From CASE to CIVE: A Future Challenge !

'DASIA 96'

- Data Systems in Aerospace -

Rome, Italy

May 20 - 23, 1996

Rainer Gerlich
~~Bodan~~ Software and System Engineering
Auf dem Ruhbuehl 181
D-88090 Immenstaad
Phone: +49/7545/911.258, +49/7545/529
Mobile: +49/171/80.20.659
Fax +49/7545/911.240
e-mail: gerlich@t-online.de

~~Bodan~~ System and Software Engineering

From CASE to CIVE: A Future Challenge !

Rainer Gerlich
~~Bodan~~ System and Software Engineering
BSSE
Auf dem Ruhbuehl 181
D-88090 Immenstaad, Germany
Phone: +49/7545/911.258, Mobile: +49/171/80.20.659
Fax: +49/7545/911.240
e-mail: gerlich@t-online.de

Abstract: Computer Aided Software Engineering (CASE) has been used for a long time without being able to enhance software development and to reduce costs significantly. Based on experience and results for efficient organisation of life cycle and system decomposition the benefit of "Computer Integrated Validation" (CIV) is shown. Theoretical analysis of the cost saving potential and results of recent study cases are given. Finally, potential conflicts with current practice are identified which may prevent use of CIV.

Keywords: Life cycle approach, integrated tool environment, system validation, system decomposition scheme, simulation, cost reduction, cost figures, reuse

1. INTRODUCTION

CASE (Computer Aided Software Engineering) was the keyword of the last decade. It was used as a synonym for efficient software development. However, these expectations have not been fulfilled. It is still difficult and cost-expensive to develop software. No significant progress was made by use of CASE tools, sometimes they even seem to be counterproductive. Other application areas, e.g. production of hardware, made much more progress in the same time. Looking for the reasons one major disadvantage of CASE is the organisation of development itself.

1.1 Current Situation

It is state-of-the-art that most tools are supporting dedicated life cycle phases, only. They put little emphasis on a coherent transition between the phases. One can get the impression that a coherent transition backwards (reverse engineering) is better supported than a forward transition.

Most tools support functional aspects, only. Performance, proof of correctness and completeness are still a matter of final integration coming at the end of the life cycle because the means which are used at the beginning are not sufficient.

CASE tools do only "local" optimisation. Tools try to support a single phase but they do not support "global" optimisation of all phases. If they claim to support several life cycle phases this support is still incomplete. E.g. it may just mean that a data base is shared by different phases. But it does not mean that the tools allow a coherent and automated transition.

So a lot of manual effort must still be spent for the transitions from one phase to the next one. Even if the effort for the activities of a life cycle phase would decrease to zero due to support of locally optimising tools, the manual effort for transient activities would remain as it is.

Hence, tools leave a lot of effort to the engineer although they could do this job. This is wasting of engineers' time. There are methods and tools which could improve the situation like SDL, HRT-HOOD and performance analysis tools. But it is not very common to use them.

Most of the methods and tools are more relying on documentation when concluding on the correctness of specification and design rather than on the feedback from the real system. But only the practical results of system operation decide on whether a system fulfils the user needs or not.

Between the "paper specification and design" and the correctly working system at the end there is usually a gap which is closed with a lot of headaches during the test, integration and acceptance phases. The needed feedback is obtained at the very end. It is well known that the later corrections are done the higher the costs are. To get higher efficiency validation of a system's properties is needed in an early life cycle phase.

1.2 Towards an Integrated Approach

The current understanding of the term "computer-aided" does not help to solve the problems of software development. What is needed is full integration of all life cycle activities by tools so that the human effort, which is "the" bottleneck, can be reduced to a minimum. Consequently, we either need another class of tools or we have to look if and how we can build such tools out of the existing ones to get more efficient support. The second choice is for the time being the most promising one. First results are now available. They were obtained during the ESTEC studies OMBSIM [1] and DDV [2].

In order to distinguish it from "CASE" this class of tools shall be called "computer integrated" ("CI"). "Computer Integrated Manufacturing" (CIM) is a well known keyword, but obviously not for software development until now.

What we need is:

1. better integration of life cycle phases by methods and tools,
2. validation already during early life cycle phases,
3. better organisation of system decomposition[1], system refinement, standardisation and reuse,
4. continuos improvement of organisation by analysis of observed bottlenecks and weakness.

The point 'organisation' seems not be considered in any of the software development approaches. Beside general recommendations no support is given to an engineer how he can efficiently organise the development of his application towards reuse and standardisation. During his education a software engineer does not learn to look rigorously for bottlenecks and to improve them.

The approach we have in mind we call "CIVE": Computer Integrated Validation Environment.

We will see that it is possible to achieve the needed improvements

a. by use of executable models during specification and design which can incrementally be refined to the final system, and
b. by a simple, but rigorous organisation concept.

In the second chapter we will discuss the principal needs for better organisation of development. Productivity figures are given in chapter 3 which confirm the expected potential for increase of efficiency. Chapter 4 discusses potential conflicts with current development guidelines and standards which should be solved in order to be ready for more efficient development in future.

[1] The term "decomposition" is used here because a system is incrementally refined into lower level components. However, one could also take the opposite point of view and use the term "synthesis". Then a system is built from pre-defined structures.

2. BETTER EFFICIENCY OF DEVELOPMENT BY BETTER ORGANISATION

2.1 What Needs To be Organised?

In order to succeed we need a better organisation of life cycle activities than now.

Firstly, we have to organise the style of expressing and implementing users' needs to be able to achieve higher efficiency. E.g. a lot of checks for consistency and correctness are still done manually because the used notations like textual requirements do not allow that tools can do this job.

Secondly, we have to ensure that the same notation can be used during all the life cycle phases. This means that e.g. the design notation should be compatible with notation of specification. Then design can automatically be derived from the specification without throwing away major parts during this transition. Similarly, the code could be derived from design directly.

In case of embedded systems we also need to consider hardware and software in a coherent manner. When we specify a system property we may not know whether we need hardware or software to provide it. However, when this becomes clear then it should be possible to make a transition to the right notation for either hardware or software.

Thirdly, we have to organise reuse. Reuse helps to decrease development effort significantly. But if one does not build-in reuse one will not succeed. Hardware is successfully reused because available components are taken into consideration for a new system. If a definition of the capabilities of such reusable components is missing like in software, then reuse is not possible. Also, if one does not introduce standards which support reuse either for the same project or for several ones one will not get a sufficient degree of reuse.

We will approach the organisation needs by two steps: by introduction of a formal notation and by rigorous rules to be applied with the formal notation.

The formal notation will allow that tools can take over more of the time-consuming manual activities. The simple, but rigorous rules lead to high stability during system development and refinement and to a high degree of reuse. They also allow to integrate other 'foreign' components in an efficient manner.

2.2 What Is the Benefit of a Formal Notation ?

2.2.1 Higher Degree of Automation

When applying a formal notation tools can check the properties expressed by it. So an engineer does not need to care about such checks any more, he just has to look for the results a tool produces for him.

In order to get maximum support such a notation should not only formalise interfaces like for subprograms, data types and data declarations, but also a system's (concurrent) behaviour, the data exchange and the algorithms by which the dedicated functionality is expressed.

To allow for coherent transitions between specification, design and coding a formal notation should be usable for each of these phases. To get an early feedback by the system-under-development obviously a formal notation should support automated generation of host and target code. It does not make sense to verify and validate a system representation, but to throw it away and to start for coding of the target software from scratch in the conventional, troublesome way again.

Modern languages like Pascal, Modula and more recently Ada and C++ are supporting well formalization of interfaces by introducing "strong typing" and object-oriented principles like encapsulation, abstraction and information hiding. However there are only a few formal notations like SDL88 [3], SDL92 [4], LOTOS [5], RAISE [6] and Statecharts [7] formalising behaviour.

Formal notations like Z [8], VDM [9] are supporting non-concurrent systems only. Object-oriented ideas are supported by SDL92 [4]. The advantage of SDL is that tools can perform automatically checks on behaviour and can generate target code.

None of such notations fulfils all the demands. So one needs to make a compromise. We decided for SDL because it meets most of the requirements, is standardised and sufficiently open to be complemented by other tools.

2.2.2 Higher Efficiency

During the ESTEC study OMBSIM [1] a first CIVE called EaSySim (Simulation environment for Early Validation of Design) [10, 11] has been established and reused for the follow-on study DDV [2]. This CIVE consists of two commercial tools Geode [12] and SES/workbench [13] and add-on software enhancing the capabilities of both tools towards the capabilities expected from a CIVE.

Experience could be gathered on how to get an efficient organisation and which effort can be saved. The cost saving potential due to SDL capabilities is explained below in detail.

In order to be able to identify the cost saving potential the life cycle phases are divided into principal activities which are:

1. a dedicated activity
   characteristic for a certain phase only e.g. requirements analysis in case of specification phase, architecture definition during design phase,
2. generation of documentation like specification, design, test and tracing documents,
3. verification, e.g. of interfaces and behaviour,
4. definition of tests,
5. coding, and
6. execution of tests.

Iterations and modifications are already covered by each of above activities.

In order to compare a CASE and a CIVE approach the effort of each such activity is estimated per phase. Firstly, a weight is assigned to each life cycle phase describing the percentage of its effort in the overall life cycle. Then for each phase above activities are weighted. The overall life cycle weight and the weights of activity per phase are normalised to 100%. The product of both weights gives the contribution of each activity to the life cycle effort per phase and for the total life cycle. Figs. 1a and 1b show the assigned weights for each of the phases and activities.

Conventional Lifecycle							
LC Phase	Weight	Relative Weights of Activities					
		Dedicated	Doc.	Verif.	Test Def.	Coding	Test Exec.
Spec	10	5	2	2	1	0	0
Design	20	10	4	4	2	0	0
Coding	15	0	3	3	3	6	0
MT	15	0	2.25	0.75	3.75	2.25	6
IT	20	0	4	3	4	1	8
AT	20	0	4	4	2	2	8
Total	100	15	19.25	16.75	15.75	11.25	22

Fig 1a: Effort Figures for CASE Approach

Incremental Lifecycle							
LC Phase	Weight	Relative Weights of Activities					
		Dedicated	Doc.	Verif.	Test Def.	Coding	Test Exec.
Spec	15	0	1.5	3	1.5	4.5	4.5
Design	40	0	4	2	8	12	14
Coding	0	0	0	0	0	0	0
MT	0	0	0	0	0	0	0
IT	10	0	2	1	2	0.5	4.5
AT	5	0	1	0.5	0.5	0.25	2.75
Total	70	0	8.5	6.5	12	17.25	25.75

Fig. 1b: Effort Figures for CIVE Approach

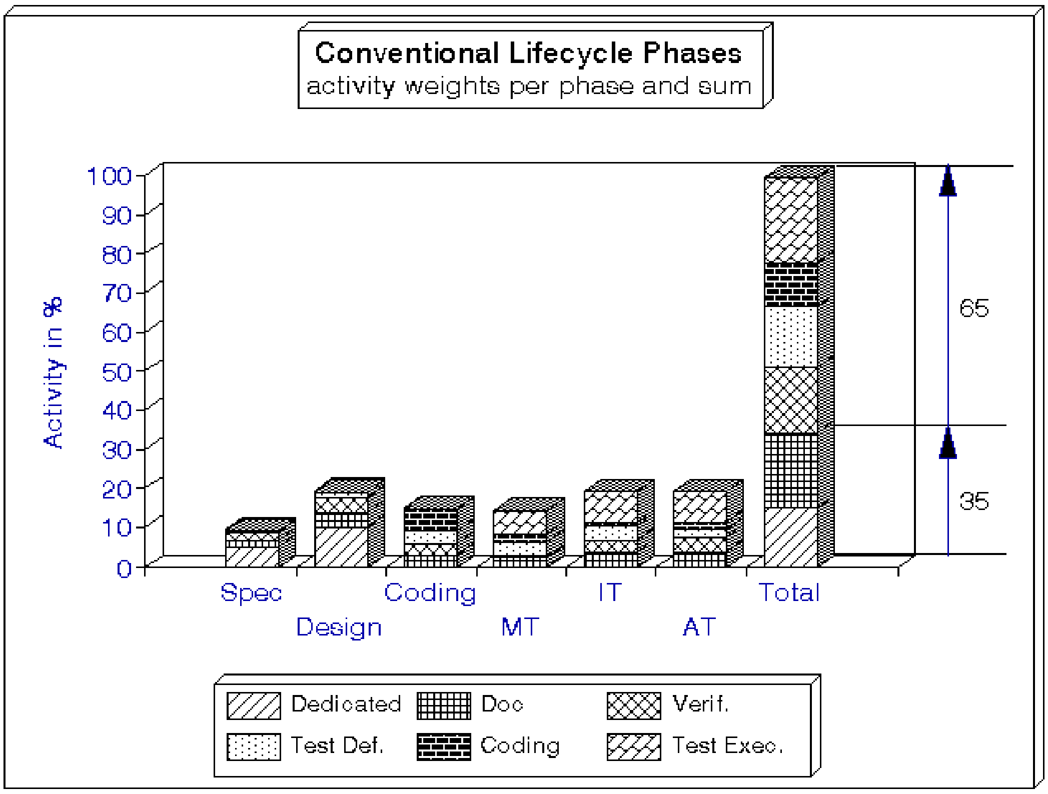


Fig. 2a: Effort per Activity for CASE Approach

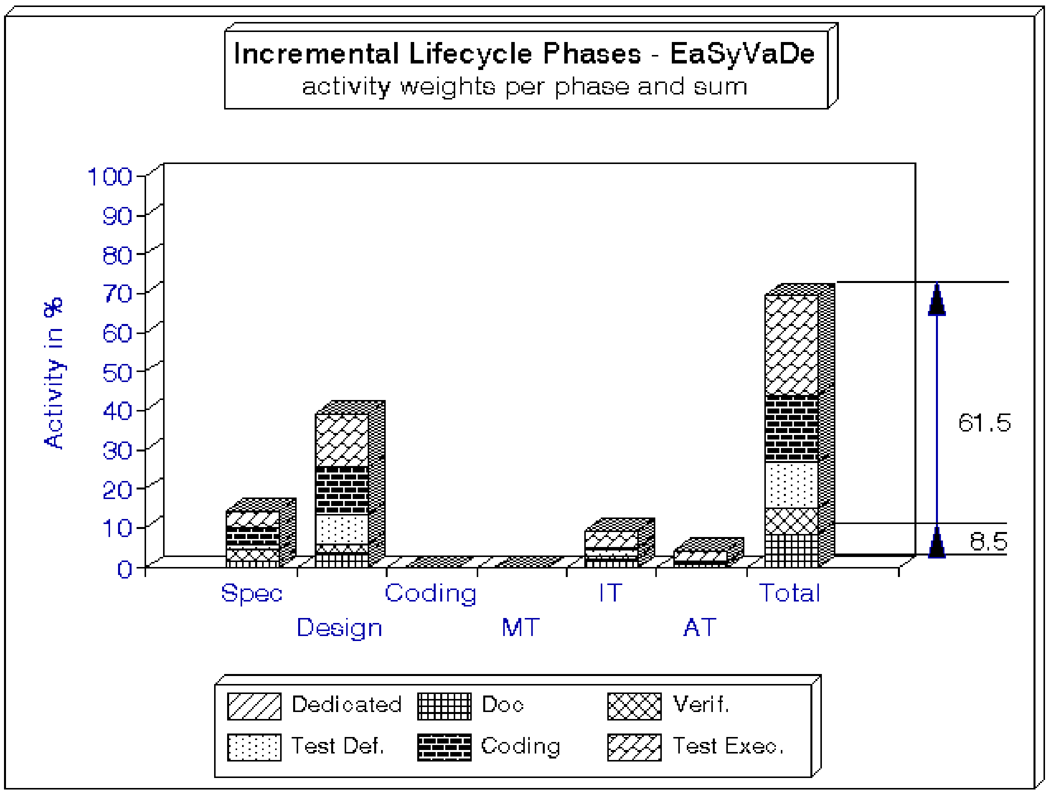


Fig. 2b: Effort per Activity for CIVE Approach

Fig. 2a shows graphically the percentage of effort for the CASE approach per phase. Fig. 2b gives the equivalent figures for a CIVE approach. The right-most columns indicate the contribution of each activity to the overall life cycle.

For CIVE the following modifications compared to CASE have been considered:

a. As a formal notation with capability for automated target code generation is used *coding* and *testing* is integrated into *specification* and *design*.

b. Due to the merge with specification and design the columns for the coding and test phases disappear in case of CIVE[2].

c. The dedicated activities for *specification* and *design* disappear for CIVE because such activities are producing paper only. They are completely replaced by coding in a formal notation.

d. As verification of interfaces and behaviour is well supported by SDL tools, the effort for verification is reduced significantly, also the related effort for documentation and reviews. This is reflected by the decreased weights of such activities.

e. For integration and acceptance testing the test effort is reduced due to better pre-testing and early validation during specification and design phases. Also, the effort for test definition and test execution is partially covered already by pre-validation and is therefore smaller.

In fact, in case of CIVE more effort can be spent on coding, test definition and test execution. The effort for documentation and verification is significantly reduced. The total effort is decreased from 100% for the CASE approach to (about) 70% for the CIVE approach due to getting rid of paper work and reviews which is covered to a higher degree by CIVE tools.

When looking on the summary figures for each activity in the life cycle (which are shown on the right side of Figs. 2a and 2b) one recognises that the ratio between the "paper" activities ("dedicated" and "documentation") and the "sophisticated" activities (verification, testing, coding) is changed from 35 : 65 to 8.5 : 61.5 which shows that by CIVE an engineer can concentrate more on the challenging activities.

Of course, this is just a rough estimation which shall identify the cost saving potential as such. Next projects which are executed over a full life cycle will allow to get more accurate figures from practice.

2.3 What Can Be Improved by Rules?

2.3.1 Incremental System Development

The process of system development starts at the highest level with user requirements. Usually, such requirements are addressing all system properties from top to bottom level because a user just expresses some requirements but not all. He thinks about details for some cases, for other cases he does not. He establishes the user requirements according to his point of view but not from what yields an optimum for system development.

So the first step is to convert the user requirements to a formal notation according to helpful rules.

Having converted the informal textual requirements to a formal notation - yielding executable models - and having defined test and operational scenarios we can use tool capabilities for

[2] Instead of removing the coding and test phases one could have removed the specification and design phases instead, of course. But the intention is to show that coding and testing is already covered by earlier phases when using executable models expressing specification and design. Specification and design phases are kept because they are the essential activities for implementation of a system.

verification and can execute the models. According to the feedback from model execution one can evaluate whether the system provides the expected services or not. If it does, the specification of a system is validated, if it does not one has to change the models and to start the next iteration of specification.

Having validated a specification one needs to think about resources and allocation of functionality to resources. Hence, we introduce a topology and an architecture. Functional components are further decomposed and mapped onto resources. They express requirements on the next lower level and are now the starting point for continuation with validation of a specification. Like we started with user requirements we are now starting with new requirements. We also have to consider the still open user requirements addressing this level.

In this manner we get an iteration between specification and design and a hierarchy of system components. From specification we go to design, during design we specify the requirements on next lower level, and continue with such specifications again. At each hierarchy level we introduce physical resources and we do a hardware-software trade-off.

A coherent iteration between specification and design is only possible if the same notation can be used for both phases.

2.3.2 Organising System Decomposition and Reuse

An approach which has been turned out to be stable is the so-called GIFTBox approach: "Generic Interfaces and Fault-Tolerant Boxes". The GIFTBox rules describe refinement of system components and communication interfaces. By the communication mechanisms one can identify flow of exceptions. This allows to master anomalies and to isolate them into a "box" which gets fault-tolerant properties.

Only the most important GIFTBox rules are given here:

1. *generic interfaces*
   they define the types of communication between generic fault-tolerant boxes and the semantics of the types (Fig. 3)
2. *master-slave concept*
   a co-ordinator ("master") is needed for a set of boxes (components, modules, "slaves") appearing on the same level to avoid conflicts in responsibility between the slaves (Fig. 4)
3. *top-down decomposition*[3]
   a. identification of dynamic components (modes) (Fig. 5)
   b. identification of static components (operations, functions, services, subsystems) (Fig. 6)

[3] In reverse order ("bottom-up" synthesis) the related rules can also be applied, of course.

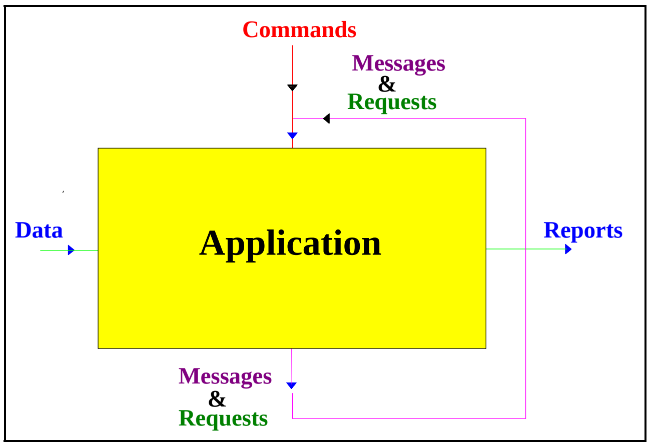
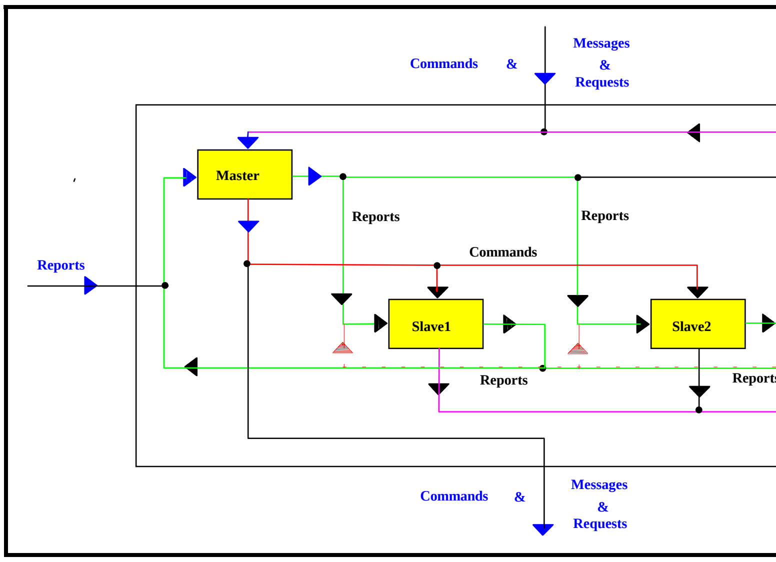
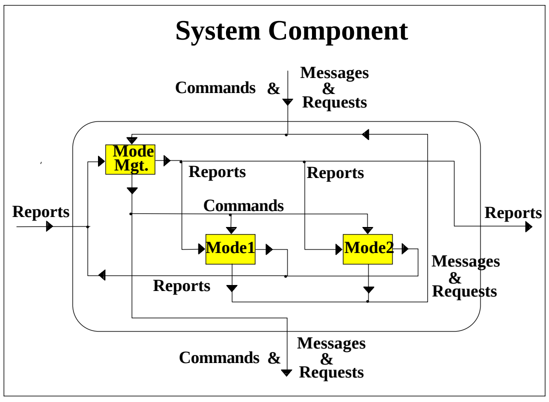


Fig. 3: Generic Interfaces





one mode within a certain period
mutually exclusive

Fig. 4 Control Hierarchy by Master-Slave Concept

Fig. 5: System Components and Its Dynamic Internals

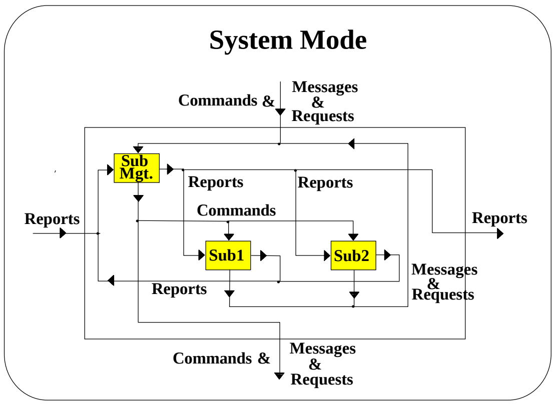
The interface types (Fig. 3) are divided into *control flow* (from top to bottom) and *data flow* (from left to right). Data flow is subdivided into *data* as input and *reports* as output. Control flow is subdivided into *commands*, *messages* and *requests*. The structure of a component's interface is fixed by the shown interface lines. Both streams may be merged into one channel, if needed. In SDL such data and control types may be defined as signals and an interface as signal list. The signals and signal lists are related to an SDL channel.

By using the fixed interface structure and the well-defined semantics we are able to build a component's internals in a clean manner, understanding what is coming in and what is leaving, and what is the impact on a component.

The *master-slave* concept (Fig. 4) is introduced in order to get a clean approach for responsibilities and commanding. Also, the master represents the interface to the outside world. If the contents of an interface is changed (the structure is fixed according to rule 1) then the slaves are not (necessarily) effected. Vice versa if the contents of the internal interfaces is changed the outside world is not effected because the master absorbs such changes.

Each slave communicates with his master only. However, another slave may listen to the reports sent by any slave. This allows to implement slaves which act as communication medium (channels). A master may provide to his slaves a communication mechanism to the outside world in a transparent manner. In SDL the merge of channels ("signal routes") and signals is supported by the notation itself. Therefore no additional functionality ("box") must explicitly be shown.

When we look at a system component we first see its different modes (Fig. 5), e.g. powered or de powered, standby or fully operational. For each mode a certain set of operations is provided by subcomponents (Fig. 6). In SDL such modes are expressed by states and a mode manager may be provided as a set of statements managing state transitions. In SDL subcomponents may also be represented by a sequence of statements or procedures or by processes.



several subcomponents exist over time
complementary functionality

Fig. 6: System Mode and Its Static Internals

In this manner we perform iterations between specification and design when we are going down in the system hierarchy. Inside a box we add new boxes: modes or subcomponents. For SDL this means we add new states and state transitions, procedures (as encapsulation of statements). If the functionality of an added box is too complex we shift it down one level to another subhierarchy.

From the leaves of a hierarchy code is generated automatically by a CIVE. Such "leaves" are "processes" in SDL. As we always end up in a hierarchy with leaves we can always generate code automatically and hence can execute a system and monitor its behaviour and results according to the actual state of refinement.

However, consideration of a certain component as a "leaf" may be only a temporal view. By the next refinement step such a "leaf" may be decomposed into other "leaves". As we do not know whether a current process remains really a leaf or has to become a composite box again, we encapsulate in SDL each such "leaf" by a SDL block even if there is only one leaf inside. This allows to do changes inside a box without effecting the outside world.

3. ACHIEVED PRODUCTIVITY

This chapter presents productivity figures obtained by practical exercises with SDL and the EaSySim CIVE. After the first pilot exercise during OMBSIM the implementation strategy was continuously improved according to the organisation rules described above. There is still a further potential for improvement. These exercises were not related to a large project. As little effort had to be spent for each phase one could not really distinguish and measure the effort per activity. Therefore a comparison with the estimated weights of section 2.2.2 is not possible.

3.1 The Modelling Task

The last exercises were performed during the ESTEC project DDV: DMS Design Validation. The task of the DDV project[4] was to establish a generic FDIR concept of an autonomous satellite and to validate it. The EaSySim CIVE, an outcome of the ESTEC project OMBSIM[5], was selected for this purpose. It consists of the SDL tool GEODE [12] and the performance simulation tool SES/workbench [13]. Based on the rules above the data communication of a fully redundant Data Management System (Fig. 7) was modelled in SDL down to automated code generation (UNIX) with an effort of about 120 man hours.

The data communication between devices and flight processors consists of

- the transfer of sensor data and actuator commands via the two data buses,
- the generation of "heart beat" signals generated by each flight processor,
- the cross-channel data link between the two flight processors,
  by which the sensor data and actuator commands are exchanged between flight processors and monitored by the supervising processor
- the cross-coupling of the devices with the two data buses,
- cyclic initiation and acquisition of sensor data,
- checks for completeness of received sensor and heart-beat signals by time-out, and
- capability for reconfiguration and fault injection.

[4]The ESTEC project DDV [2] was executed by Matra Marconi Space-France as prime and Dornier Satellitensysteme (DSS) and Verimag (Grenoble) as subcontractors. The author was project manager at DSS and performed the modelling activities.

[5] The ESTEC project OMBSIM [1] was executed by Dornier Satellitensysteme (DSS) as prime and CAST Research Centre, University of Linz (A) and Forschungszentrum fuer Informatik (FZI), Karlsruhe (D) as subcontractors. The author was project manager at DSS.

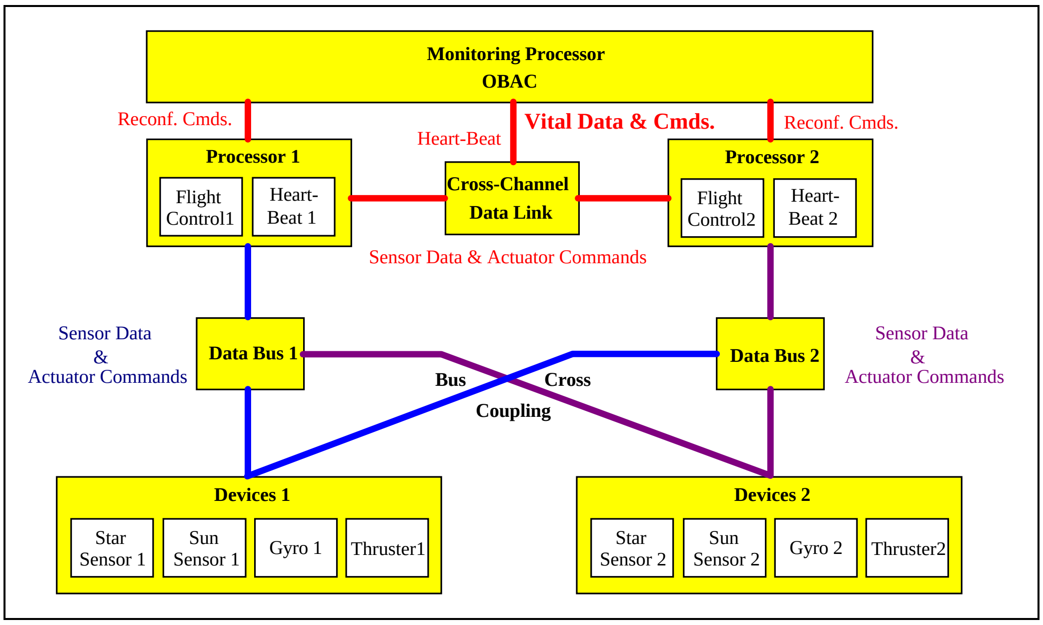


Fig. 7: Fully Redundant Autonomous Data Management System

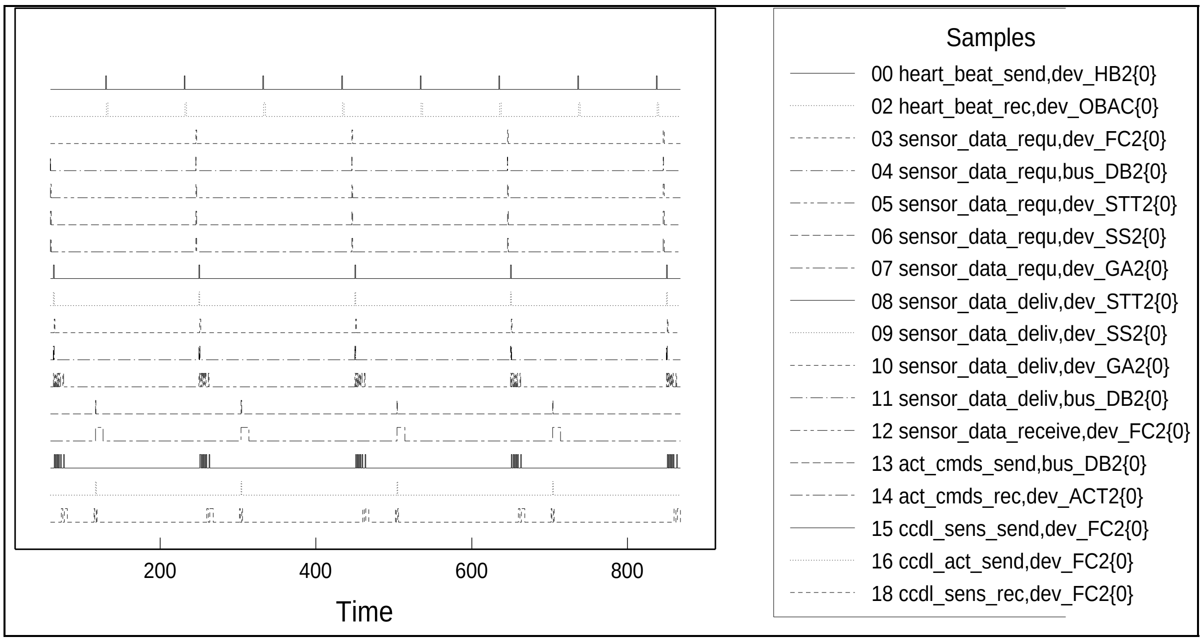


Fig. 8: Visualisation of System Behaviour by Timing Diagram

Also, the capability for fault injection and processing of test commands was implemented during this time. This allows to switch a component to a faulty state and to analyse if the system does react in the expected manner to solve the problem. By checks on consistency and completeness anomalies can be detected.

The behaviour of the system was visualised by Message Sequence Charts and Timing Diagrams (Fig. 8) which show the flow of data requests and actuator commands from the on-board control computer OBCC2 to the sensors and back to the processor, the exchange of cross link data (CCDL) and heart-beat signals.

Two versions have been considered: DDV V1 and DDV V2. DDV V1 consists of the Monitoring Processor, the CCDL, Processor 1, Data Bus 1 and a full set of devices. About 110 man hours were needed to establish this version.

The final, fully redundant version DDV V2 was derived from DDV V1 by copying[6] of the code of processor, bus and device components to the code of the redundant components. Signal channels had to be established, renaming of some variables and of the components had to be performed and the scenario file (test and operational commands) and the generation of timing diagrams had to be extended from DDV V1 to DDV V2. This took about 10 man hours until the version was completed. In total, about 120 man hours were needed to get the full model correctly running.

3.2 Productivity Figures

Achieved productivity is given by Figs. 9 and 10. However, a reader should be warned to take these figures as accurate figures. There is too much lack of background information to be able to do an exact comparison with CASE productivity figures. One should take the figures just as indicators for a trend. Fig. 9 gives figures on the application characteristics in view of size of code. Fig. 10 provides the figures on productivity.

	LOC SDL	**LOC SDL % print-stmts.**	**LOC SDL % non-functional**	**LOC SDL functional**	**LOC *.h**	**LOC *.c**	**LOC Total C**	**C LOC per SDL LOC**	**No. of Proc.**	**Funct. SDL LOC per Process**
DDV V2	5888	11.8	20.2	4700	4862	14912	19774	4.2	41	115
DDV V1	3528	11.5	20.7	2800	2975	8694	11669	4.2	24	117
PUS	1183	n/a	15.5	1000	1665	3140	4805	4.8	13	77
PAP	4507	5.8	15.7	3800	3563	11694	15257	4.1	28	136

Fig. 9: Application Code Characteristics

All Lines of Code (LOC) were counted by the word count utility of UNIX which does not distinguish between comment lines, blank lines and real statements. Therefore a rough estimation was done what the amount of blank and comment lines could be. For SDL the number of test printout statements was roughly counted (no continuation lines considered) and subtracted from the overall SDL LOC's. The considered percentage of non-functional LOC's[7] is given for each line of Fig. 9 to give a reader the information which amount of overhead was considered to get the functional LOC's.

For the LOC's of h- and C-files a 50% contribution of blank and comment lines has been considered. Some files have been inspected and it seems that this figure of 50% is a fair one. The real percentage of non-functional LOC's may even be less than 50%. The figures for size of C code are derived from version 2.1 of Geode. For version 2.2 they are higher because code variants for different platforms are included in the files.

Fig. 9 gives the lines of code (LOC) for SDL and for the automatically generated C code (executable under UNIX or pSOS), the number of (UNIX) processes and some other figures of interest.

For two other activities productivity figures are added for comparison. "PAP" is the Pilot APplication of OMBSIM [1]. PUS refers to implementation of parts of the ESA "Packet

[6] The current version of the Geode tool does not support model types and instantiation. The next version, ObjectGeode, will support this feature.

[7] print statements, blank and comment lines

Utilisation Standards" for telecommands and telemetry which was an additional activity at the end of OMBSIM.

The ratio between SDL and C code lies between 4 and 5. This does not mean that the generated code is less efficient by such a factor, but it means that the code generator does a lot for the engineer. It establishes automatically all the communication between the processes. The impact on productivity can be seen by comparison of the PUS figure with the other figures. In case of PUS a lot of processes is created with a rather small number of SDL LOC's. Creation of a process is expressed in SDL by a few LOC's only, but a number of C LOC's are automatically generated. Therefore in case of PUS the C/SDL ratio is higher than for the other applications .

Fig. 10 derives productivity figures in terms of LOC's per man hour from Fig. 9. The last column gives the sequence in which the activities were executed, i.e. PAP was the first activity and DDV V2 the last one.

For the application PAP and PUS which are already completed now the amount of man hours has

Complete Modelling[8]	**Effort in man hours**	**Degree of Reuse % (estimated[9])**	**LOC SDL[10] / man hour total**	**LOC SDL / man hour kernel**	**LOC C / man hour total**	**LOC C / man hour kernel**	**Sequence of Project Execution**
DDV V2	360	70	13.1	4.0	55.0	16.5	4
DDV V1	330	50	8.5	4.3	35.4	17.7	3
PUS	350	20	2.9	2.3	20.0	16.0	2
PAP	1500	30	2.6	1.8	10.2	7.2	1
CASE	n/a	n/a	n/a	n/a	1 .. 20 1 .. 10 [11]	n/a	n/a

Fig. 10: Productivity Figures for Modelling

roughly been estimated because the spent man hours were not documented directly[12].

For DDV V1 and V2 only the effort until completion of coding and testing is an observed figure because this is the current status. The effort for the completion of the whole task (which includes verification, documentation and acceptance) has been estimated according to experience.

[8] As DDV activities are not yet completed the remaining activities for extension of functionality and documentation are estimated to about 240 hours. Then the total effort is about 360 man hours for DDV V2 and about 330 man hours for DDV V1.

[9] The degree of reuse was just roughly estimated. The fact that the figures for the kernel yielded nearly the same value in case of versions 1 and 2 indicate that the degree of reuse (at least the ratio) was correctly estimated.

[10] functional LOC's

[11] In [14] a range of 1 .. 20 is given. However, for aerospace applications a range of 1 .. 10 seems to be more appropriate. For a prototype implemented in C according to ESA PSS-05 standards we achieved 3 LOC/man hour.

[12] The man hours for DDV V1 were derived from the DDV V2 figure by consideration of the 10 hours needed to get DDV V2 running and additional saving of time for verification and validation due to reduced complexity.

For a product the achieved productivity is the number of LOC's of the final software per man hour[13]. It does not matter how the source lines are produced: by reuse or from scratch. What is of interest for a customer is what he has to pay for a certain capability: the cheaper, the better (at same level of quality, of course).

However, as significant degree of reuse (given by column 3) was achieved one should try to separate between figures including reuse ("achieved" productivity figure) and such ones not including reuse ("normalised" productivity figure). Therefore "normalised" figures were derived in addition referring to the kernel of the application which does not include reused code. Hence, "normalised" figures give the efficiency of producing code from scratch without reuse.

To derive the normalised figures the degree of reuse was roughly estimated based on the number of reused components and reused (copied) code. For DDV V1 the reuse is estimated to about 50%, for the full model it is estimated to about 70% (i.e. size of kernel is 30% of total size). For PUS and PAP 20 % and 30% are estimated respectively. Columns 5 and 7 of Fig. 10 show the figures for SDL and C kernel code. The correspondence of the values for DDV V1 and V2 shows that the percentage of reuse seems to be correctly estimated.

The current productivity figures for a High Order Language like C or Ada lie in the range of 1 .. 20 LOC's per man hour [14]. For the type of aerospace applications considered here the figure may lie in the range of 1 .. 10 LOC per man hour, only. Looking on the figures for SDL-kernel productivity in column 6 this value was nearly achieved for PAP. For PUS, DDV V1 and DDV V2 it is significantly higher.

The only measure for the achieved productivity is the figure for the complete product under consideration of effort saving by reuse. To get a measure for the real progress by a CIVE one has to compare the LOC's of the final product which are the LOC's of C code in our case. A factor of about 4 (about 40 : 10) w.r.t. CASE may be taken as a realistic value for future projects.

Although an exact comparison is not possible the trend is evident: by better organisation and higher degree of formalization a significant increase of productivity can be achieved.

3.3 Impact of Experience and Improvements

The figures for PAP and PUS show the history of using SDL and the EaSySim environment. The PAP was the first application ever done with EaSySim and the engineer did not have experience with SDL. Add-on's and improvements to EaSySim were identified and work arounds were needed until the maintained software was available. Bugs of EaSySim had to be located and also work-arounds were needed until the bugs were removed. A first organisation of the application environment had to be introduced. This added a lot of overhead to the basic task and explains the lower productivity figure for PAP.

By PUS we wanted to get figures for the case of already experienced engineers. This exercise also should demonstrate the capability of automated target code generation. pSOS was selected as target real-time operating system. The same engineer who implemented the PAP executed this task. So the SDL implementation was rather easy. However, again a new area had to be discovered: the generation and execution of target code. So additional effort was needed to familiarise with generation of the target code and its execution.

For DDV a reusable kernel was extracted from the PAP and a small example was implemented to get this kernel running. This activity allowed to identify further bottlenecks of organisation. By the following improvements the degree of reuse could be significantly increased during implementation of DDV.

[13] The LOC's per man hour are taken as a measure of productivity regardless of the discussion whether this is a good measure or not.

Again, some overhead was observed and new experience was collected how to avoid it. The whole DDV modelling activity for versions 1 and 2 was completed in about 11 1/2 working days. The overhead may amount to about 2 working days. By the obtained feedback a further potential for improvements was identified which will be considered for the next modelling activity.

The capability of SDL tools for checking of consistency of data flow turned out as a very powerful feature which increased the productivity significantly. With pure C one would have needed probably several months to get the model correctly running.

Column 6 of Fig. 10 indicates the increase of efficiency concerning the benefit of experience and organisation[14] whilst column 5 indicates the progress due to experience only. For "organisation and experience" an increase of about 550% (55 : 10 in column 6) could be achieved, whilst the progress for "experience" is about 200% (4 : 1.8 in column 5). Hence, by appropriate organisation an increase of productivity of about 250% was achieved.

4. CONCLUSIONS

It was described what the difference between CASE and CIVE is, which improvements are needed and which increase of productivity can be achieved. Although CIVE figures cannot exactly be compared with CASE figures due to lack of equivalent and accurate information, it is evident that a significant increase of productivity can be obtained by an integrated validation approach due to a higher degree of formalization and automated code generation.

However, before this higher productivity becomes reality for projects one has to care about that

1. current development standards may not allow to use a CIVE or they may decrease the achieved efficiency by imposing an unnecessary, but significant overhead, and
2. current education and motivation of software engineers may be non-compliant with what a CIVE requires.

So we have to ask us if we are really ready for a more efficient approach. It is not sufficient to demand improvements for software development but one has also to accept to improve the contractual and educational conditions.

We have now sufficient experience down to code generation for a (UNIX) host, but we also see that we need more practice for target code generation and tuning of related procedures.

4.1 Conflicts with Current Practice

Even by the increase of efficiency shown above it is (may) not so easy to convince people for a CIVE approach. The reason is that current practice is not compliant with CIVE tools. As projects have to get approval of their results by certification authorities they have to follow the instructions defined by such authorities.

However, current development standards are closely linked to the way implementation is done by CASE. Such CASE approaches require a lot of documentation for tracing between life cycle phases and review of results. By a CIVE approach such activities can be automated by tools. E.g. tools can check interfaces between elements of specification and design and correctness of behaviour. Hence explicit tracing made visible by documentation is not required any more because tools will flag inconsistencies.

Review of specification or design documents is not needed any more because a real feedback from the system-under-development is already available. Extrapolation from paper work to final

[14] improvements of tool environment, formalization, reuse, removal of bottlenecks and traps, and code generation

operational results, a procedure bearing high risks, is no longer needed. But current development standards request such procedures.

It would be counterproductive for a CIVE approach if one would insist on the current type of documentation and review procedures which are strongly related to an inefficient life cycle approach and to ignore the more reliable and more efficient outcome of a CIVE.

Therefore it is a "must" to reconsider existing development standards when one goes from CASE to CIVE. Otherwise one prevents that engineers will apply the more efficient approach because they fear that their results are finally not accepted or they loose productivity due to (manual or semi-automated) conversion of CIVE results to a form compliant with current CASE standards.

4.2 Education and Motivation

The essential increase of productivity was only possible by rigorous organisation and formalisation, by sufficient experience and by consequent evaluation and improvement of observed weakness. This requires discipline and motivation from an engineer.

It cannot be expected that above productivity figures can be achieved just by buying a CIVE. It is like for driving a car: everybody can do it, but one has to learn it. And if one wants to win in "Formula 1" one has even to do more. But this is true for every profession.

In view of formalisation and organisation an engineer has to accept that "software engineering" is not a matter of "fine arts" but a real engineering discipline which requires to apply certain rules and continuos improvements of such rules to make progress.

If e.g. one uses a robot in car manufacturing one does not just buy this tool. But one thinks about how this tool can efficiently be used in the given specific environment. The procedures of tool usage are optimised. Also, engineers are trained for identification of bottlenecks.

In case of software engineering one buys a tool or applies a method and everybody thinks this is sufficient. But nobody thinks about how its usage can be optimised. One just waits until a tool provider does that for each customer. Wouldn't it be much better if each customer would care about optimisation of the development procedures for his own applications?[15] This was the motivation we had in OMBSIM when we enhanced tool capabilities and development steps by small pieces of add-on software.

Finally, let me conclude with another comparison between software engineering and car driving. One or two decades ago only a few people could drive at a speed higher than 100 km/h due to lack of experience or car capabilities. Today, everybody can drive a car at a higher speed. And the progress in "tools" is such that we need speed limitations.

The decrease of needed training and increase of "tool" capabilities is what could become reality for software engineering as well, at least it should be like that. For software engineering we have "speed limitations" since the beginning due to lack of tool capabilities and "driving" style. This should change when using a CIVE. Hopefully, we will never have such "speed" limitations at the end by a regulating decree as for car traffic.

[15] Of course, this does not apply to every customer. It is only of relevance for such customers who can really take benefit of better organisation, i.e. the effort of improving organisation is less than the saved development effort.

6. REFERENCES

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

[2] DDV (DMS Design Validation), ESTEC contract no. 9558/91/NL/JG(SC), Formal Methods and Tools, Selection & Justification Report, TR/171/PhH/95, 30.06.95

[3] ITU, Recommendation Z.100, Specification and Description Language, SDL, 1989, Geneva. Blue Book, Vol. X.1, and appendices A, B, C, D, F1, F2, F3

[4] ITU, Recommendation Z.100, Specification and Description Language, SDL, 1993, Geneva. Blue Book, Vol. X.1, and appendices A, B, C, D, F1, F2, F3

[5a] LOTOS: A Formal Description Technique based on the Temporal Ordering of Observational Behaviour, ISO - International Standards Organisation, TC97/SC21, 1988, Report IS 8807

[5b] J. Quemada, L. Pires, J. Ma'nas, A. Azcorra, T. Robles: "Indroduction to LOTOS", pp. 47-84 in "Using Formal Description Techniques", Ed. K. Turner, John Wiley 1993, ISBN 0-471-93455-0

[6] The RAISE Specification Language, The RAISE Language Group, Prentice Hall, 1992

[7] D. Harel: Statecharts: "A visual formalism for complex systems", Sci. of Comput. Prog., vol 8, pp. 231-274, 1987

[8] J. Spivey: The Z Notation - A Reference Manual, Prentice Hall, 1989

[9] Cliff B. Jones: Systematic Software Development Using VDM, Prentice-Hall, 1990

[10] R.Gerlich, Ch.Schaffer, Y.Tanurhan, V.Debus: EaSyVaDe / EaSySim: "Early System Validation of Design by Behavioural Simulation", ESTEC 3rd Workshop on "Simulators for European Space Programmes", November 15-17, 1994, Noordwijk, The Netherlands

[11] R.Gerlich, Th. Stingl, Ch. Schaffer, F. Teston, G. Martinelli, Y. Tanurhan: Use of an Extended SDL Environment for Specification and Design of On-Board Operations, Systems Engineering Workshop, November 28-30, 1995, ESTEC, Noordwijk, The Netherlands

[12] GEODE SDL-Tool, Verilog, 150 rue Vauquelin, F-31081 Toulouse Cedex, France

[13] SES/workbench, Scientific and Engineering Software Inc., Building A, 4301 Westbank Drive, Austin, Texas, 78746-6564, USA

[14] Max Schindler: Computer-Aided Software Design, John Wiley Sons, New York, ISBN 0-471-50650-8