and this trend poses new problems.

Integration, compositionality and re-use appear to be the very challenging issues in the validation (of both design and implementation) of complex systems, in particular dependable ones used for controlling critical applications, and require a special effort towards the emergence of a new discipline - System Engineering - which will encompass and integrate the current design disciplines.

This paper aims at a discussion in the direction of identifying possible advanced approaches to the validation of dependable systems.

1: Introduction

Computer controlled systems are used in many fields of applications, with different levels of criticality requirements. A common characteristic of such systems is the increasing complexity in intrinsic terms (management of distribution, redundancy, layering of functionalities, etc.) and of the in-the-field operation (interfaces towards the environment, timing constraints, criticality of the controlled applications, etc.). This increasing complexity rarely can be completely mastered and usual design practices often suffer from partial approaches, overlooked details, inadequate modelling, insufficient prototyping, limited design tools or techniques available; not to deal with incorrect or incomplete or not understood user requirements, which are often the cause of the final failure of a design or system [58].
The solution to this situation is far away to be reached, but for sure requires the development and the testing in the field of design processes which address the “system” in its entirety and integrate as many as possible relevant requirements, including the interactions with the environment.

The current disciplines of hardware, firmware, software, application engineering, commonly considered as separated issues have to evolve towards a discipline of “system engineering” which encompasses all the previous ones, if we have to rely on systems which control critical environments [60].

Mil-STD-499B has defined System Engineering as: “an interdisciplinary approach to evolve and verify an integrated and optimally balanced set of product and process designs that satisfy user and society needs and provide information for management decision making.”

This definition is the evolution (and abstraction at a certain level) of the definition given in MIL-STD-499A:
- “the application of scientific and engineering efforts to
  - transform an operational need into a description of system performance and a system configuration through the use of an iterative process of definition, synthesis, analysis, design, test, and evaluation;
  - integrate related technical parameters and ensure compatibility of all physical, functional, and program interfaces in a manner that optimises the total system definition and design;
  - integrate reliability, maintainability, safety, survivability, human engineering, and other such factors into the total engineering effort to meet cost, schedule, supportability, and technical performance objectives.”

Another interesting definition of System Engineering is the one by Joe DeFoe and Jim McAuley: “the processing of building real things to solve real problems within technological, environmental, economic, legal, ethical, cultural, and institutional constraints.” [52]

Although sounding philosophical, this definition is focused on the different types of constraints which a system has to fulfil.

Having this in mind, we can concentrate on the definition given by Mil-STD-499B, which has the merit of pointing out:
- **the interdisciplinary approach used to satisfy user needs**: computer engineers have to interact with other engineers (like mechanical or chemical etc.) and mainly with the final users and the society (who will present all kinds of needs – environmental, legal, ethical etc.);
- **the evolution and verification of product and process designs**: the quality of a product design is dependent on the quality of the process in which is inserted, and these qualities have to be verified; the product-in-a-process approach is an evolving one and changes and modifications have to be taken into account;
- **the evolution and verification of an integrated and optimally balanced set** of product and process designs; if the word “optimally” is taken literally, only techniques which allow quantitative measures of quality parameters, and their combination, could be applied and this would limit the applicability of a system engineering approach (simply consider a system to be designed for high security). Probably the current status of the art is better represented by rephrasing with “best balanced set”, which allow the inclusion of all best-effort techniques. The “integration” and “the balanced set” are crucial points, which requires some sort of “verifiable compositionality” property of component parts that contribute to the total quality of the system from the points of view of structure, functionality, non-functional properties and interaction. The concept of “verifiable compositionality” is completely different from and much more demanding than the usual approach based on composition of building blocks. It implies the preservation of properties (both structural, functional and non-functional) and of
the ability of verifying them, as well as those that are added (mainly interactional), in the process of building a complex system made of components.

- **the provision of information for management decision making**: this point is of prevalent importance in the product-in-a-process approach and is self-explaining.

Based on these considerations, the INCOSE (International Council on System Engineering) - SE Practice Working Group - Subgroup on Pragmatic Principles has issued on January 21, 1993 a document, edited by J. C. DeFoe, “Pragmatic Principles” of Systems Engineering”, which present a list of points to be followed for a correct product-in-a-process approach [52]. The complete list is reported in Appendix A.

The points of the list covers the entire spectrum of the product-in-a-process approach and we limit ourselves to the discussion on those points which relate intrinsically to the system engineering approach applied to the validation of dependable system architecture design. We are aware that a total quality product and process design cannot overlook the points that we do not discuss, but our opinion is that something beneficial can be obtained even with our limited discussion.

Our attention will therefore be focused on the points indicated in the section E) of the list in appendix A.

2: Validation of dependable system architecture design.

The increasing complexity of computer controlled systems has exposed the limits of the validation techniques traditionally used in industry, like code review, testing, Fault Trees or Failure Mode Error Analysis to cope with the increasingly demanding dependability requirements asked to these systems.

Moreover, new technologies such as Object-Oriented Design and Programming, Advanced User Interfaces, Hardware-Software Codesign, the use of COTS (Commercial Off the Shelf) software components, all present new challenges for the validation process.

The traditional validation techniques are being more and more complemented with advanced validation techniques, such as Formal Verification, Model based Dependability Evaluation, Schedulability Analysis, Fault Injection, which are individually discussed later in this section.

These techniques are not aimed to replace the traditional validation techniques, but should rather be integrated with them.

2.1: Formal verification

Formal verification is a hot topic nowadays in the field of system engineering, specially for the development of critical dependable systems. The use of formal methods for the specification and verification of properties of systems is one methodological improvement of the system production process, which, together with other techniques, can make it possible to reach high quality standards: the use of formal methods is increasingly required by the international standards and guidelines for the development of safety critical computer-controlled systems.

Formal methods are mathematically-based techniques that can offer a rigorous and effective way to model, design and analyse computer systems, and they have been a topic of research for many years and the question now is whether these methods can be effectively used in industrial applications. Tool support is necessary for a full industrialisation process and there is a clear need for improved integration of formal method techniques with other software engineering practices.

Several approaches to the application of formal methods in the development process have been proposed, differing for the degree of involvement of the method within the development process,
ranging from the mere use of formal specifications in order to write rigorous specifications, to the (more or less automated) generation of code from the formal specification, to the use of formal verification as an additional validation technique aimed to reach a high level of confidence of the correctness of the system.

Industrial acceptance of formal methods is strictly related to the investment needed to introduce them, to the maturity of tool support available, and to the easy of use of the techniques and tools. The use of graphical representation is often preferred, even if some formality is lost. So, despite some successful experience of application of development processes built around a specific formal method (see for a main example [35]), the industrial trend is currently directed towards the use of graphical specification formalisms (a notable example is SDL [83]), which nevertheless have a (almost completely) formal semantics; due to the lower costs of training and innovation, industries are more keen to accept formal verification techniques assessing the quality attributes of their products, obtained by a traditional life cycle, rather than a fully formal life cycle development.

Formal verification methods and tools can be roughly classified into two categories, the so-called model-theoretical approaches and proof-theoretical ones.

In the proof theoretical approaches, the system state is modelled in terms of set-theoretical structures on which invariants are defined, while operations on the state are modelled by specifying their pre- and post-conditions in terms of the system state. Properties are described by invariants, which must be proved to hold through the system execution, by means of theorem proving.

Model theoretical approaches, on the other hand, work on a finite state representation of the behaviour of the system. Verification is usually carried out by checking the satisfiability of some desired properties over the model of the system by model checking algorithms or equivalence relations. In particular, safety requirements may be expressed as temporal logic formulae and may be checked on the model of the system.

Model theoretical approaches give a direct automatic verification method of system properties. Unfortunately, this approach suffers of the so called "State Space Explosion" problem: systems composed of several subsystems can be associated to a finite state model with a number of states which is exponential in the number of the component subsystems. Moreover, systems which are highly dependent on data values, share the same problem producing a number of states exponential in the number of data variables. Hence, traditional "model checking" techniques [30], have shown themselves not powerful enough to cope with many "real" systems.

Recent advances to cope with the state explosion problem have seen the use of symbolic manipulation algorithms inside model checkers: the most notable example is the SMV model checker [21]. In SMV the transition relations are represented implicitly by means of Boolean formulae and are implemented by means of Binary Decision Diagrams (BDDs, [20]). This usually results in a much smaller representation for the systems' transition relations thus allowing the maximum size of the systems that can be dealt with to be significantly enlarged. This kind of tools have been successfully applied to very large state spaces in the realm of hardware verification.

On the converse, proof-theoretic approaches, which can exploit their generalisation capability in order not to be affected by the state explosion problem, require in general more skill in the use of theorem proving tools, and, therefore, more investment, in terms of know-how and training. This is because proofs usually need to be guided by humans and so the theorem proving process is not totally automatic. A consequence of this is that the proof details cannot be hidden to the user.

The last years have seen some effort in the integration of model-checking and theorem-proving approaches [66], and this seems to be the most interesting perspective.
2.2: Model based dependability validation

Various methods and tools for dependability modelling and analysis have been developed which provide support to the analyst, during the phases of definition and evaluation of the models. In general, model types used for dependability analysis are usually classified in two categories; combinatorial types and state-space based ones, which include both Markov models and high level approaches which have an underlying Markov model [61].

Combinatorial models: these include reliability block diagrams, fault trees and reliability graphs. It was shown in [61] that Fault Tree with Repeated Events (FTRE) is the most powerful type. A method that combines the simplicity of Fault-Trees with the more powerful representative capabilities of state based approaches has been proposed in [37].

Markov chains: The approach to the modelling based on Markov processes has been widely accepted in the dependability community because of their powerful representative capabilities, and the relatively cheap solution techniques. Constant failure rates are used. While this seems to be natural and has always been accepted for hardware elements it has also been applied to software failures. A failure rate \( \lambda_j \) for software can be computed as a product of a constant execution rate of the software and its failure probability:

\[
\lambda_j = P_j \lambda
\]

where \( \lambda \) is the execution rate, and \( P_j \) is a failure probability.

Such an approach for modelling software fault tolerance architectures has been adopted in [6] and many subsequent papers. The service of the system is modelled through execution rates and the fault manifestation process by failure probabilities.

High-level modelling: Markov-based approaches for dependability modelling and evaluation must face with the increasing complexity of systems, and consequently of the models. The state space combinatorial growth leads to a dramatic increase in the size of Markov chain models, which tend to become difficult to define and computationally intensive to solve. A solution is offered by modelling tools at a higher level than Markov chains, like Queuing Networks, Stochastic Process Algebras, and Stochastic Petri nets. All exhibit the ability to deal with different abstraction levels of the analysis. Usually, the solution of these models is based on a direct transformation to Markov models. However, high-level models have advantages in the model generation phase, because very compact models can be given even for complex systems, and can enjoy efficient state space reduction algorithms in the solution phase as well.

Those based on Petri nets models are becoming more and more popular due to the appealing graphical visualisation of the models and the natural way in which concurrency, competition and synchronisation are all easily represented within the formalism.

Many different classes of Petri nets have been proposed over the past decade. The basic untimed class of place/transition Petri nets was augmented with the time for the sake of quantitative analysis of performance/dependability attributes, thus defining the class of Stochastic Petri Nets (SPN) [63]. SPNs only consider activities whose duration is an exponentially distributed random variable. This limitation was overcome with the introduction of Generalised Stochastic Petri Nets [1] (GSPN), which allow for both exponential and instantaneous activities. The stochastic process underlying a SPN and a GSPN model is a discrete space continuous-time homogeneous Markov process, which must be solved to derive the measures of interest for the system.

Nearly all the tools for dependability modelling and evaluation that are based on Petri net models can be used to define and solve GSPNs. What may be different from one tool to the other is merely a matter of the syntax used to define the model; in this sense, GSPN models can be seen as a standard language which is understood by the majority of the tools for the automated
evaluation of dependability attributes, like SURF-2 [57], UltraSAN [79], SPNP [29], GreatSPN [26], TimeNET [44].

**GSPN extensions:** Several extensions of the GSPN class of Petri nets have been further introduced. These extensions can be distinguished in two classes, depending whether the representative power of the extended models is increased beyond that of GSPNs.

For the extensions that do not increase the representative power, the stochastic processes underlying the models are still Markov processes. In this case the extensions provide useful shorthand notations to represent in a concise way complex dependencies among the elements of the model. The Stochastic Activities Networks (SAN) [72] and the Stochastic Reward Nets (SRN) [77] are two classes of Petri nets that include such extensions. UltraSAN and SPNP are the automated tools for the solution of SANs and SRNs, respectively.

On the other hand, there are classes of Petri nets whose underlying stochastic process is not a simple Markov process. For instance, consider the class of Deterministic and Stochastic Petri Nets (DSPN) [2]. DSPN include transitions having exponentially distributed, immediate and deterministic firing time, and are therefore able to represent models GSPNs can not deal with. The tool TimeNET [44] was especially developed to solve DSPN models. An even more powerful class of Petri nets is represented by the Markov Regenerative Stochastic Petri Nets (MRSPN) [28], which allow for transitions having generally distributed firing times. No automated solution tool exists for MRSPNs, yet.

The quantitative analysis of the dependability attributes of computer systems using stochastic modelling is a process that requires ability and experience. Building the model of a system needs the introduction of assumptions, simplifications and *abstractions*, whose impact on the final results can not be estimated a priori. Also, slight variations in the value of a crucial parameter might cause dramatic changes in the final measures. Actually, analytical models had always as their usual target mechanisms or specific parts of the systems. When they have been used for modelling entire complex systems (e.g. [56, 64]) several problems have been encountered like subtle interactions between hardware and software, stiffness of the models (i.e. when the rates of transitions differ by several orders of magnitude) and primarily state explosion. Despite modularity in defining the model can be achieved, the model has to be solved in its entirety to maintain the Markovian properties. Unless more powerful methods can be used, it is not possible to define hierarchies of models where sub models can be separately solved and the results obtained plugged in a higher level model of the hierarchy. Therefore models of real - even rather simple - systems tend to be very abstract and usually do not account for (maybe important) details. This need of carefully selecting the characteristics to be represented is probably also the cause of the extreme difficulty in the automatisation (or even the definition of rigorous methodologies) of the process of model definition.

### 2.3: Schedulability analysis

In real-time systems, a failure of a system can be caused not only by a functional fault or by an hardware fault, but also by its inability to perform the needed computations within the time limits established for them. Such inability is typically due to the attempt of executing too many different tasks with limited computational resources.

The validation of temporal constraints mainly consists in the search of a *feasible* schedule, i.e. an ordering of the execution of the tasks admitted to the system and for which it can be proved that each task will complete before its deadline. Scheduling has been studied for long time and many proposals exist for static scheduling and (much fewer) dynamic approaches. Without entering in details on the individual algorithms [74] it is remarkable to note that guarantees (absolute or probabilistic) are given by making many, often unrealistic, assumptions
on the number of tasks, their maximum execution time, precedence constraints and resource conflicts, etc. Moreover, very restrictive assumptions are often made on the admitted faults so that a few works exist on real-time fault-tolerant systems [43, 76, 81]. On the other hand, modern design languages, including those based on the OO paradigm, require restrictions to the computational models, i.e. the allowed interactions among tasks, for allowing to design system with real-time requirements.

For these reasons, until recently no commercial tool supporting schedulability analysis were available. In the recent times, tools like PERTS from TriPacific Software [67] and TimeWiz from TimeSys [78] devoted to the early schedulability analysis have supplemented the available low level debugging or execution tracing tools. What is actually missing is an overall integrated design environment that, within a coherent design methodology, allows to evaluate the impact of the different priority schemes and scheduling algorithms on the real-time system performance and functionality, already at the design level. One step in this direction has been taken by the HRT-HOOD [HOOD 1992] design methodology, which addresses explicitly the design of hard real-time systems in an object based framework, providing means for the verification of their performance. The overall methodology can be supported by automated tools, such as HRT-HoodNICE [53].

2.4: Validation based on fault injection

Despite the continuous advances in systems modelling, several limits remain on the possibilities offered by analytical approaches. Therefore, a specific experimental effort remains necessary to test the behaviour, in presence of faults, of fault tolerance mechanisms of the systems. The fault injection is an activity aiming at measuring the characteristic parameters of fault treatment and error treatment mechanisms. Fault injection experiences contribute to the estimation of the coverage and efficacy of fault tolerance and complement the analytical techniques previously described.

Independently of the abstraction level applied, fault injection campaigns contribute to the ‘Fault Forecasting’ by studying either error propagation [24, 27, 75], or error latency [25, 42] or the coverage of fault tolerance mechanisms [70, 80]. Many techniques and methods have been developed and integrated in tools specific for fault injection, e.g. [3, 23, 55].

As previously mentioned, fault injection is particularly tailored for estimating the coverage of fault tolerance techniques. In particular, the notion of coverage can be applied to errors, specifically to trace the efficacy through the different steps characterising error treatment. Moreover, one should also consider the dynamic dimension of the treatment process leading to the definition of coverage as a function of the time at which the expected treatment is performed. Despite many studies dealt with both coverage and latency, only a few tried an integration by considering them in a unified way, see [5, 38, 62].

The imperfect coverage of fault tolerance mechanisms is due to two main causes:
- the wrong definition/application of the mechanisms with respect to the considered fault assumptions, which results in a lack of the coverage of the mechanisms
- wrong assumptions on the faults which are different from those that hit the system in operation, which results in a lack of coverage of the assumptions

Fault injection is particularly good in revealing imperfections due to the former cause. To estimate coverage of the fault assumption other kind of analyses are required.

Two main criteria must be considered for the experimental validation of fault tolerant systems: the system abstraction level and the form in which injection is applied. For what concerns the system abstraction level it is possible to distinguish the case in which a physical system is available from a simulation model describing the structure and or the behavior of the system. The
form in which injection is applied can be either physical injection - faults are directly introduced on physical components through mechanical or electromagnetic alterations, or logical injection - alteration of Boolean variables or data.

Most studies on physical injection used pin-level injection although other techniques like heavy-ion radiation have been used for specific systems. Despite problems of the doubtful representativeness of the injection on the pins of integrated circuits, the other main problem arising is the accessibility of the physical component themselves, due to higher and higher integration and clock rate. The main solution is then represented by the usage of hybrid injection - faults are injected at the logical level of the physical components.

Fault simulation is heavily used for developing test cases in industrial environments, e.g. for quality control of integrated circuits. Still currently there are only a few examples of evaluation of the coverage of fault tolerant mechanisms performed by using simulation models.

As a last comment about fault-injection, we notice that the same principle can be applied also at a more abstract level: in the framework proposed in [10] for the formal validation of fault-tolerance mechanisms, specific anticipated faults are injected in the specification of the mechanisms, with the aim of analysing the behaviour of the specified mechanism in case of faults, and to guarantee that the mechanism satisfies given correctness properties even in case of faults.

2.5: Needs for integration of validation

The discussed validation techniques are separated and not integrated among them; that is, each technique requires to define a particular model of the system, each focused on those aspects which are specific of the technique itself, with no relation with the other models built to apply the other validation techniques.

In a coherent "system engineering" approach the application of these techniques cannot be seen as a mere juxtaposition of unrelated techniques, but rather as an integration of techniques using different, but semantically related models.

In the next two sections we discuss what has been done in this respect in two recent European projects, GUARDS and HIDE. While GUARDS is a very updated best effort example of use of different advanced techniques for design validation, HIDE tries to propose a possible approach to integration.

3: The GUARDS validation framework

In this section we first recall the GUARDS validation framework and discuss its limitations regarding integration. GUARDS [68] is an effort, partially supported by the European Community as ESPRIT Project 20716, to design and develop a Generic Upgradable Architecture for Real-Time Dependable Systems, together with its associated development and validation environment. GUARDS aims at reducing the life cycle costs of such embedded systems. The intent is to be able to configure instances of a generic architecture that can be shown to meet the very diverse requirements of critical real-time application domains. The cost of validation and certification of instances of the architecture thus becomes a critical factor. The effort to reduce this cost has exploited:

• re-use of already-validated components in different instances
• the support of software components of different criticalities
• and focus validation obligations on a minimum set of critical components.

The GUARDS validation strategy considers both a short-term objective which is the validation of the design principles of the architecture and a long-term one being the validation of instances
of the architecture for specific requirements. Different methods, techniques and tools contributed to these validation objectives.

The validation environment that supports the strategy is depicted in Figure 1, which illustrates also the relationship between the components and their interactions with the architecture development environment.

![Figure 1 — Main interactions between architecture development and validation (from [68])](image1)

The figure explicitly identifies the main validation components: formal verification, model-based evaluation, fault injection and the methodology and the supporting toolset for schedulability analysis. The figure also depicts the complementarity and relationships among the validation components. In particular, fault injection, carried out on prototypes, complements the other validation components by providing means for: a) assessing the validity of the necessary assumptions made by the formal verification task, and b) estimating the coverage parameters included in the analytical models for dependability evaluation.

### 3.1: Formal verification

Formal approaches were used both for specification and as a design-aid and applied for the verification of critical dependability mechanisms, namely: a) clock synchronisation, b) interactive consistency, c) fault diagnosis, and d) multi-level integrity. The first three mechanisms constitute basic building-blocks of the architecture, and the fourth one corresponds to a major innovation.

The formal approaches that have been applied included both theorem proving and model checking.

The work carried out on the verification of clock synchronisation relied heavily on PVS (Prototype Verification System) [66]. It led to the development of a general theory for averaging and non-averaging synchronisation algorithms [73].
The verifications concerning interactive consistency [7, 9], fault diagnosis [8] and multi-level integrity [40] were all based on model checking using the JACK (Just Another Concurrency Kit) toolset [14] and on the framework introduced in [10] to deal with dependable systems. This integrated environment provides a set of verification tools that can be used separately or in combination. Due to the complexity of the required models, the toolset was extended to include a symbolic model checker for ACTL [39].

3.2: Model-based dependability evaluation

Model-based dependability evaluation in the GUARDS context posed several of the modelling problems discussed previously ("stiffness", combinatorial explosion, etc.).

To master them, different modelling activities have been carried on, choosing modelling details and levels to fit the specific evaluation objectives. This was achieved by focusing either on generic or specific architectural features, or on selected dependability mechanisms. The “focused” models addressed several issues concerning the analysis of generic mechanisms (e.g., $alpha$-count [11]) and of specific features for selected instances (phased missions, for space prototype instances [13], intra-channel error detection for railway prototype ones).

Then, elaborating on previous related work (e.g., [56]), an incremental approach proposing modular constructs has been devised. This second viewpoint aims to establish a baseline set of models of the prototype instances of the architecture [69]. A general notation is introduced that allows for a consistent interpretation of the model parameters (layers, correlated faults, etc.). This work guides the choice of a particular instantiation of the architecture, according to the dependability requirements of the end-user application. Last, a third viewpoint was considered that aims to provide detailed models needed to allow for a more comprehensive analysis of the behaviour of the instances (dependencies, error propagation, etc.). Specific work has addressed hierarchical modelling with the aim of mastering the complexity attached to the development of such detailed models [54]. This work is directed mainly at: a) enforcing the thoroughness of the analysis, b) helping the analyst (i.e., a design engineer who is not necessarily a modelling expert).

Although different tools have been used all modelling is based on Stochastic Petri nets to allow re-use of the results of the various studies (both models and modelling methodology).

3.3: Schedulability analysis

The design and development of a GUARDS software application are centred on a hard real-time (HRT) design method, which allows real-time requirements to be taken into account and verified during the design. The method also addresses the problem of designing replicated, fault-tolerant architectures, where a number of computing and communication boards interact for the consolidation of input values and output results.

Despite GUARDS does not force the selection of a specific method, a survey and an analysis of design methods have shown that only HRT-HOOD [22] addresses explicitly the design of hard real-time systems, providing means for the verification of their performance. Therefore, HRT-HOOD was selected as the baseline design method and HRT-HoodNICE adopted as supporting tool [53].

To deal with the design of distributed systems, HRT-HOOD was extended to include the concept of Virtual Nodes, similar to that in the HOOD 3.1 method [50]. The HRT-HoodNICE toolset has been accordingly enhanced.

The application tasks (i.e., HRT objects) are mapped onto the infrastructure architecture. They are coupled with the real-time models of the selected components, in order to analyse and verify
their schedulability properties. This is done by the Temporal Properties Analysis toolset, which analyses the performance of the resulting distributed software system.

The Temporal Properties Analysis toolset includes a Schedulability Analyser and a Scheduler Simulator, based on those available in HRT-HoodNICE. They have been enhanced to provide a more precise and realistic analysis (by taking into account the concept of thread offsets) and to cope with the specific needs of a redundant fault-tolerant architecture (by allowing the analysis of the interactions over the ICN).

A further result is that, on the basis of the real-time models produced by the verification tools, the critical interactions among software functions on different channels are scheduled in a deterministic way. The transfer slots allocated to them and a set of pre-defined exchange tables are produced automatically.

3.4: Fault injection

Fault injection has been considered in GUARDS as an experimental verification activity to be carried on prototype instances. Its main objectives are: a) to complement the formal verification of GUARDS mechanisms by overcoming some of the behavioural and structural abstractions made, especially regarding the failure mode assumptions, and b) to support the development of GUARDS instances by assessing its overall behaviour in the presence of faults, in particular by estimating coverage and latency figures for the built-in error detection mechanisms [4].

Both for cost-effectiveness and flexibility, the fault injection environment is based on the software-implemented fault injection (SWIFI) technique [51]. Although available tools could have been used — albeit with some extensions — a specific fault injection toolset (FITS) is being developed. Such a toolset is made available to support end-users in the development of specific instances of the generic architecture.

Besides injecting specific fault/error types, FITS allows injection to be synchronised with the target system by monitoring trigger events. Of course, the observations depend on the targeted mechanisms. While it is primarily intended to inject on a single channel, observations are carried out on all channels.

3.5: Discussion

The GUARDS validation framework has been based on a set of advanced techniques that cover all the aspects mentioned in section 2. However, it is pretty clear that the used validation techniques have been separately applied, with very little integration. The only relation linking different models (dependability models, formal models, HRT diagrams, code) are mostly constituted by written documents.

On the other hand, it could not be in the scope of the project to provide such a strict integration, which requires a much deeper research effort. One of the aims of the project was, certainly, to define a validation environment as a set of support tools, what can be called a "tool kit", rather than a really integrated environment, since it was not possible to create a common theoretical framework in which to unify the different modelling techniques adopted.

In this sense, the validation policy of GUARDS is not so different from the best current industrial practice, where different techniques are applied to different validation phases, at best linked by an appropriate requirement tracing mechanism, sometimes supported by a dedicated tool. In GUARDS, more advanced techniques and tools (but this means also academic, prototypal and less mature tools) were employed within an overall validation policy aimed to provide a wide validation coverage.
Really integrating the adopted techniques along a System Engineering approach requires a step further, that is working on the models underlying the different techniques.

4: The HIDE approach to integration of validation

One step towards integration of different validation techniques within a common framework is represented by the ESPRIT Project 27493, HIDE (High-level Integrated Design Environment for Dependability) [47]. The main objective of HIDE is the development of an integrated environment for system design via the Unified Modelling Language (UML). The step forward which allows a more integrated validation is represented by the choice of using a semi-formal design description language (UML) for describing the entire systems under design. This allows to develop models for analyses which can be suited to the different properties of interest. Obviously the design language has to offer mechanisms for decomposing the system into manageable parts. Once the approach proposed within HIDE has been fully implemented it can be used for the validation of systems like GUARDS, provided the effort to describe the entire system with UML is undertaken.

The UML is a standard modelling language. It may be used for visualising, specifying, constructing and documenting several aspects of - or views on - systems. It is based on the object-oriented paradigm and it is heavily graphical in its nature. Different diagrams are used in order to describe different views on a system. For instance, Class Diagrams show sets of classes and their relationships thus addressing the static design view on systems. On the other hand, Statechart Diagrams show state machines thus addressing the dynamic behaviour of systems and their components.

It is outside the scope of this paper to further analyse the UML. We refer the interested reader to [41, 71].

A key point of the HIDE philosophy is to assist the practitioner designer by offering her/him basically a UML interface while providing translations from such a notation to mathematical models suitable for different kinds of validation. This way, the designer can easily validate her/his design since the bulk of the validation technicalities are hidden to her/him. Automatic or semi-automatic translations into the individual mathematical models for the particular validation, or validations, of concern will take care of such technicalities. The results of such validations will be presented back to the designer in the form of annotations in the original UML model.

Usually, each translation focuses on those components of the UML model which are relevant for the validation of concern, possibly enriched with some external information needed for the validation itself. Whenever possible, such enrichment is itself modelled using UML. In the following we shall briefly describe a couple of translations defined within the context of HIDE, as examples of the overall approach.

A first translation maps UML Structural Diagrams to Timed and Generalised Stochastic Petri Nets for dependability assessment [12].

Dependability modelling and analysis of complex systems consisting of a large number of components poses formidable problems. The most important is complexity. The existing tools are not able to deal with the state explosion problems, which plague big size models. To master complexity, a modelling methodology is needed so that only the relevant aspects are detailed, still enabling numerical results to be computable. A feasible strategy to cope with complexity considers starting from simple models, and making them more and more complex and detailed by including refinements of those parts of the system which are reputed to be relevant.

The translation from UML structural diagrams to timed Petri nets in order to keep small the size of the models, tries to capture only the features relevant to dependability leaving aside all other information. It
allows a less detailed but system-wide representation of the dependability characteristics of the analysed systems.

- provides early, preliminary evaluations of the system dependability during the early phases of the design. This way, a designer can easily verify whether the system that is being built satisfies predefined requirements on dependability attributes.

- deals with various levels of detail, ranging from very preliminary abstract UML descriptions, up to the refined specifications of the last design phases. UML higher level models, (structural diagrams) are available before the detailed, low levels ones. The analysis on these rough models provides indications about the critical parts of the system which require a more detailed representation. In addition, by using well defined interfaces, such models can be augmented by inserting more detailed information coming from refined UML models of the identified critical parts of the system. These might be provided by other transformations dealing with UML behavioural and communication diagrams.

It is important to point out that not only the UML diagrams that form the input of such transformation do not have a formal semantics, but also the specification this set provides might be incomplete or ambiguous. Therefore, a “formalization” of the transformation in the sense of formal correctness cannot be provided.

The second translation maps a subset of UML Statechart Diagrams to Kripke Structures (or in general transition systems) for formal verification of functional properties [59].

This translation defines a reference formal operational semantics for UML Statechart Diagrams within HIDE. Formal semantics are obviously necessary whenever formal verification is at issue: they are a necessary prerequisite for any sensible formal verification or analysis, which is the ultimate goal of the translation. In particular, the Kripke Structure resulting from the translation of a Statechart Diagram can be conveniently used as a basis for model checking. To that purpose it is of course necessary to specify the requirements against which the model has to be checked. Such requirements are usually expressed by a temporal logic formula or by another Kripke Structure or automaton. In the first case the formula is not part of the UML model, since the UML does not provide an explicit notation for temporal logic. In the second case, the requirement can be expressed again as a (simple) Statechart Diagram and its resulting semantics be used for model checking the (semantics) of the original Statechart Diagram.

A nice aspect of the translation definition proposed in [59] is that it is parametric in aspects which are not (yet) completely defined for UML. In particular, parametricity of the semantics definition w.r.t. transition priorities makes it suitable for describing the behaviour of systems under different priority schemes. A different approach for the definition of a semantics of UML Statechart Diagrams and for their model checking has been proposed in [18, 82].

5: Discussion

A first, maybe obvious, benefit of the HIDE approach is integration as such. Integration brings homogeneity and uniformity which, in turn, help in managing complexity. The possibility of having a unique model of the system to which reference can be done during different phases of system design certainly helps the designers in dealing with all technical aspects of such a design, and also with documentation issues. This is specially true when the modelling technique allows designers to define different views on the system, as is the case with the UML. All this is moreover beneficial because it makes it easier to compare different designs, and thus to evaluate different alternative approaches to the design of a system. HIDE goes a step further in the sense of ’opening further views' on the system, in the form of functions to proper validation domains.

Another advantage of the HIDE approach is the fact that the typical designer is not necessarily concerned with all the mathematical issues on which a particular validation approach is based,
neither with the technicalities of the related validation tool(s). On the other hand, such a driving philosophy of HIDE, although attractive in theory, still needs experimental evidence.

Moreover, the use of an automatic (or semi-automatic) translation may suffer from a classical problem, i.e. efficiency; it is extremely difficult to automatically generate validation models which are more efficient, in terms of size, than those generated manually, by skilled experts. One way to tackle this problem is to use powerful optimisation techniques in the representation of validation models. In the case of formal verification it is worth mentioning the already cited BDD technology and the various compression techniques described in [49].

Another issue is the level of integration. In the HIDE experience, in the end, such a level is not very high: different translations usually work on different kinds of UML diagrams, i.e. on different views, and there is not much integration between different mappings. This alone is not enough. The level of integration should be increased by allowing also the specification of functional as well as non-functional (e.g. quantitative) aspects within the same kind of diagrams in order to derive integrated models for different validations. An example of this could be the possibility of annotating Statechart Diagrams with deterministic-time and/or stochastic-time and/or probability information and then translate them into enriched semantic models like timed and/or stochastic and/or probabilistic automata.

Similar work has already been done in the context of other formalisms for the specification and verification of systems. Notable examples are Petri Nets and their quantitative extensions mentioned in Sect. 2.2, and Process Algebras and their timed, probabilistic and stochastic extensions [16, 17, 33, 34, 45, 46, 48, 65].

The possibility of having both functional and quantitative aspects of (a critical part of) a system within the very same notation is extremely interesting and convenient since it fills the gap which usually separates formal verification from quantitative validation. Usually, these two aspects of dependability assessment make reference to completely different models and there is no proven guarantee that such models are consistent in some sense. The use of enriched notations equipped with enriched semantic models like those mentioned above may help in providing rigorous definitions of what consistency means and related formal proofs. The advantage of applying such an approach to Statechart Diagrams, comes from the fact that Statechart Diagrams naturally offer simple and effective abstraction as well as composition mechanisms. Both features are extremely useful or even essential for clean and manageable system specifications, specially when non-functional aspects are considered.

Compositionality allows the time/probability information to be assigned locally to the components of concern, leaving the composition of such information to the semantics definition, i.e. to the translation function. The opposite approach, namely decorating with time/probability information the semantic model is impractical for systems of reasonable size, because of the monolithic nature of such a model. However compositionality appears to be a property very hard to achieve in the context of model based dependability validation. In fact, Markovian models can be composed just by connecting sub models but a hierarchical composition results in losing the Markovian property. In order to achieve hierarchical compositionality it is necessary to resort to more powerful formalisms such as Markov regenerative processes (see Section 2.2) which require the development of techniques to limit and control state explosion and of new efficient tools.

Abstraction is even more important. As stated above, it is simply unthinkable to have a detailed model of the system behaviour enriched with quantitative information because of its prohibitive size. It is then important to be able to produce abstractions of components of the systems in such a way that they are correct with respect to behavioural semantics and preserve the non-functional properties of such components. This way, one can specify different (critical) components separately, translate them into the enriched semantic models, build conservative abstractions of those models and compose them with the (abstract) models of other components.
The use of proper behavioural relations developed in the context of process algebra and automata theory, like bisimulation and its quantitative extensions, can be useful for formally dealing with such abstractions. Additionally, proper techniques, like abstract interpretation may be of great help [19, 31, 32, 36].

The problem of relating different views of the same system has been addressed also in the work done about the formalization of consistency between different viewpoints of the Open Distributed Processing computational model [15].

References


Appendix A. The list of “Pragmatic Principles” of system engineering.

A). Know the problem, the customer, and the consumer

1. Become the "customer/consumer advocate/surrogate" throughout development and fielding of the solution.
2. Begin with a validated customer (buyer) need - the problem.
3. State the problem in solution-independent terms.
4. Don’t assume that the original statement of the problem is necessarily the best, or even the right one.
5. When confronted with the customer's need, consider what smaller objective(s) is/are key to satisfying the need, and from what larger purpose or mission the need derives; that is, find at the beginning the right level of problem to solve.
6. Determine customer priorities (performance, cost, schedule, risk, etc.).
7. Probe the customer for:
   • new product ideas
   • product problem/shortfall identification
   • problem fixes
8. Work with the customer to identify the consumer (user) groups that will be affected by the system.
9. Use a systematic method for identifying the needs and solution preferences of each consumer group.
10. Don’t depend on written specifications and statements of work. Face to face sessions with the different customer/consumer groups are necessary.
11. State as much of the each need in quantified terms as possible. However, important needs for which no accurate or quantified measure exists, still must be explicitly addressed.
12. Clarify each need by identifying the power and limitations of current and projected technology relative to the customer's larger purpose, the environment, and ways of doing business.

B). Use effectiveness criteria based on needs to make system decisions

1. Select criteria that have demonstrable links to customer/consumer needs and system requirements.
   • Operational criteria: mission success, technical performance.
   • Program criteria: cost, schedule, quality, risk.
   • ILS criteria: failure rate, maintainability, serviceability.
2. Maintain a "need based" balance among the often conflicting criteria.
3. Select criteria that are measurable (objective and quantifiable) and express them in well known, easily understood units. However, important criteria for which no measure seems to exist still must be explicitly addressed.
4. Use trade-offs to show the customer the performance, cost, schedule, and risk impacts of requirements and solutions variations.
5. Whenever possible, use simulation and experimental design to perform trade-offs as methods that rely heavily on "engineering judgement" rating scales are more subject to bias and error.
6. Have the customer make all value judgements in trade-offs.
7. Allow the customer to modify requirements and participate in developing the solution based on the trade-offs.
C). Establish and manage requirements

1. Identify and distinguish between specified (fundamental or essential), allocated, implied and derived requirements.
2. Carry analysis and synthesis to at least one level broader and deeper than seems necessary before settling on requirements and solutions at any given level. (Top-down is a better recording technique than it is an analysis or synthesis technique.)
3. Write a rationale for each requirement. The attempt to write a rationale for "requirement" often uncovers the real requirement.
4. Ensure the customer and consumer understand and accept all the requirements.
5. Explicitly identify and control all the external interfaces the system will have - signal, data, power, mechanical, parasitic, etc. Do the same for all the internal interfaces created by the solution.
6. Negotiate interfaces with affected engineering staff on both sides of each interface and get written agreement by the two parties before the customer approves the interface documentation.
7. Document all requirements interpretations in writing. Don't count on verbal agreements to stand the test of time.
8. Plan for the inevitable need to correct and change requirements as insight into the need and the "best" solution grows during development.
9. Be careful of new fundamental requirements coming in after the program is underway. They invariably have a larger impact than is obvious.
10. Maintain requirements traceability.

D). Identify and assess alternatives so as to converge on a solution

1. Take the time to innovate by generating a wide range of alternative solutions to satisfy the need. (A common mistake is to converge a "comfortable design" concept too early because of time constraints.)
   - Consideration of seemingly bizarre alternatives often yields additional insight into the requirements and provides a reasonableness check for trade-off criteria and weights.
   - Include the "do nothing solution" in the system level solution trade-off to provide a measure of the value-add the new system will bring the customer/consumers.
2. Use a systematic architecture/design method.
   - Abstract the requirements to identify the essential design problems.
   - Establish functional structures.
   - Search for solution principles to fulfill the sub-functions.
   - Combine the solution principles to fulfill the overall functions.
   - Select suitable combinations.
   - Firm combinations into conceptual alternatives.
3. Evaluate each alternative against the requirements and the effectiveness criteria. Determine the alternative that provides the best weighted value combining:
   - effective
   - efficient
   - safe
• reliable
• producible
• testable
• maintainable
• easiest to learn

4. Elaborate the customer's top-level concept of operations to show how the consumers will use each solution alternative to satisfy the consumers' and the customer's needs. This detailed concept of operations must be reflective of the design aspects of the system's operation.

E). Verify and validate requirements and solution performance

1. Quality must be designed in, it cannot be tested in.
2. Use preplanned peer reviews and inspections.
3. Prototype critical elements.
4. Use models to demonstrate feasibility before bending metal and writing code.
5. Explicitly identify and sanity check all model assumptions.
6. Work the critical and controversial requirements and design areas first.
7. Plan the verification and validation for every requirement.
8. Know the expected results before testing.

F). Maintain the integrity of the system

1. Maintain a systems engineering presence throughout the program (even though SE staff starts to drop off after PDR and more after CDR) to provide technical oversight of the ongoing design process and to resolve requirements/technical issues that invariably arise, including resolution of test discrepancies/anomalies.
2. Prevent process and product contamination.
3. Ensure the system design meets the requirements, satisfies the need, and reflects the voice of the customer.
4. Ensure the requirements address not only the operational objectives but all the life-cycle objectives for the system.

G). Use an articulated and documented process

1. Start with established principles - avoid reinventing the wheel and really learn from "lessons learned" investigations.
2. Use the principles to develop a process tailored to the need, the system, the customer, and the development organization.
3. Use the process consistently across the program.
4. Train the development staff in the process and its application - technical education is one key to productivity, quality, and cost reduction.
5. Use standardized analysis techniques, document formats, design review formats, etc. to reinforce the consistent application of the process.
6. Use readily available automated tools wherever appropriate.
7. Maintain process integrity but never let the process prevent the "best" solution from being discovered or used - do whatever it takes to build in product quality.
H). Manage against a plan

1. Use a "tasks are executed to produce useful work products" focus for the plan.
2. Prepare a plan that is success oriented, achievable, defendable, and cost-effective but which can absorb the changes that will come.
3. Have a contingency plan for each identified risk.
4. Develop a plan that reflects organizational commitment to systems engineering.
5. Look for and abolish fraction-of-a-job situations.
6. Perform each task according to the plan.
7. Change the plan as soon as experience shows a better way to do a task.
8. Remember: micromanagement is not planning.
9. Remember Dwight D. Eisenhower's words: "Plans are nothing. Planning is everything."