BSSE System and Software Engineering

DASIA 2002

"Data Systems in Aerospace"

organised by Europsace

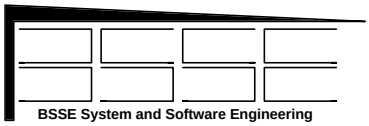
May 13 – 16, 2002

Dublin, Ireland

Rainer Gerlich

Benchmarks on Automated System and Software Generation

Higher Flexibility, Increased Productivity and Shorter Time-To-Market by ScaPable Software



Benchmarks on Automated System and Software Generation

Higher Flexibility, Increased Productivity and Shorter Time-To-Market by ScaPable Software

Rainer Gerlich

BSSE System and Software Engineering

Auf dem Ruhbuehl 181
D-88090 Immenstaad, Germany

Phone +49/7545/91.12.58
Mobile: +49/171/80.20.659
Fax +49/7545/91.12.40
e-mail: gerlich@t-online.de
URL: http://home.t-online.de/home/gerlich/

Abstract: "ScaPable" is an acronym derived from "scalable" and "portable". The attribute "scalable" indicates that specific application software can automatically be built from scratch and verified without writing any statement in a programming language like C, thereby covering a large variety of embedded and/or distributed applications. The term "portable" addresses the capability to automatically port parts of such an application from one physical node to another one - the processor and operating system type may change - only requiring the names of the nodes, their processor type and operating system. This way the infrastructure of an embedded / distributed system can be built just by provision of literals and figures which define the system interaction, communication, topology and performance. Moreover, dedicated application software like needed for on-board command handling, data acquisition and processing, and telemetry handling can be built from generic templates. The generation time range from less than one second up to about twenty minutes on a PC/Linux platform (800 MHz). By this extremely short generation time risks can be identified early because the executable application is immediately available for validation. A rough estimation shows that one hour of automated system and software generation is equivalent to about 5 .. 50 man years. Currently, about 50% of a typical space embedded system can be covered by the available automated approach. However, the more it is applied, the more can be covered by automation. A system is constructed by applying a formal transformation to the few information as delivered by the user. This approach is not limited to the space domain, although the first industrial application was a space project. Quite different domains can take advantage of such principles of system construction. This paper explains the approach, compares it with other approaches, and provides figures on productivity, duration of system generation and reliability.

---

1. INTRODUCTION

There have been big concerns about the problems associated with software development for a number of decades: too poor quality, too high costs, too long development time. Although every project manager urgently looks for a solution, the principal problem is still unsolved. Only little progress was made in the past.

In our opinion these problems exist because human beings do not have sensors to understand what the produced code really does. So we cannot see if something is wrong with software. This is different from products of daily life like cars, TVs, buildings etc.. If the motor of a car is missing or not well working, everybody recognises this. However, if in case of software a certain function is missing or is producing wrong results, it is a problem to identify such a defect. Therefore much effort and time is needed to build software of acceptable quality.

Chapter 2 analyses the current situation and concludes on a feasible solution based on scalable and portable software which helps to improve the situation. The related concept and tool has already been applied to an industrial project to the customer's full satisfaction and to further in-house projects at BSSE. More external applications are in preparation.

Chapter 3 describes the chosen approach and chapter 4 illustrates the principles of this appraoch by an example of daily life, a spreadsheet application.

Chapter 5 presents productivity figures and generation times. Finally, chapter 6 makes the conclusions.

2. THE CURRENT SITUATION

In the past the progress in software development was mainly made in support of code generation, but not in the area of verification and testing. Currently, front-ends exist by which an engineer can define the code on graphical level, from which the source code is automatically generated. However, the amplification factor is close to 1 in this case, i.e. nearly the full code still has to be defined by the engineer.

A number of methods came up which aim to support an engineer in describing and structuring of the software, and identification of potential inconsistencies. But no method really improved and speeded up the feedback from the developed system to the engineer, especially in giving him an immediate response from reality. Due to manual design and coding usually (for sophisticated applications) a number of months pass by until a first feedback from the executable system is received. As mentioned before this feedback is rather incomplete, especially regarding performance which bears the highest risks. At the time of the first response much effort has already been spent and it is probably difficult to move to another architecture if needed. Due to lack of time and money usually it is hardly possible to reach the envisaged goal.

Automated testing is becoming popular now, but the test objectives and test inputs still have to be derived manually from specification and design. This does not improve the situation regarding the first feedback, because even automatic testing requires manual coding before.

A number of environments already exist like Virtuoso [1], Rhapsody [2], ObjectGeode / SDT [3], which support an enigneer in building the real-time and process communication layer of an application. However, an application consists of much more than such layers. So a lot of work remains to be done manually, and the risks still exist until the full system is executable.

It was observed that use of a tool like ObjectGeode significantly increases the productivity [4]. However, the work which formed the base of such analysis was limited to some exercises [5,6,7], but not to a real project and to all parts of an application. Continuous internal benchmarking at BSSE showed that the productivity decreases from the initially observed high value (as given by [4]) towards the figures known for development in a high-order language like C [8]. We identified as reason the remaining much higher amount of software which is not supported by such environments. An application requires more than what is supported and this significantly compromises the productivity. Consequently, the advantage is the smaller the more software remains which is not covered and the more manual intervention is required. Our conclusion was that only by a coherent, self-consistent framework of automation the problems of productivity and quality can be solved.

From a theoretical point of view software reuse should improve the situation. However, in practice it does not for application software (while reuse is high for commerical software like operating systems). In fact, in thie application area the degree of reuse is very poor, and there have been a lot of discussions why it is like that. Certainly, one reason is that the properties and quality of software are poorly known. Even if software is reused by the same software engineer who established it, still risks exist due to the imperfectness of the software development environments [9].

There are activities, e.g. [10,11] which attempt to improve quality by better education and more training of engineers, better and possibly more detailed standards and documentation. However, this adds significant overhead and the amount of paperwork and formal procedures is increased. It is well known that the differential productivity decreases when the total effort and man power increase. Hence it is rather clear that continuous increase of manual effort and man power will at some point reach an upper limit beyond which an increase of such resources does not yield any increase of output and quality.

Our current conclusion is: human beings have very limited visibility on the properties of software and this significantly constraints the complexity which can well be handled. So increase of man power resources makes it worse. The gain in productivity is poor compared to the additionally needed effort, and the development time is not shortened at all.

However, in other domains higher complexity can be handled well. Manufacturing costs have decreased while quality increased. About fourty years ago the price of a TV was equivalent to about one man month of work. Today, the equivalent is about 10 man hours, yielding an increase of productivity of about 10, while quality is much better and the provided functionality and the system complexity increased dramatically. As for software, man does not have sensors for the signal flow in analog or digital integrated circuits. So how was it possible to improve development and manufacturing of TVs? The answer is: by automation. The increasing complexity is handled by automata and man is only concerned with a complexity level which can well be handled by human beings. As automata produce goods by following always the same manufacturing procedure, the quality is well defined and always the same. If something has to be improved, all goods produced after the improvement of the production process take benefit. This is the way how quality can continuously be improved. And it is the philosophy of ISO 900x and ECSS.

Now, why has this progress not already been applied in the software domain? First, it is argued that a software application is unique. When an application has been developed, no further manufacturing of copies is required. If more copies are needed just the files needs to be copied. So definition of a production process for the copies seems not to be required. However, this is a matter of orgnaisation. As shown below, software development can be organised such that similar production methods can be applied as for serious production of cars, TVs etc..

Second, it is believed that development of software is a very creative process. Hence, it cannot be performed by automata. However, looking in detail on how software is currently produced, it is like Edison was saying: 1% inspiration 99% transpiration. When developing software, experienced engineers apply - roughly spoken - some kind of reuse by deriving new code from similar code previously established. Of course, usually some modifications are needed for the actual application. Moreover, most of the activities and most of the "99% transpiration" result directly from the "1% inspiration" like coding, verification, validation and testing. However, this dependency is currently not well understood. The automated correlation of the inspiration work with the later transpiration work requires a good organisation so that a formal transformation from the inspiration input to the output can be described and the transpiration work can be done by automata. Again it is a matter of organisation to increase reuse by automated production.

At least one realisation of such an idea has already been available for a number of years, although the application domain is limited and quite different from the domain of embedded, distributed real-time systems. We are talking about spreadshet programs. Spreadsheet programs are providing a framework which automatically updates the calculation scheme according to user operations. If a user changes the structure of such a scheme, the calculation formulas are transformed automatically and the result remains always correct. The similarity between spreadsheets and the automated construction of ScaPable software by automata is discussed in detail by chapter 4.

3. GETTING SCAPABLE SOFTWARE BY THE AUTOMATED APPROACH "ASAP"

The efficiency of automation significantly depends on how much of the lifecycle can be covered. Even little human intervention may compromise productivity and quality. Therefore an automated approach should apply to the whole process chain. And such a concept needs to support a wide range of application domains.

The tools which already provide some degree of automation, suffer from the fact that they are open for a continuum of possible implementation architectures, which implies limitation of automation to such parts which are obviously generic.

The cornerstone of "Automated Software Production" (ASaP) [12] is to provide a generic solution for such parts which so far have been excluded because they were considered as too specific. ASaP takes one feasible solution from the continuous spectrum of possible solutions, but it remains general enough to support a wide range of applications. By focusing on a certain doamin only, ASaP opens the door for automation. This is just the opposite of what UML does. UML intends to cover all software domains, therefore it cannot take benefit from assumptions and related optimisation which are valid for certain domains only. Hence, the goal to cover the full software domain is in conflict with automation.

A scalable and portable generic solution has been defined by BSSE in 1999 and is called "Instantaneous System and Software Generation" (ISG) [13]. ISG generates the full infrastructure of a distributed and/or real-time system. It supports functional topics like message exchange, message processing, configuration of the desired topology, platform portability, and non-functional topics like fault tolerance, automated verification, instrumentation, evaluation of system properties and automated generation of the compilation and (distributed) run-time environment.

By ISG a complete application can automatically be built from literals and (performance) figures. The immediately executable system delivers a representative feedback on its properties. Whatever the system specification is in terms of processor types and nodes, processes, exchanged messages, communication channels, priorities, operating syxstems, the system will be built accordingly and automatically. Features such as time synchronisation between nodes and recovery actions in case of loss of nodes are supported as well. ISG follows immediately the instructions of the engineer (made on system level) and provides a correct result accordingly like a spreadsheet program does.

With ISG it is possible to execute a n-processor system on a subset of m physical processors with $1 \leq m \leq n$. Having completed testing for any such (sub)configuration, the system may be ported to anotehr sub-configuration of the full configuration of n processors without changing any source code and without any need for re-testing. This way the "virtual two-processor" version of a distributed on-board database was tested on a single physical procesor and ported to a real two-processor system later on - and it ran immediately [14]. It is even possible to test it on a single PC and port it to a two SPLC processing system without re-testing.

Regarding scalability, by ISG the distributed two-processor system of MSL [14] with 30 .. 40 process types (40 .. 50 process instances) and a mix of asynchronous and synchronous processing was derived from inputs provided by a

spreadsheet. Also, a sixteen-processor system was automatically built by ISG, a purely synchronous application with automated buffering of cyclic data, aiming to control the back-up power supply of a power plant. In latter case the code of the synchronous application was generated by another tool (Scade [15]) and plugged-in into the interface "boxes" of ISG. Hence, the infrastructure for quite different application types can automatically be generated.

The software as automatically produced by ISG provides the dedicated infrastructure of a real-time and/or distributed system. But it is not the end of the day because more software may be needed which cannot built this way.

What remains is software which does e.g. data acquisition and processing. Experience shows that a generic approach is possible as well, but its application domain is limited and not as broad as for ISG. Such automatically produced software may only cover the needs of space, but not such of automotive applications.

But even in such more limited cases ScaPable software still brings a big advantage: the generated code is immediately available and its quality is exactly known. Our experience is: the more the automated approach is applied, the easier it is to find an automated solution and the more efficient is the solution. The more exercises of automation have been performed, the less has to be developed for a new exercise.

This way the remaining part of software which cannot automatically be generated so far will continuously decrease and be limited to very specific functionality or to cases where the development costs of automated software are higher than the development of the dedicated software. However, such a trade-off between manual and automated software development has to be taken very careful, because our experience shows that the costs of the automated approach may lead to later cost savings, when the potential application domain is broadening. Consequently, a decision towards an automated approach and ScaPable software is always of advantage, even if the immediate return of investment is not directly visible.

4. SPREADSHEETS: AN EQUIVALENT APPROACH FROM A PRINCIPAL POINT OF VIEW

The support as provided by a spreadsheet tool is a good and easy-to-understand example for what the term "ScaPable Software" stands. All the issues on automation can easily be explained by a spreadsheet example.

In case of a spreadsheet the user only needs to insert numbers into the fields of the calculation scheme. After changing the contents of a field, he just needs to press a button to get the actual result. If he needs more or less columns or rows, he inserts or deletes such columns and rows, presses a button and he immediately gets the updated result, in textual or graphical manner.

A wide range of applications can be covered by spreadsheet software. It provides the infrastructure for calculation of the result: generic functions for calculation and correlation of the data fields.

To cover more specific functionality, macros can be used to cover what is missing, e.g. application specific conversion of data. Hence, macros correspond to the more specific automation software while the equivalent of the basic spreadsheet software is the ISG generator of a system's infrastructure.

But ISG does a lot more. Similarly to the capability of building a variety of user-defined calculation schemes by macros, ISG builds the structure of the processes and the system topology a user is asking for.

In case of spreadsheets interfaces exist into which a the user can plug-in his macros. Of course, the same mechanism can be applied to ISG. Then the next step of automation can be entered: the automated constrcution of macros, and so on. This will allow to continuously increase the percentage of automatically constructed software.

To succeed for ScaPable software ASaP takes two principle steps:

- The construction of the system ("spreadsheet") from a generic template which allows to put macros in.
  This is what ISG does: provision of the complete infrastructure of a distributed / real-time application.
- The construction of the specific macros which can be plugged into the generated "spreadsheet".
  This is what the part of ASaP does which is complementary to ISG. It addresses e.g. data processing and sequential software.

5. ACHIEVED RESULTS

Principal investigations on automated construction of software applications and the impact on productivity were started by BSSE in 1996. At that time the approach was called "EaSySim II" [16], it was a semi-automated approach which still required much effort to build the infrastructure of an application. In autumn 1999 a first version of ISG was delivered to the MSL project [14]. In autumn 2000 the ASaP software of MSL was finally delivered including an enhanced ISG

version and a number of on-board software packages automatically constructed from form sheets. Since that time the ISG and ASaP software have continuously been improved.

5.1 Automated Generation of Systems

In case of systems not only the operational software as specified by the user's inputs and the compilation and linking environment is generated, but every software which is needed to interface with the operating system for real-time scheduling and process communication, to distribute the process instances, to execute them and to collect and evaluate the results.

5.1.1 The MSL Project

For the MSL project the complete distributed real-time infrastructure (two SPLC nodes) has automatically been generated. This software runs on Sparc, SPLC and PC platforms with VxWorks [17] and Solaris [18] / Linux [19] in any combination of the OS and the processor types. The software covers scheduling, message exchange, command handling (including time-tagged (ground and on-board) command sequences), exception handling, node and time synchronisation.

This system is of type "DMS" (Data Management System). The complete data flow between the 40process types as defined for the study on MSL validation [14b] is shown by Fig. 5-1. This figure shall just give an idea on the sysatem's functionality, it is not intended to give details. It has to be mentioned that each process type is well structured by a FSM (Finite State Machine).

For each process type clear patterns exist in the figure. However, due to the communication between the process types the overall data flow yields a complex graphical representation.

User inputs are checked for completenes, correctness and consistency. Only if the inputs are accepted as complete, consistent and correct, the system is constructed. This is similar to Ada which identifies a lot more bugs during compilation than C, and hence reduces the testing effort for bug identification. However, the checking mechanism of ISG also covers operational and architectural aspects.

The output corresponds exactly to the user's specification. A number of instrumentation options and options for automated testing are supported by ISG. Here, "automated testing" means the automated derivation of test cases and automated stimulation of the system including stress testing and fault injection. For system validation an evaluation report on the observed properties of the system (including performance and resoure information) is immediately provided after system execution. Hence, the user provides the inputs and defines the configuration, presses a button and waits until the textual and graphical evaluation reports are available, after about 20 minutes latest + the (application dependent) time needed for testing.

Complementary to the infrastructure as generated by ISG, software for data acquisition, monitoring and processing, data calibration, on-board database operations and telemetry handling was automatically derived from spreadsheet inputs, which already existed in the MSL project to document the hardware interfaces and architecture of sensors and actuators. The MSL on-board database is distributed over the two SPLC nodes. Continuous updates of the instances on the two nodes are required. The two-node version could be tested on one physical node. After such tests the automatically generated two-processor version ran immediately

The EM integration of MSL is nearly completed now. From this perspective the MSL users consider as very useful:

- the high stability and correctness of the ISG / ASaP software library and the automatically generated software,
- the full coverage of the application by the automated generation process
  supporting the complete application chain from command handling via distributed real-time processing to telemetry handling
- the flexibility to easily change the system's topology and platforms without any change of source code statements and any need for further manual intervention
- the inherent correctness of the software so that additional tests after a configuration change are not required.

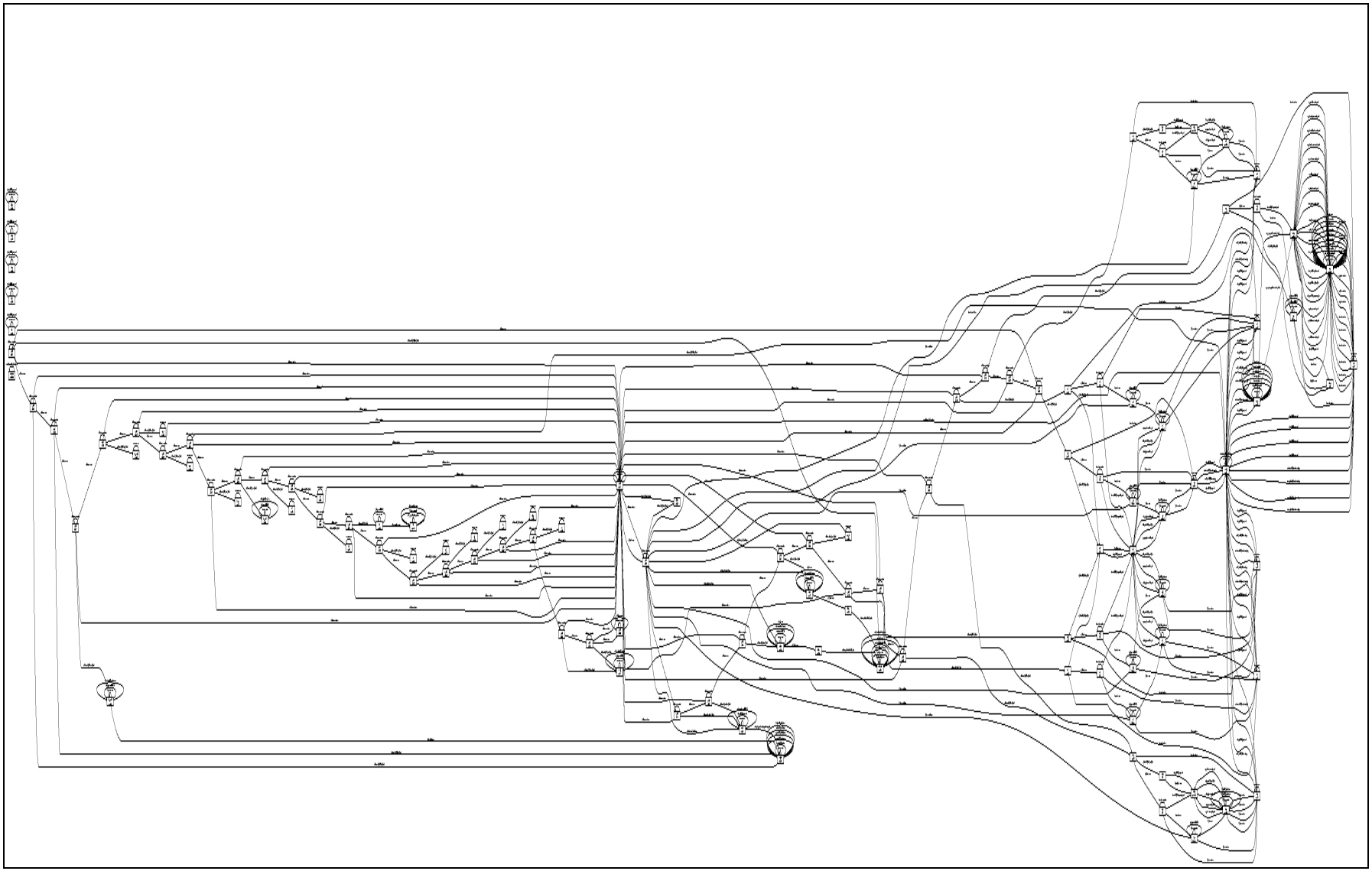


Fig. 5-1; Complete Data Flow of A Distributed Real-Time System

5.1.2 The CRISYS Project

For the ESPRIT project "CRISYS" (Critical Control Systems) [20] a 16-processor system has been automatically generated and executed on processor subsets between 1 to 6 nodes. Subject was a synchronous distributed application. The software generated by ASaP / ISG was used to exploit the impact of time jitter and data faults. The system included redundant units (up to four equivalent units) and a number of components for supervision and monitoring of unit and system status. This system is of type “AOCS / GNC” (Attitude and Orbit Contrl System, Guidance and Navigation).

Code generated by the Scade tool was plugged into the interfaces provided by ISG. A number of scenarios were executed with different types of time jitter and injected faults. The impacts on the different stages of data processing were analysed and the weakness of the plugged-in software was immediately identified.

5.2 Dedicated Application Software

For a number of (in-house) applications sequential code has been automatically generated. For each such application type the needed tool had to be established. However, a significant part of existing code could be reused either directly or by slight modification, to built the generators of ScaPable software.

The automatically generated software consists of the operational software as specified by the user’s inputs and the compilation and linking environment, and instrumentation if appropriate.

5.2.1 Operations on User-Defined Data Types

Operations on user-defined data types are rather complex if a number of nested data structures exist. Usually, for each such structure manual coding / manual intervention is needed. Although this coding is straightforward, it is time consuming and errorprone. Now, for automated generation by ASaP a user only has to provide a template and code for the basic types of the language line "int", "char", "double" etc.. In the simplest case a user only has to specify by the template the number of parameters of the functions to be generated.

This way easily software for initialisation, processing and printing of the contents of such types can be established for any types a user defines.

An example is the conversion between "Little Endian" and "Big Endian". By little effort a complete test and verification scenario can be generated covering random initialisation, forward and backward conversion, comparison and documentation of the results.

Similarly, software for conversion of enumeration literals between the enumeration type, integer and string in both directions for about 100 enumeration types was generated. This software was needed to support random initialisation and change of the data representation (between user-friendly text format in data files or output and C numerical format).

5.2.2 Adaption of Language Interfaces

For a C-like scripting language an interface had to be established for integration of C-functions into the scripting environment. This included description/definition of types and parameters, stack description and adaption of data alignments. The required transformation procedure is so complex that it is nearly impossible to manually provide the needed code even for a few functions.

This transformation capability is applied now to more than 500 functions and this software is generated within about 20 seconds on a PC-800 MHz being always correct. The interfaces of a number of functions were changed several times and stack alignment, too. Without the capability for automated code generation our estimate is that about nine man month would have been needed in addition.

5.2.3 Environment for On-Line Help Facility and Training

For matters of training an environment is needed to allow a user to experiment with above functions. This requires an embedding environment to call a function and to provide valid and meaningful parameters. With little effort above tool could be expanded so that the needed environment is now included in the automated generation process.

5.2.4 Automated Documentation

Extraction of interface documentation from source code into e.g. a MS-word document is very time-consuming. Tools for automated documentation already exist, but - as far as we know - not for automated generation of a MS-word

document and mixing of automatically with manually generated text. In addition, links shall be provided extensively between the function's description and appendices about examples, taking the information about the examples directly from their source code.

Again, above tool was expanded and it does now generate about 1200 pages of the user's manual within about 1 minute. MS-Word needs about 1 minute to read the file. Conversion from MS-word to pdf by the pdf-distiller of Acrobat [21] takes about 2 hours.

5. 3 Benchmarking

Section 5.3.1 gives the amount of generated code and the related generation time on a PC-800 MHz for a number of quite different applications. Productivity figures and the man-power equivalent of the automatically generated systems and code are derived in section 5.3.2. In section 5.3.3 we are discussing aspects of scalability and portability of these figures. Finally, a figure on observed bugs/LOC is derived in section 5.3.4.

Table 5-1 provides information on the code size in terms of bytes and lines and the duration of system generation. By Table 5-2 we compare and calibrate roughly the unit "lines" of Table 5-1 to "LOC" (Lines Of Code") taking a reasonable assumption on the ratio between a "LOC" and the number of bytes of the target object code. This allows us to associate the amount of lines and the generation times to an equivalent man power which would be required for manual development (Table 5-4).

5.3.1 Timing and Sizing Budgets of Automated Code Generation

The provided generation times as given by the last column of Table 5-1 were measured on Solaris, Linux and VxWorks platforms and CPUs of different power. To make them comparable they have been normalised to a PC-800 MHz / Linux platform.

The sizing figures refer to the total size of the application consisting of c- and h-files, scripts and data. This is the amount which usually has to be covered by manual coding, testing and verification.

In case of # 1 - 4 a BSSE in-house tool "linecount" for counting of lines was applied which allows to distinguish between comment and blank lines and active language statements. To count only active statements the BSSE utility removes conditional statements like "#if" or "#ifdef" etc.. Similarly, script and data files are processed.

In case of # 5 - 7 the Un*x word count utility "wc" was applied to get the number of lines and bytes. To derive the "net" number of relevant lines we applied figures on the average ratio between comment, blank and relevant code and data lines as obtained by our "linecount"-tool. The ratio between relevant code and total size was in the range of 50 .. 70 %. Therefore we applied the pessimistic figure of 50 % to downsize the total amount to the relevant amount. Such more detailed figures are available, too.

The column "Size of Input" of Table 5-1 considers the form sheet and data files which are provided by the user for automated system generation. The column "Size of Output" gives the size of the executable system (including the size of the scripts and data representing the automatically generated complation and run-time environment).

The column "Ratio Output / Input" compares the output with the provided input. These figures indicate a trend on what can be saved in terms of man-power and time. The observed minimum is about 5, and for ISG applications (#1-3) the ratio is far beyond 10. Taking 10 as a typical figure this means: if about 50% of an application are covered by ScaPable software like ISG, about 40% of costs and time-to-market can be saved, apart from the increase of quality.

For comparison, figures on the ISG environment are also provided (#4). The size of ISG code does not depend on the selected instrumentation, because instrumentation options only affect the application code. However, they impact what is being linked from ISG code to the application.

The column "Comment" provides additional information on the type of the system.

The ISG instrumentation options for system generation are:

- none
  The code corresponds to what is finally running on the target system, no additional instrumentation.
- medium
  Instrumentation is added for analysis of coverage, peformance and resource consumption.
- full
  "medium" plus automated test stimulation plus recording of data flow by MSC's and timing diagrams (graphics).

#	Application	Comment	Size of Input files / l ines / bytes	Size of Produced Code files / lines / bytes	Ratio Output/Input lines / bytes	Duration
1a	distributed real-time system 40 process types, 50 process instances, 2 nodes the input covers the spreadsheet and user code *Type: DMS (Data Management System)*	full instrumentation generation from scratch one processor, Un*x/Linux, VxWorks	25 / 1600 / 150,000	1454 / 542,000 / 23.5 MB	338 / 156	20 minutes
1b		medium instrumentation, one processor Un*x/Linux, VxWorks	same figures	1443 / 513,000 / 22 MB	320 / 145	20 minutes
1c		no instrumentation, one processor Un*x, Linux VxWorks	same figures	1443 / 484,000 / 19.5 MB	302 / 133	20 minutes
2	data processing and database software The input includes the full system definition (HW and SW) by Excel and Access sheets., of which only a small part is needed for SW generation. *Type: AOCS / GNC (Attitude and Orbit Control)*	distributed database generation from scratch from data acqu. to telemetry handling full instrumentation	3000 / 1.4 MB	16,000 / 24 MB	5 / 1.8	10 minutes
3	distributed synchronous systrem 9 process types, 16 process instances, 16 nodes	generation from scratch	838 / 24100	201,820 / 8.6 MB	240 / 350	10 minutes
4a	ISG	full instrumentation	---	506 / 101,400 / 3.7 MB	---	---
4b		medium instrumentation	---	506 / 101,300 / 3.7 MB	---	---
4c		no instrumentation	---	506 / 101,300 / 3.7 MB	---	---
5	Operations on User-defined Types	e.g. Little / Big Endian	341 / 6.8 KB	2300 / 42 KB	6 / 6	1 sec
6	Interface adaption + On-Line Help Facility	510 functions	1300 / 77 KB	145,00 / 3.4 MB	110 / 45	30 sec
7	User's manual	510 functions, 1200 pages the input includes the word-templates	3400 / 425 KB	27,000 / 2.5 MB	8 / 6	1 minute

Table 5-1: Productivity Figures and Duration of System Generation

5.3.2 Derivation of Productivity Figures and Man Power Cost Equivalents

Now, we want to derive the manual effort which corresponds to the size of the automatically generated system. For this task we need to execute a number of intermediate calculations and to make a number of assumptions because no figures on manual development of the same system are available for obvious reasons. The derived figures should only be considered as a rough estimation indicating a trend, but not as accurate figures. For the calculation we will make pessimistic assumptions, i.e. taking either the minimum or maximum value of a range, whatever is pessimistic.

To derive the equivalent man power we will convert the size of the target code into "LOC" and derive a calibration factor between the "counted lines" and a "LOC". Here we have to make an assumption on the rate "target bytes/LOC" which we take from a previous benchmark on Ada software plus an additonal pessimistic factor of 5. To convert the LOC into man power we take published data plus an additional pessimistic factor of 10.

This is the calculation scheme (so a reader may take his own data and derive own results according to this scheme):

(1) derivation of "LOC" from the counted numbers of bytes and lines of Table 5-1,

  (1a) calculation of "operational net lines" from the overall amount of generated lines (Table 5-2)

  (1b) comparison of size of executable, binary code with counted lines (Table 5-3)

  (1c) consideration of ratio between "counted bytes" and "counted lines" ("average contents of a line") and derivation of an "effective number of LOC" for each application (Table 5-3)

(2) derivation of the "man power equivalent" and "cost equivalents" for the automatically generated systems and code, taking a reasonable productivity figure for LOC per hour (Table 5-4).

In Table 5-1 the unit "line" is understood as a non-comment, non-blank line. If both, code and comments, do appear in a line, this line is counted, but the commenting bytes are not counted. As a line may only include a "{" or "}", a statement may spawn over a number of lines, and the amount of counted lines does not comply with the definition of "LOC" which usually is understood as the number of colons (";").

Moreover, we have to consider that a "LOC" only applies to the real code of the application loaded on the target, while the sizes and lines given by Tabe 5-1 represent the complete environment including scripts and support utilities. Therefore we subtract from the lines of Table 5-1 such "non-operational lines". The number of script lines and of "environmental support software" which is not loaded onto the target are subtracted in any case. The operational support software is fully, partly or not loaded on the target depending on the instrumentation options. Table 5-2 shows the results for application #1. The net values of Table 5-2 are taken as input for Table 5-3.

Instrumentation Option	Total # of Lines and Bytes	Script Lines and Bytes	Target Support Software Lines and Bytes	Environmental Support Software Lines and Bytes	Net # of Lines and Bytes (rounded)
full	542,000 23.5 MB	49,000 1.7 MB	fully included	100,000 4.5 MB	400,000 17.3 MB
medium	513,000 22 MB	49,000 1.7 MB	2000 90,000 partially included	100,000 4.5 MB	380,000 15.8 MB
none	484,000 19.5 MB	49,000 1.7 MB	5400 260,000	100,000 4.5 MB	330,000 13.3 MB

Table 5-2: Derivation of Net Lines

Then we compare the amount of counted lines with the size of the executable having loaded it on the target acording to the linker's map listing (without operating system and other software), and correlate the unit "lines" with unit "LOC" by making a reasonable assumption on the ratio between "LOC" and the number of generated bytes of the executable code.

This size is derived for VxWorks and 386 and SPLC targets. Table 5-3 gives an overview on the relevant figures for the "one-target" configurations and three instrumentation modes. "one-target" mode means the two processors were emulated by one physical processor, while "two-target" mode means the two-processor system would really run on its intended platform. For derivation of the calibration factor "Lines/LOC" only the "one-target" version has been considered, because a fixed amount for management of the application is included on each node and processes types which have instances on both nodes would be counted twice for the two-processor configuration. From Table 5-3 we derive a ratio "Lines/LOC" between 4 and 10. Hence, we will take the pessimistic value 10.

However, we need also to consider the ratio of "source bytes / source lines", otherwise we would be wrong in case of application #2 which includes much more bytes per line than the other applications. In order to get the effective number of LOC we made the assumption that one LOC yields about 50 bytes on the avarage, which again is very pessimistic as for Ada programs an average figure of about 10 bytes/LOC was derived for a CISC (PC) platform [22]. By Table 5-4 we derive the effective number of LOCs for each application.

Instrumentation	**Platform**	**CPU** Size of executable / MB from map listing gcc 386 gcc SPLC	**Net Lines** **Net Bytes** from Table 5-2	**Bytes / Line** **and** **Source Bytes / Object Bytes**	**Bytes / LOC** previous benchmark [22}	**Calibration Factor** Lines / LOC
full	one-target	4.7 -	400,000 17.3 MB	12 / 3.7 -	40	3
medium	one-target	3.9 3.2	380,000 15.8 MB	10 / 4 10 / 5	40	4
none	one-target	1.6 1.7	335,000 13.3 MB	5 / 8.3 5 / 7.8	40	8

Table 5-3: Final Correlation of Counted Number of Lines with "LOC"

When knowing the size of the target code of an application (as for #1), we can also derive the LOC directly from the size of the executable. In case of #1 this yields about 40,000 LOC taking a value of 40 bytes.

#	Application	Bytes	Lines	Bytes / Line	Lines / LOC	Effective LOC
1	Distributed Real-Time System	20 MB	335,000	40	8	40,000
2	Data Processing and Database Software	2.4 MB	16,000	150	3	5,000
3	Distributed Synchronous System	8.6 MB	200,000	43	8	25,000
5	Operations on User-defined Types	42,000	2,300	18	10	200
6	Interface adaption + On-Line Help Facility	3.4 MB	145,000	23	10	15,000

Table 5-4: Counted Size vs. Effective LOC

Finally, by taking the LOCs from Table 5-4 and assuming a productivity rate of 10 LOC / man hour we can calculate the man power equivalent (MPE) for each application (Table 5-5). Please be reminded that the value of 10 again is a pessimistic assumption, because the usual rates for space applications are in the range of 1 .. 2 LOC / man hour [23], and even for other application types this is a rather high value. We are taking a value of 1600 man hours per man year, and

40 man hours per man week. To derive the equivalent costs we take a value of 80,000 Euro per man year, which considers the net in-house costs, only.

The MPEs depend on the application type and the generation mode. Applications #1 and #3 represent code of systems and the coresponding MPEs are identical. In case of #4 possibly the conversion factor "Lines/LOC" should be still lower due to the rather high density of source bytes per source line. For generation of functional software (#5 and #6) the MPEs are much higher. This can be well understood because there is no overhead for the compilation and execution environment and a system's topology.

Now, we are looking on the generation modes. For #1 and #3 generation is done by a mix of C programs and scripts. In case of #2 pre-processing of data is done by scripts and this step consumes about 80% of the total generation time. #5 and #6 are generated by C programs and the time consumption by scripts is marginal. If the scripts would be replaced by C programs the genration time of #4 would probably decrease from 10 minutes down to about 2 .. 4 minutes. This would bring the results of #4 into the range of the other applications. Just to give an idea on time consumption during system generation: for #1 and #3 the compilation and linking process takes about 30% of the generation time. Obviously, there is still a huge potential for tuning when replacing the scripts by C programs

#	Application	Generation Time / *Dominating Generation Type*	LOC	Equivalent Man Power	Equiv. Man Years / 1 h Generation Time	Equiv. Costs / 1 h Generation Time
1	Distributed Real-Time System	20 minutes *C programs and scripts*	40,000	2.5 man years	**7.5**	**600 kEuro**
2	Data Processing and Database Software	10 minutes *scripts*	5,000	500 man hours 0.33 man years	**2**	**160 kEuro**
3	Distributed Synchronous System	10 minutes *C programs and scripts*	25,000	1.5 man years	**9**	**720 kEuro**
5	Operations on User-defined Types	< 1 secomd *C programs*	200	20 man hours 0.5 man weeks	**> 36**	**> 3 Mio.Euro**
6	Interface adaption + On-Line Help Facility	30 seconds *C programs*	15,000	1 man year	**120**	**10 Mio. Euro**

Table 5-5: Generation Times vs. Equivalent Man Power and Costs

These rather high man power equivalents are an outcome of the full automation of the development process. An engineer only provides inputs and receives the results, no further human intervention is required. Even when an engineer needs about 2 .. 3 man months to provide the inputs by iterations and incremental refinement and to evaluate the results, the saving of costs and time is still huge. Moreover, an engineer can spend more time on the system and operational aspects and still needs much less time, because the implementation effort disappears.

The achieved degree of full automation may be compared to the process of compiling and linking. In this area it is state-of-the-art that no human intervention is required between the different compiler and linker paths. Imagine, how long compilation would take if an engineer would have to take the output of the previous phase and to convert it into the input of the next phase.

Another aspect of cost saving is automated generation of documentation. From the provided input form sheet of #1 about 500 graphics are automatically generated and included into a RTF-document giving summary and detailed views on data flow, inverse data flow, overall data flow, state transitions, system topology and calling hierarchies of functions (caller-called and called-caller). User descriptions are automatically be merged. Even if the system is restructured, the

human contribution is still placed at the right location. If such views would have to be drawn manually, about 1.5 man months would be required if an effort of 1/2 hour is assumed per graphic, plus the later maintenance effort.

5.3.3 Scalability and Portability

The equivalent man power costs as given by Table 5-5 refer to one hour generation time. The upper limit currently reached for system generation amounts to about 20 minutes. So there is still a large potential to enter the next higher step of system complexity.

The major part of system generation depends only linearly on the system size, like in case of the number of process types. Only a small part depends on second order of components which occurs e.g. for set-up of topology by point-to-point connections. Therefore the generation time will roughly increase by first order for a larger number of components, becasue it depends only weakly on second order of system size. For applications needing functional code generation the dependency is of first order on the user inputs.

An own generation process is started for each platform type in case of a heterogeneous system configuration. Usually, each such process runs on an own host, so that the generation is not increased due to sequential processing of a number of platform cconfigurations on the same host. The distribution of such generation processes is part of automation. All processes wait on each other until the last processes finishes linking, and then agree to start execution of the whole distributed system.

5.3.4 Quality Considerations

The first system built by ASaP / ISG was delivered in September 2000 (#1 and #2). Since that time the final version of the software was built and integrated (EM intergation which is the final software integration). Two bugs were reported by the user: (1) propagation of an overflow of a command (wrap-around) counter after injection of 2^{15} ground commands at a rate of about 1 command / second (about 8 hours of continuous system exedcution), identified by a built-in error check, and (2) wrong assignment of processes priorities in case of distribution.

The total amount of LOCs of #1 and #2 is 45,000 according to Table 5-5, this gives a bug rate of about $5*10^{-5}$/LOC, or roughly less than 10^{-4}/LOC. In [24] figures are provided on the occurence of bugs per LOC. It is stated there, that a figure of $<10^{-3}$/LOC is considered as “very good” and the range $3 .. 8 * 10^{-3}$/LOC is typical.

However, we have to comment our figure: 10^{-4}/LOC does not mean that for an automatically generated system of 100,000 LOC or 1,000,000 LOC 10 bugs or 100 bugs should be expected. Apart from the fact that the bugs have already been fixed, the bugs are related to the AsaP / ISG kernel. Therefore an extrapolation to a larger size is not meaningful at all. This raises the principal question, how a representative bug rate for ScaPable software can be derived, so that it can be compared with usual figures Hence, above figure of 10^{-4}/LOC just represents a snapshot for a certain application.

6. CONCLUSIONS

To master the challenges of development of complex software a higher degree of automation is needed, in order to minimise the impacts by the most critical resource "man". As "man" does have problems with complex and non-visible interfaces the only way to escape from this problem is

- to delegate the most critical tasks to automata and to limit the contribution by "man" to the tasks for which inspiration is really required,
- to set-up an appropriate organisation for system construction which allows automata to build correct software from inputs which "man" can reliably provide.

It does not make any sense to solve a problem which is caused by having "man-in-the-loop" by adding more human resources. Such a problem can only be solved by reducing the impact by the imperfectness of "man" and by relying on a formal generation process based on automata which continuously can be improved.

Such automated generation of software must address the whole development chain of an application, and not only parts of it. If not, it is not possible to get rid of the problem. If the full application cannot completely be covered, it is essential to divide an application into smaller self-consistent parts, which can be subject of full automation.

To succeed this way the approach on "ScaPable software" divides the problem into two principal steps:

- step1:
  provision of a coherent framework to automatically construct a distributed (real-time) application which covers all aspects of such an application regarding parallel processing, (real-time) scheduling and communication, allocation of processes within a network, the aspects of fault tolerance and portability etc.,
- step2:
  provision of dedicated, more specific frameworks for a number of application domains which address the part complementary to step 1, the sequential, functional software, usually related to data processing and decision making.

Moreover, it is essential that the generation scheme limits the dependency of the manual effort on the application size. For the approach chosen by BSSE the manual implementation effort is zero once the generation environment has been established.

By benchmarking the impact of automation on productivity was evaluated and the feedback was used to continuously tune and improve the actual approach and to enter the next step of automation since a number of years.

This way the productivity was significantly increased and the development time was shortened. However, what is even more important is: for the first version or an update only very little information is required. When the input is accepted it is guaranteed that a correct result will be produced for the complete application. "Man" just has to press a button and then can relax and wait for the first or the next software version correctly produced and immediately provided. A first rough estimation on the man power equivalent of automatically generated systems and software yields a range of 2 .. 100 man years per hour of automated generation time.

The provided figures on productivity indicate a large potential for saving of costs and development time, even if only some part of an application is covered by ScaPable software. Although the figures were derived from pessimistic assumptions, the results indicate that a huge amount of time and costs can be saved by automation:

Recommendations as ISO 15504 may lead to the conclusion that they are the only means to improve the situation. This is wrong. As has been proven by domains like manufacturing of TVs and cars, and as the provided figrues indicate, a new technology based on automation allows to escape from the current big problem software development has.

The important point to succeed is to identify potential subjects of ScaPability, and to organise the implementation such that a complete framework can be built automatically. If this is not completely possible right from the beginning, it is still of high benefit to start, to get experienced and to make progress in increasing the degree of automation by ScaPable software. A positive attitude regarding automation is very important: to be open for automation, to identify opportunities for automation and to change the organisation like other domains did, instead of finding arguments against automation.

Regarding the discussions on the "Software Crisis" which shall happen during this conference, we believe that this problem will become less critical by automation. It is our impresssion that this problem is mainly caused by the following facts:

(1) the believe that higher quality implies more man power,

(2) missing benchmarks on the benefits of software methods and tools, and

(3) missing tuning of software development approaches as a consequence of (2), especially towards higher productivity.

The current standards are an outcome of manual-oriented development and presumably prevent a higher degree of automation. E.g. a request for implementation-related documentation and reviews is not compliant with an automated process chain at all. Today, nobody does request documentation and ask for reviews on compiler-internal results. The automated process chain ensures the quality of the results. Therefore no analysis of such intermediate results is necessary. Of course, the process chain itself needs to be subject of an analysis, but only once or in case of later improvements.

Our current experience is: the more we apply the ASaP approach ("Automated Software Production), the more we learn on how to get the appropriate organisation to build ScaPable software in domains not yet covered.

Based on the current experience the ASaP approach shall be applied now to quite different application domains like medical robots, robot systems which require a high degree of availability and correctness, and automated re-engineering of legacy systems. Also, we plan to support DSP operrating systems, so that ASaP can fully cover a mix of µPs and DSPs and a mix of operating systems. Due to the short turn-around time between provision of inputs and availability of

the real system, the potential customers expect to have significant competitive advantages because new features can be provided much faster or only due to the high dynamics of iterations possible by fully automated development.

7. QUESTIONS RAISED AFTER THE PRESENTATION

Q1: What is the impact by a bug compared to the manual approach?

A1: Due to use of templates more executable code may be impacted than for the manual approach, e.g. in case of 50 instances all 50 instances may be corrupted. However, as much more components are affected the probability to detect such a fault is higher, at least by the instance number (50). Therefore the chance is higher that a bug will be detected before a system is put into operation. As the automated approach implies full reuse of the AsaP / ISG code the bug probability decreases rapidly the more systems are generated, while for the manual approach the bug rate is always approximately the same for each system.

Q2: How is the experience from practice considered by this approach?

A2: Experience is directly fed back into and saved by the production chain. What is saved into a “knowledge base” in case of the manual approach, and possibly applied by the engineers to the next project, is immediately made available for the next version of the application and any other following application, for sure. Hence, the automated production chain is the knowledge base. And the expereience saved remains available even if engineers do leave the team.

Q3: How can other code be attached?

A3: AsaP / ISG provide interfaces to plug-in code, so that this code is preserved when a new version of the automatically generated part of the system is produced. If the other ocde applies own interface standards the merge of both code segments can be automated, too.

Q4: Which percentage of a system’s code was covered by ASaP / ISG?

A4: In case of #1 the complementary task was not performed by BSSE, therefore no figures are available. For #3 we checked our files and counted about 15,000 lines by applying wc and taking all lines of the files. This amount has to be compared with the about 200,000 automatically generated lines and yields a figure of about 10%. As manual code was only provided for about 50% of the processes (for the other processes the automatically generated dummy code was used), we get roughly 20% of manually generated code. Hence, we can conclude that about 70 .. 80% were covered by ASaP / ISG in case of #3.

Q5: Which effort is needed to prepare the inputs?

A5: Usually, a system is incrementally refined. To start by a first version may require about half a day. Then a number of iterations should be performed until the system gets its final shape. However, it is also possible to start by a refined inputs derived from previous work with other tools. A rough estimation can be derived for application #1. The amount of inputs is about 1600 lines. Taking this as the amount of LOC and applying the figure of 10 LOC / man hour (as for the output), the effort required to provide such inputs would amount to 160 man hours or 1 man month. From an overall perspective a system engineering effort of about 3 man months seems to be reasonable, which implies a figure of about 3 LOC / man hour. However, this work has also to be performed in case of pure manual development.

REFERENCES

[1] "Virtuoso" is a product of WindRiver Systems Inc., supporting distributed processing on DSPs
WindRiver Systems, Inc. 1010 Atlantic Avenue, Alameda, CA 94501-1153, USA

[2] Rhapsody, i-LOGIX Inc., 3 Riverside Drive, Anover research Park, Andover, MA 01810, USA

[3] ObjectGeode and SDT are products of TeleLogic
TeleLogic AB, PO Box 4128, S-20312 Malmö,Sweden

[4] DASIA96, Eurospace Conference on "Data Systems in Aerospace", May 20-24, 1996, Rome
R.Gerlich:"From CASE to CIVE: A Future Challenge"

[5] HRDMS (Highly Reliable DMS and Simulation), ESTEC contract no. 9882/92/NL/JG(SC),
Final Report, Oct. 1994, Noordwijk, The Netherlands

[6] OMBSIM (On-Board Mangement System Behavioural Simulation), ESTEC contract no. 10430/93/NL/FM(SC),
Final Report Nov. 1995, Noordwijk, The Netherlands

[7] DDV (DMS Design Validation), ESTEC contract no. 9558/91/NL/JG(SC),
Final Report Dec. 1996, Noordwijk, The Netherlands

[8] SDL'97 Forum", September 22-26, 1997, Evry, France, ISBN 0 444 82816 8
R.Gerlich:"Tuning Development of Distributed Real-Time Systems with SDL and MSC:
Current Experience and Future Issues" (figures were provided by the presentation)

[9] Eurospace Symposium DASIA'01 "Data Systems in Aerospace", May 18-June 1, 2001, Nice, France
R.Gerlich: "Platform-Dependent (Cost) Impacts on Portability, Software Reuse and Maintenance"

[10] ISO/IEC TR 15504: Information Technology - Software Process Assessment, Technical Report Type 2, 1999

[11] A. Dorling, Ch. Völcker, A. Cass, L. Winzer:
SPiCe for Space:A Method of Process Assessment for Space Software Projects
ESA Report SP-483, DASIA 2001, May 28 - June 1, 2001, Nice, France

[12] Vortragsreihe der Bezirksgruppe VDE Albstadt-Sigmaringen, SS 2001
April 26, 2001, Fahchochschule Albstadt-Sigmaringen, Germany
R.Gerlich: Automation in der Softwareentwicklung - ausfuehrbare Programme ohne Programmierung

[13a] ISG, 1999-2002, Rainer Gerlich BSSE System and Software Engineering, Auf dem Ruhbuehl 181, D-88090 Immenstaad, Germany

[13b] MOVEP'2k: Modelling and Verification of Parallel Processes
June 19 - 23, 2000, Nantes, France
R.Gerlich: An Implementation and Verification Technique for Distributed Systems

[14a] Eurospace Symposium DASIA2000 "Data Systems in Aerospace", May 22-26, 2000, Montreal, Canada
M.Birk, U.Brammer, K.Lattner, M.Ziegler:
Software Development for the Material Science Laboratory on ISS by Automated Generation of Real-time Software from Datasheet-based Inputs

[14b] Behavioural Validation of MSL, ESTEC Contract No.: 13309/98/NL/MV,
Final Report SMSL-RP-001-BSSE, Nov. 1999, Noordwijk, The Netherlands

[15] SCADE, Telelogic Technical Center Toulouse, 150 rue Vauquelin, F-31081 Toulouse Cedex, France

[16a] EaSySim II, 1996-1999, Rainer Gerlich BSSE System and Software Engineering

[16b] ESA News Vol. 7, No.4, December 1997, "Preparing for the Future", pp. 18-19
R. Gerlich: EaSySim II: Software System Validation Using Executable Models
ISBN 1018-8657, http://esapub.esrin.esa.it/esapub.html

[17] VxWorks is a product of WindRiver Systems, Inc. (see [1])

[18] Solaris is a trademark of SunSoft Inc. 2550 Garcia Avenue, Mountain View, CA 94043, USA

[19] For information on Linux and download of Linux software see e.g
http://www.linux.org or http://sunsite.unc.edu/pub/Linux/kernel/linux

[20] CRISYS (Critical Instrumentation and Control System) ESPRIT project EP 25514

[21] Acrobat is a product of Adobe Systems Inc., http://www.adobe.com

[22] R.Gerlich, U.Wagner: Tuning Ada Programs in Advance, Eurospace Symposium "Ada in Europe", September 26-30, 1994, Copenhagen, Denmark

[23] K.Maxwell, L.Van Wassenhove, S.Dutta: Benchmarking: The Data Contribution Dilemma, Proceedings of the European Control and Metrics Conference, 1997, Berlin, Germany

[24] N.E.Fentonm, N.Ohlsson: Quantitative Analysis of Faults and Failures in a Complex Software System, IEEE Transactions on Software Engineering, Vol. 26, No. 8, August 2000