An Implementation and Verification Technique for Distributed Systems

Rainer Gerlich

BSSE System and Software Engineering, Auf dem Ruhbuehl 181,
D-88090 Immenstaad, Germany
gerlich@t-online.de

Abstract. The technique „ISG“ (Instantaneous System and Software Generation“)[1] combines implementation, verification and validiation of software by a coherent and automated development process. No human intervention is required between delivery of high-level system engineering information and reading of the evaluation report as produced by the automatically generated and executed software system. The automation is based on standardisation, organisation and rigorous formalisation of all the development stages. Consequently, the productivity is increased by about one to two orders of magnitudes when ISG is applied. A feedback is available within a range of minutes to an hour depending on the size of the system. Verification and validation of a system is significantly eased by excluding the engineer from errorprone development activities, by applying formal checks, by combining system specification and generation, and by providing detailed reporting capabilities. Although standards are applied, the ISG approach can fully cover the application area of distributed and/or real-time systems.

1 Introduction

Reasonable effort has been spent in the past to improve the software development process by methods and tools. However, all of the known approaches address only one or a few of the facettes of software development. This results in a partial optimisation only and neither does it speed up significantly the development process nor does it solve all of the problems related to verification and validation. Hence, most of the activities related to software development still need to be done manually, although a number of tools are on the market.

[1] The ideas and implementation details related to ISG are property of Dr. Rainer Gerlich BSSE System and Software Engineering. They are protected by international copyright © 1999 - 2000.All rights reserved

2. Survey on State-Of-The-Art and Its Evolution

2.1 Analysis of State-Of-The-Art Approaches

Object-oriented methods (OOM) put emphasis on reusable components. OOMs have been and still are successfully used to construct loosely-coupled, reusable components by applying abstract interfaces and classes. But classes are not sufficient to support a broad range of applications because structural differences, which may occur for even the same application type, cannot be covered in a simple manner which allows for complete automation. Human intervention is still required which is expensive and may introduce errors into the only partially reused components.

The „Unified Modelling Language“ (UML) [1] aims to provide a general concept for software development based on object-oriented ideas. It mainly addresses the generation process and nearly spares the verification and validation process. But UML still recommends manual execution of the various activities needed for software development, as suggested e.g. by „deployment diagrams“, and does not encourage a software engineer to more efficient approaches of software production based on automation.

Tools like Teamwork [2], Statemate [3], StP [4], ObjectGEODE [5], SDT [6] (this is a non-exhaustive list) concentrate more on the visualisation of the engineer’s ideas and still remain on the informal level, but do not necessarily guide him towards a feasible and efficient solution. Therefore a number of serious problems usually come up during the coding, testing and integration phases, i.e. during phases which are not (well) supported by such tools. It is a well-known fact that a large number of software projects fail, overdraw the budget and the schedule, because the problems were not identified early enough. Usually, problems related to performance, resources and system integration are underestimated and not well identifed during early development phases.

Performance analysis tools, such as SES/workbench [7] or OPNET [8] are very helpful to identify bottlenecks early enough. But they increase the overall effort, while their code does not directly contribute to the target implementation. It seems that such tools are mainly applied when performance is considered as a high risk for a project. Actually, their usage is not well accepted by software engineers and they are usually not recommended by standards as it is true for UML.

The role of verification and validation (V&V) in the lifecycle is treated very poorly so far. During the first phases priority is given to expression and visualisation of ideas on an informal level, but not to V&V. Even in the V-model the validation of the software system is planned only at the end of the lifecycle, when corrective actions are rather expensive or impossible.

Tools like ObjectGEODE, SDT and Statemate/Rhapsody encourage engineers to simulate a system-under-development already at an early stage, but they do not support a coherent transition from the simulation prototype to the final target version. Also, they only concentrate on behaviour, but they do not support timing and performance analysis very well. ObjectGEODE and SDT support code generation for real-time applications, but they do not support the verification and validation of the real-time properties like meeting deadlines.

Regarding verification a number of formal approaches are known like B [9], Z [10], RAISE [11], VDM [12] for algorithmic verification (also known as „formal methods“), and SDL [13] and Statecharts [14] for behavioural verification (so-called „semi-formal methods“), represented by ObjectGEOD and SDT which were already mentioned above.

Due to their complex mathematical notation the „formal methods“ are not widely applied so far. Also, a potential disadvantage is that a user needs to ask for whether a property is fulfilled or not, the tools themselves do not inform automatically about problems. Hence, a problem which is not known in advance may still remain hidden.

ObjectGEODE, SDT (based on SDL) and Statemate (based on StateCharts) take a mathematical approach to verify the correctness and completeness of a system’s behaviour. So SDL has been successfully applied to protocol validation.

However, when applying the verification procedures to practical systems including a number of components with non-trivial behaviour, they fail in practical cases due to state explosion [15, 16]. Filtering techniques which are often applied to reduce the system’s state space to be explored are helpful to understand what is going on in certain parts of the system, but they are not adequate for system verification and validation. Problems related to interdependencies may not be identified this way.

One reason why state explosion occurs is related to the purely mathematically-oriented verification approach based on the exploration of all possible combinations of states and data, and limited or missing capabilities (more adequate than filtering) to exclude what will not occur in practice. Focusing on the relevant states is needed similar e.g. by data Flow Analysis“, an optimisation technique which is already applied in the area of compiler design, or by a formal prove which does not require exploration of the full state space.

ROOM [17], a method for Real-time Object-Oriented Modelling, and the related tool ObjectTime [18] address early validation of distributed systems, but also mainly concentrate on system decomposition, interfaces and the behavioural aspects, while timing and performance problems are left unsolved.

Synchronous languages, such as Lustre [19] and related tools such as Scade [20] simplify the verification problem by making rigorous assumptions on the timing and order of execution, so that verification can succeed in case of synchronous systems. The verification scheme can also be applied to distributed control systems [21], but not to other types of systems.

To summarise: a large number of methods and tools exist which provide good support for the application area and the lifecycle phase they have been built for. But they all lack features which are needed to make software development significantly more efficient and less risky. They do neither cover the full lifecycle nor every aspect which has an impact on the success of software development. As each method and tool provides a very specific solution only within a limited domain, an overall optimisation cannot be achieved by combining such methods and tools.

This does not mean that the methods and tools are not useful a priori. What is discussed here is their contribution to an optimised software development process, and from this perspective a number of problems have been identified. Each of the mentioned tools has contributed to improve software development, but so far an overall optimisation could not be achieved, which allowed a transtion from „handcrafted“ to „industrial“ software development.

1.2 Towards a Coherent and Automated Approach

In a first iteration the integration of complementary existing tool towards a more complete tool environment was considered as a potential solution to the problems identified above. During the OMBSIM project [22], which was executed for ESA, ObjectGEODE and SES/workbench were coupled to cover both, behavioural and performance verification. Although rather successful, some drawbacks were identified which prevented a broad usage: (1) the needed investment was doubled, a strong argument against such a solution in case of small or medium-sized projects, (2) maintenance was difficult because of diverging evolution of both tools, (3) the observed effort reduction only occured for a small percentage of the whole application, and (4) the chosen appproach increased the state space significantly.

Nevertheless, this approach allowed to experiment, and it was applied to a number of projects, improving it stepwise. Finally, as a self-standing solution ISG is now available, which is a coherent tool environment supporting the features which are needed to reduce risks and costs and to make software development more efficient by one to two orders of magnitude for such parts of a system to which it is applied.

By introducing standards [23], by identifying construction rules, and by manually trying the construction of a system by the standard procedures, the ISG approach was prototyped, before an automated process model could be defined. Now, ISG completely automates the generation process and combines system generation with handling of distribution, monitoring and reporting, verification and validation.

Apart from the automation aspect, reasonable effort has been spent to succeed with verification. Firstly, the problem of system verification has been divided into two principal steps:

1. verification of behaviour and performance, and
2. verification of functionality and algorithms.

This separation of the verification problem also implies the separation of software development into two principal parts: the concurrent part related to step (1), also dealing with distribution, and the sequential part related to step (2). This approach eases incremental development, because it decouples the concurrent part from the sequential part. In consequence, both parts can be refined separately.

Within ISG the concurrent parts form a framework with drawers into which the sequential parts can be plugged-in. This allows independent (pre-)verification of the sequential code and reduces significantly the system’s statespace because most of the data do no longer contribute to the behavioural state space.

Step (1) is based on a formal description of behaviour by Finite State Machines (FSM) (as known from SDL) together with a formal definition of performance properties and constraints by a formalised input scheme.

For step (2) formal methods like the ones already mentioned above may be applied independently. Also, the usual test means may be used. As another alternative such software also may be generated and verified from formal inputs provided by datasheets as supported by ISG [24,25].

3. The ISG Implementation Technique

To break a system down into its components, to describe the data flow and its behaviour, ISG follows OOM regarding interfaces and encapsulation, and SDL regarding formal definition of behaviour and data exchange. However, to allow for complete automation and to master state explosion, ISG introduces new concepts. Regarding efficiency the most important feature is that literals and figures are needed only to define (major parts of) a distributed / real-time system [24,25]. Even an engineer not being very familiar e.g. with real-time programming, communication protocols or programming languages, can easily create a distributed software system which is executable and immediately delivers a feedback.

The user inputs are mixed with the ISG files and both are subject of the generation process as shown by Fig. 1.

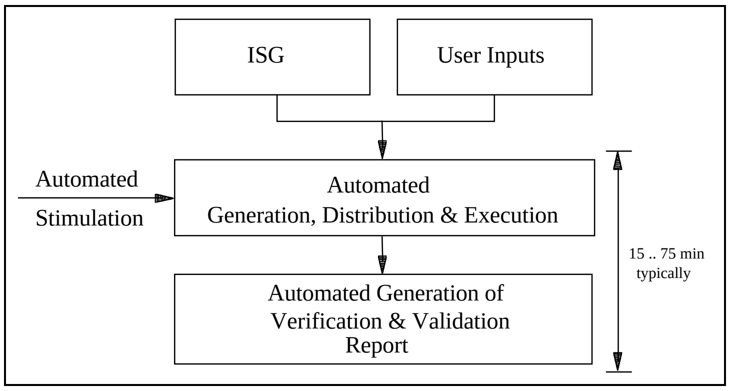
ISG
User Inputs
Automated Stimulation
Automated Generation, Distribution & Execution
Automated Generation of Verification & Validation Report
15 .. 75 min typically

Fig. 1. System Synthesis by ISG

Having generated and partitioned the software according to the user directives, the software is distributed across the network and executed by ISG. A network may consist of a number of heterogeneous platforms, i.e. operating systems (OS) like Solaris, Linux and VxWorks and processors like Intel and Sparc. To run a certain part of the system on another platform just three literals of the user input need to be changed related to the OS, the processor type and the IP network address.

During execution information is collected by ISG according to the automated instrumentation as defined by the user. Such data are evaluated for the verification and validation (V&V) report.

Tests can be derived automatically to stimulate the system due to the formal system definition. Such tests may cover stress testing and fault injection. The procedure from delivery of the user inputs until provision of the evaluation report may take between 15 and 75 minutes depending strongly on the number of process types (about 1..2 minutes each on an UltraSparc I/143) and sligthly on the other parameters like states, data inputs, topology and the set of external inputs. The given upper limit of 75 minutes applies to about 40 process types and about 150 external comamnds [24].

More details about the generation process are given by Fig. 2. The user inputs are divided into three principal parts: (1) the definition of the concurrent and distributed elements, (2) the configuration options and (3) the sequential code.

The system is defined by means of literals identifiying the processes, the network and the data flow, and by figures related to timing, scheduling, CPU consumption and the amount of data. By configuration options implementation features can be selected, the mapping onto the platforms and the interfaces to the environment are specified, the degree of instrumentation and reporting and the type of testing are controlled.

The ratio between user-provided inputs and the ISG generated output is in the range of about 100 (as measured for the MSL project [24]). This indicates the high saving of effort and time. As little effort is needed only to create a version of a system, a number of iterations may be executed to find the optimum implementation. Having a rough idea only a user may already start creating an executable system, and then incrementally refine its definition, always getting a feedback from the real system.

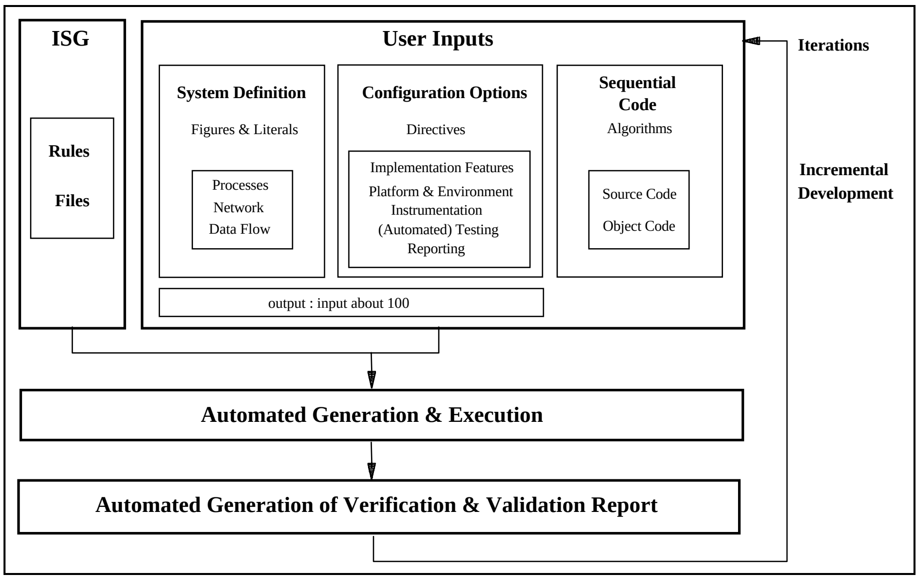

Fig. 2. System Definition and Generation in More Detail

Fig. 3 shows the software parts and the elements which are needed to build and verify the complete application. The environment complements the system by components which are needed during development and testing. Such components may represent hardware or external interfaces. The test software stimulates the system. From the ISG point of view there is no difference between the system software, its environment and the test software. The environmental and test software can be built by taking the same elements as used for the system software. E.g. the part to handle external commands is generated automatically and then added to the user inputs [24].

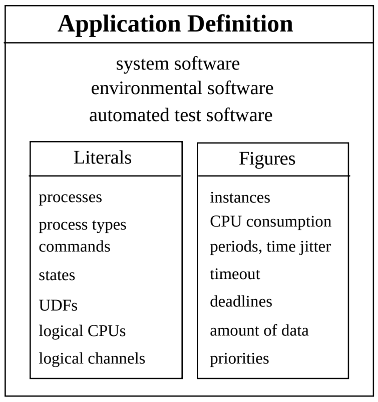

Fig. 3. Elements of the Application

The top-level entity to structure a system is the *process*. A process is derived from a process type. A process type defines a template for a set of software components having all the same behavioural and performance properties. Hence, a process is a physical entity which is derived from a process type by instanciation. It gets an *instance* number for unique identification.

Processes are communicating by exchange of data (messages in the sense of OOM). For the data exchange a standard format [21] is used. On reception of a message the receiving process needs to know which data have arrived and what to do with the data. The *command* which is included in the message provides the required information. A *priority* is assigned to a command indicating the urgency of its processing at the receiver's site. For analysis of the data traffic the *amount of data* associated with an outgoing command has to be given by a range.

A number of *states* may exist for a process reflecting that a process' reaction may depend on certain conditions. The states define a „Finite State Machine" (FSM) which drives the behaviour of a process. A *state transition* is an activity which is executed between an initial state and a final state. A number of atomic actions may be associated with such a state transition. Each such action is executed according to the scheme *„input (incoming command) - processing - output (outgoing command)*".

The input is processed by a „user-defined function" (UDF) which is a part of the sequential code plugged into the concurrent skeleton. After processing, usually another command is issued.

A *logical CPU* is assigned to each instance of a process and the (estimated) *consumption of CPU time* (in terms of cycles of the logical CPU) needs to be given for each processing action. For real-time processing figures like periods, timeout values and deadlines can be specified. Random jitter can be created for all timing features in order to allow to run the system under realistic and stress conditions.

Processes communicate via *logical channels*. This allows to hide the properties of a communication channel during an early phase for which only the principal data transfer capabilities are required. The exact properties, like the degree of fault tolerance, the type of the transfer medium or the bundling of several heterogeneous transfer media, may be introduced later when really needed.

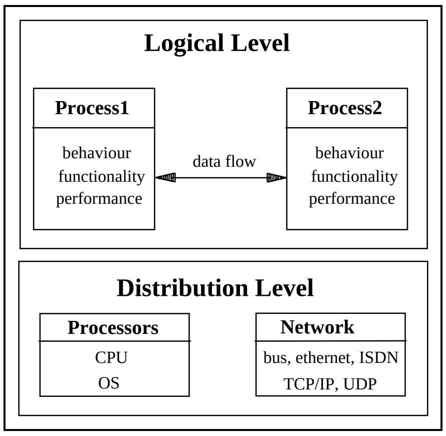

Fig. 4. ISG Logical Levels of System Definition

Fig. 4 shows the two principal levels as used by ISG to decouple the logical system level from the physical distribution level. This separation gives high flexibility for redefinition of the topology or the platforms, because the logical level is not effected when the distribution is modified, and vice versa the distribution level is not impacted when process internals or the logical communication channels are changed.

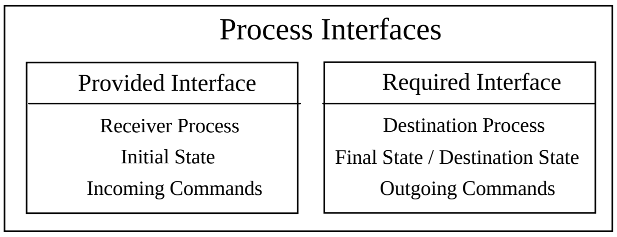

Fig. 5. Elements of the Process Intrerfaces

ISG uses an abstract, standardised interface for communication between processes as shown by Fig. 5:

– the *provide*d interface
  and
– the *required interface*.

This notation has been introduced by HOOD [26] for programming with Ada. In context of ISG the *provided interface* is what the receiver makes visible to the external world. The *required interface* is what the sender of the message wants to use. The formal definition of the two sides of communication allows to perform consistency checking and to limit the combinations of messages. What is part of a *required interface* must appear in the *provided interface* of the receiver. If not, the system definition is incomplete, the code is missing.Vice versa, what appears in a *provided interface* must be used somewhere in the system, otherwise it represents „dead code". ISG checks the user inputs for missing and dead code.

The elements of each interface type are:

– the *process*
  Either the receiving process (provided interface) or the process to which the outgoing command is sent (required interface). A process is identified by the process type and the instance number
– the *state*
  At the provided interface this is the initial state in which the process receives the incoming command At the required interface (related to the outgoing command) it is the state in which the receiver shall process the input
  If the destination process is identical with the receiving process, the destination state defines the final state, and a state transition for the receiving process is initiated.
  An exception handler has to be provided for each state to be able to process such commands which are not covered by the current state. This is represented by an incoming command named EXC<state>.
– the *command*
  At the provided interface this is the incoming command. At the required interface this is the outgoing command. Each comand is sent by a standard message format.

Consequently, the user inputs define

– the provided interface, i.e. receiving process, initial state, incoming command,
– the UDF (user-defined function) for data processing,
– the required interface, i.e. the destination process to which the outgoing command shall be sent and the desired state,
– the logical communication channel through which the outgoing command shall be issued, and
– some more figures related to performance as already mentioned.

In addition, directives are available to allow for scheduling, event generation and monitoring of timing constraints.

As described in more detail by [24,25] ISG reads the user inputs and

– generates the source code for the infrastructure of a distributed / real-time system,
– generates stubs for the UDFs, takes the UDFs as provided by the user or generates the UDFs itself,
– optionally instruments the source code to prepare it for monitoring and tracing
– stimulates the system for operational and stress trresting, and fault injection
– distributes the processes across the physical network and executes them (applying a handshake for synchronisation at system start up)

– collects the monitoring information and prepares an evaluation report (textual and graphical information).

ISG generates the source code and everything needed to get an executable system on Unix platforms (Solaris, Linux), distributes it and evaluates the results. Executables may be generated for the following platforms: Solaris, Linux and VxWorks operating systems and Intel (PC) and Sparc processors in each possible combination.

Each logical CPU is mapped onto one of the possible combinations of OS and processor type. It is even possible to map logical CPUs, belonging to the same system, onto different platforms. The set of currently available platforms may be extended on request. Only a few OS primitives are needed so that ISG can easily be targeted to other operating systems and procerssors.

4. The ISG Verification and Validation Technique

Regarding verification and validation ISG addresses behaviour, performance and functionality and is capable of providing the needed environment and stimuli.

Verification means „to check if the system is built right", validation means to check „if the right system is built". Verification as done by ISG not only addresses the system definition, but everything else related to the generation and distribution process, e.g. updating of data depending on other information.

When ISG accepts all the user inputs, they represent an executable system and ISG does the conversion. When the ISG evaluation report does not indicate errors, the actual version of the system can be considered as verified from a behavioural and performance point of view. This is different from other tools for which acceptance of user inputs does not imply that correct and immediately executable code can be generated.

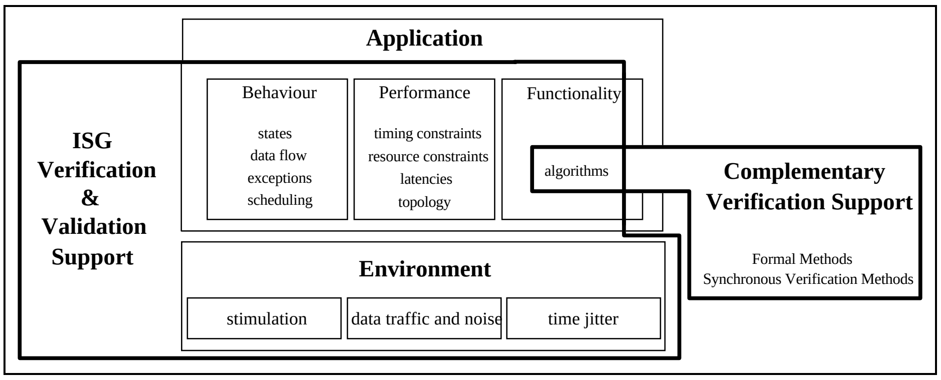

Fig. 6. ISG Support for Verification and Validation

ISG concentrates on verification of the provided and required interfaces and everything related to that, aiming to achieve full coverage of behaviour and performance

(Fig. 6). Verification of the UDFs may be done separately by specific test environments, if needed and appropriate. E.g. if code of a UDF is generated by another tool like Scade, this code may already be verified by the test and verification means the tool provides. However, if UDFs are generated in the context of ISG using formalised inputs like in case of [25], then their verification is supported by ISG as well.

Validation of a system is achieved by analysing the provided information about data flow, timing, performance, exception handling and comparing it with what is expected. i.e. by checking if the system - as defined by the engineer - really does what it ought to do. Hence, validation means to compare between what the engineer believed and intended to define and what he really defined.

In order to master state explosion ISG applies a „goal-oriented" verification approach. This means ISG takes a system engineering point of view rather than a purely mathematical respectively combinatorial one. E.g. in case of SDL the system's state space as the product of all the process' state spaces needs to be explored which neither can be exploited in practical cases nor really all are executed by the system during normal operation.

From a practical point of view missing the verification goal due to state explosion is very dissatisfying. In fact, due to allowing a high degree of freedom in system implementation, verification fails. Therefore ISG takes an alternative approach:

1. It separates behavioural and performance V&V from functional V&V as already mentioned.
2. It defines a different goal for V&V which is more related to the system specification and its automated construction: the system's state space corresponds exactly to what the engineer defined. So no exploration is needed to analyse if it fits with the specified behaviour or not.

By its construction rules ISG prevents that inconsistent sequences of messages may occur. This is different from an implementation using SDL. For SDL the desired and non-desired behaviour of a system are specified by Message Sequence Charts (MSC) [27]. The answer to this specification is the implementation by processes and FSMs in SDL. Then for verification a (usually) huge state space needs to be exploited by exhaustive simulation and a large number of message sequences is generated. Finally, the specified message sequences are compared with the generated ones. This complex procedure which needs a lot of computing resources is a consequence of separation of specification and implementation.

When they are not related to each other the big task of exploration is required because each combination of process states needs to be checked. For ISG specification and implementation are related by the provided and required interfaces and therefore an extensive check is not required.

When defining the provided interfaces and connecting them by the required interfaces ISG constraints the set of possible messages sequences due to the following rules:

1. For each outgoing command the state (destination state) must be given in which the receiving process shall handle it.
2. An exception handler needs to be defined for each of the states which is capable to handle unexpected inputs.

Therefore only such sequences are produced which correspond to the system's definition. As no forbidden sequences are generated, there is no need to exploit the state space as spawned by the product of the process' state spaces. It is sufficient to concentrate on the process state spaces, only. Hence, the total size is reduced from the product to the sum of the state spaces.

To exploit the state spaces of the processes ISG identifies

1. the commands sent from the external world to the system which define the „external required interface"
2. the set of commands which are part of the „provided interfaces" which define the „system's internal provided interface".

Now, verification is considered as successful when the required external and the provided internal interfaces fit together. When is this achieved?

From the user's (operator's) point of view the system is accepted when it executes all the external commands correctly. Looking into the system's internals, the correct execution of all external commands should correspond to a full coverage of states and internal commands. If it is not achieved the system provides more than expected. If it could not process all the external inputs, then the system's definition is not complete.

Consequently, the goal of verification in the context of ISG is to achieve compliance between the „external required interface" and the „system's internal provided interface". This defines a well-known set of test cases to which the system has to be exposed.

ISG supports mapping of the external commands onto the internal commands. External commands can be expressed as timed-sequences of internal commands by directly relating them with the provided interfaces. This way it is impossible that an external command cannot be processed.

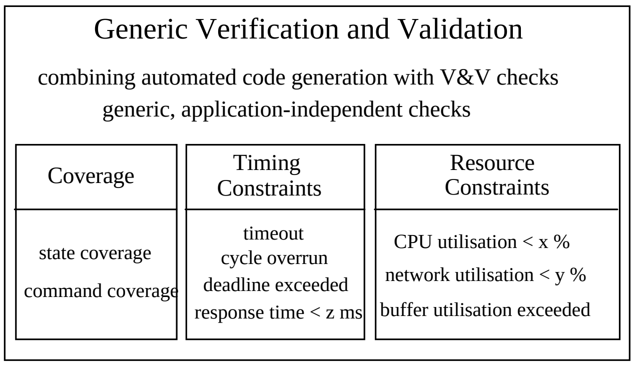
Generic Verification and Validation
combining automated code generation with V&V checks
generic, application-independent checks
Coverage
Timing Constraints
Resource Constraints
state coverage
command coverage
timeout
cycle overrun
deadline exceeded
response time < z ms
CPU utilisation < x %
network utilisation < y %
buffer utilisation exceeded

Fig. 7. Capabilities for Generic Verification

Regarding automation of verification, ISG inserts into the code generic checks which are valid for every application. They are providing answers without being asked for. This is a consequence of the ISG approach which combines system definition and code generation. Fig. 7 gives some examples of such generic checks.

As ISG knows about the states and the commands, it can check their coverage, report non-covered states and commands, and can automatically conclude if the full coverage is achieved or not. The essential point is that the check can be expressed in a normalised form and the result of the check is independent of the application, it is just e.g 100%. The same conclusion is valid for the checks of timing constraints and resource utilisation.

4. ISG in the Context of CRISYS

The development of the ISG approach has been funded in part by the ESPRIT project EP 25514 CRISYS [28]. It will be used for two exercises in the context of CRISYS:

1. an application related to the back-up power supply of a nuclear power plant,
2. a mail sorting and distribution application.

In both cases ISG provides the environment, acts as an integration platform for the synchronous components and provides reporting and evaluation capabilities (Fig. 8).

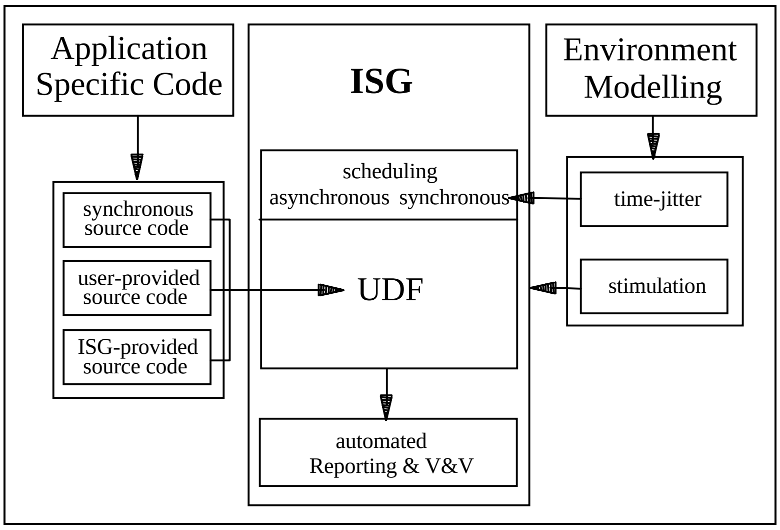

Fig. 8. ISG and an Application

Application (1) is a distributed control system which only includes synchronous components. These components are generated by Scade and the code of each component is plugged into the ISG skeleton as a UDF. Verification of the synchronous components is based on the capabilities of Scade and the theory as described by [21].

Application (2) consists of a number of synchronous components such as motor drives, light barriers, belts and feeders, and components with asynchronous, event-driven behaviour such as generation of mail pieces and pattern recognition.

ISG schedules the synchronous components and provides independent clocks (with time jitter) for the distributed synchronous components. The asynchronous components of the mail sorting application are defined as ISG processes.

5. Conclusions

An analysis of state-of-the-art techniques has been performed regarding software development and its verification and validation. It has been pointed out that for system verification and validation all features need to be covered already at an early development stage: behaviour, performance and functionality. The analysis of methods and tools showed that only limited support is provided regarding all such features over the full lifecycle. To overcome this weakness experiments with integration of existing tools were performed which guided towards a higher degree of automation and better support of verification and validation right from the beginning of the development process. As a current result the ISG toolset provides an efficient solution for system development by automated system generation and verification.

Having identified the principal problems which make verification difficult, ISG applies a verification concept which significantly reduces the state space which needs to be explored.

Due to automation of the development process ISG can instrument the source code such that the information as needed for verification can be easily derived in a generic, application-independent manner.

At the beginning ISG and its predecessors were used to provide a system's infrastructure and to support verification and validation of such parts. When applying ISG to the MSL project, it was identified that major parts of UDFs could be automatically generated as well. This experience will drive the future use and evolution of ISG to cover an increasing percentage of a system's software, because such parts which are not automatically generated by ISG drive the remaining overall effort.

ISG is in use for the MSL space project planned to fly on the International Space Station in 2001/2002. This proves the feasibility of the approach regarding complete automation of the software development process and verification and validation. It will be applied to two more applications in the area of nuclear power plants and mail sorting / automation confirming that ISG can be used for a wide range of applications.

References

1. UML, Unified Modelling Language, http://www.rational.com/uml
2. Teamwork, Sterling Software, Corporate Headquarters: 300 Crescent Court, Suite 1200; Dallas, Texas 75201, http://www.sterling.com
3. Statemate/Rhapsody, i-Logix, Three Riverside Drive, Andover, MA 01810, info@ilogix.com
4. StP, Software Through Pictures, Aonix Corporate: 5040 Shoreham Place; San Diego, CA 92122, info@aonix.com

5. ObjectGEODE, Verilog, 52, Avenue Aristide Briand; Bagneux; 92220; France, verilog@verilog.fr
6. SDT, Telelogic, Headquarters: Box 4128; S-203 12 Malmoe; Sweden. Vising address: Kungsgatan 6, info@telelogic.se
7. SES/workbench, 4301 Westbank Dr., Bldg. A, Austin, TX 78746 USA, mktg@ses.com
8. OPNET, MIL3 Inc., 3400 International Drive, NW-Washington, DC 20008, USA
9. Abrial, J.-R.: The B Book - Assigning Programs to Meanings. Cambridge University Press, August 1996
10. J. Spivey: The Z Notation - A Reference Manual, Prentice Hall, 1989
11. The RAISE Specification Language, The RAISE Language Group, Prentice Hall, 1992
12. Cliff B. Jones: Systematic Software Development Using VDM, Prentice-Hall, 1990
13. ITU Z.100, Specification and Description Language, SDL, Geneve (1989)
14. D. Harel: Statecharts: A visual formalism for complex systems, Sci. of Comput. Prog., vol 8, (1987) 231-274
15. R.Gerlich:Tuning Development of Distributed Real-Time Systems with SDL and MSC: Current Experience and Future Issues, A. Cavalli, A.Sarma (edt.) SDL'97 Time for Testing, Elsevier, (1997) 85-100
16. R. Gerlich: Some Hints about How to Reduce Size of State Space when Modelling with SDL, http://home.t-online.de/home/gerlich
17. B.Selic, G. Gullekson, P.T. Ward: Real-Time Object-Oriented Modelling, John Wiley & Sons (1994)
18. ObjecTime, ObjecTime Limited,; 340 March Road, Suite 200, Kanata, Ontario, Canada K2K 2E4, sales@objectime.on.ca
19. N.Halbwachs: Lustre Language Reference Manual, V5 (1997)
20. SCADE tool, Verilog, 52, Avenue Aristide Briand; Bagneux; 92220; France
21. P.Caspi, The Quasi-Synchronous Approach to (critical) Distributed Control System Design, MOVEP'2k
22. OMBSIM (On-Board Mangement System Behavioural Simulation), ESTEC contract no. 10430/93/NL/FM(SC), Final Report Nov. 1995, Noordwijk, The Netherlands
23. R.Gerlich: Organising Incremental, Reusable and Automated Software Development, Eurospace Symposium DASIA'99 "Data Systems in Aerospace", May 17-21, 1999, Lisbon, Portugal ESA SP-447 (1999) 141-148
24. R.Gerlich, M.Birk, U.Brammer, M.Ziegler, K.Lattner: Automated Generation of Real-Time Software from Datasheet-based Inputs -The Process Model, the Platform and the Feedback from the MSL Project Activities, Eurospace Symposium DASIA'00 "Data Systems in Aerospace", May 22-26, 2000, Montreal, Canada, ESA (2000)
25. M.Birk, U.Brammer, M.Ziegler, K.Lattner, R.Gerlich: Software Development for the Material Science Laboratory on ISS by Automated Generation of Real-Time Software from Datasheet-based Inputs, Eurospace Symposium DASIA'00 "Data Systems in Aerospace", May 22-26, 2000, Montreal, Canada, ESA (2000)
26. HOOD, Hierarchical Object-Oriented Design Method,
HOOD Reference Manual Rel. 4, ftp://ftp.estec.esa.nl/pub/wm/wme/HOOD/HRM4.tar.gz
27. ITU Z.120, Message Sequence Charts (MSC), (1993) Helsinki
28. CRISYS (Critical Instrumentation and Control System) ESPRIT project EP 25514 (1997-2000)