An Alternative Lifecycle Based on Problem-Oriented Strategies

International Symposium on
"On-Board Real-Time Software"
ESTEC, Noordwijk, The Netherlands
November 13 - 15, 1995

Rainer Gerlich

Carsten Jorgensen

CRI
Bregneroedvej 144
DK-3460 Birkeroed

e-mail: gerlich@t-online.de

An Alternative Lifecycle Based on Problem-Oriented Strategies

Rainer Gerlich

email: gerlich@t-online.de

Carsten Jorgensen
Space Division
CRI
DK-3460 Birkeroed

ABSTRACT

Up to now, quality of on-board software is achieved by tailoring the software for each application and by its final tuning to the needs of the environmental constraints like timing and sizing. A significant part of the effort has to be spent for testing at the end of the lifecycle. Hardware and software is specified separately and a system's capabilities become visible at the end, when it is usually too late for corrective actions. Today on-board users demand more functional capabilities, because processors are able to deliver the needed processing power. Therefore systems, and especially software, become more complex. Consequently, traditional development procedures are becoming inadequate, because they require too much effort for fine tuning and testing and do not consider the iterative nature of system specification and design. In all areas of software development, not only in the field of on-board software, new development methods and lifecycles are discussed. But it is felt that the required quality of on-board software and related higher effort makes it a leader concerning efficient development of quality software. Recognizing this ESA has initiated numerous activities investigating new methods and approaches for satisfying the high demands put on reliable on-board software.

We will in the following describe the problems concerned with the development of real-time embedded software in general and on-board software in specific. The activities of real-time development and required tool support will be highlighted and an alternative lifecycle will be proposed. The descriptions and proposals made will capitalize on individual ESA/ESTEC activities in the on-board domain, results and practices in the domain of avionics and safety critical software, and on experiences gained from utilizing these in operational projects.

1. INTRODUCTION

Embedded systems in general (i.e. hardware and software) are broadly used for control and data management. A large number of such applications is safety critical and is required to be highly reliable. With increasing needs (Fig. 1) for more functionality and performance, embedded systems are becoming more complex and the amount of functionality implemented in software is increased significantly. Furthermore, the higher the percentage of software in the system, the larger its potential impact will be in case of faults. Consequently, we have to assure that software will be of better quality in the future.

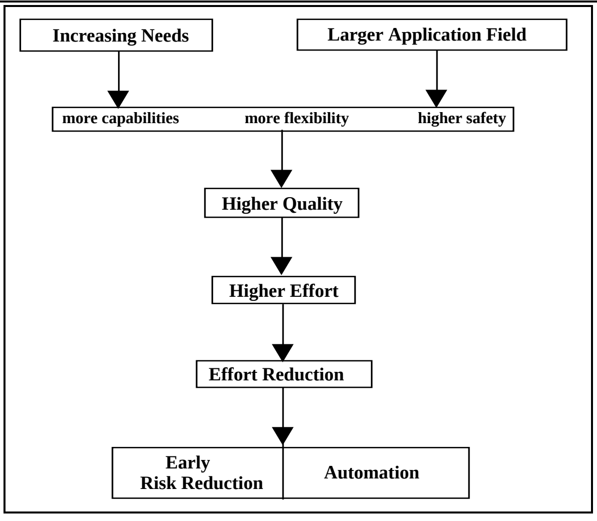

Fig 1: Increasing Needs vs. Efficiency

However, the current situation in software development in general and on-board software in specific may be characterized by:

(1) some few activities during the specification phase,

(2) a number of activities during design and coding phase including corrective actions by iterations,

(3) a huge number of activities for system integration and validation at the end of the development lifecycle.

Currently, tools are mainly used for syntactical checks of design and code. Before the coding phase usually no validation of the resulting behavior and performance of a system's implementation is done. Reuse is still an issue. Most of the used tools just help to ensure consistency between components used during a certain lifecycle phase, but not to achieve consistency between lifecycle phases. They neither do help to confirm in an early lifecycle phase, that the intended implementation will represent what the customer (expressed in the mission requirements) has in mind and that it fulfills all his expectations. The major tasks, coding and testing, needed to achieve the desired goal, are still done manually.

Complexity of systems is increasing, but the maturity of the software development approach is still not appropriate to meet the needs of (future) high quality on-board software. With increasing complexity the costs will explode due to the still low degree of automation of software development. Compared with other engineering disciplines "software engineering" is still done like "fine arts" leaving much freedom for experimenting with new ideas rather than trying a systematic development approach. Methods and tools are more supporting verification of syntactical entities than validation of a system's properties like functionality, behavior and performance.

Consequently, from this point of view more emphasis must be put at the beginning of the lifecycle on understanding of the problem and on fixing the desired properties by getting a feedback from what is defined. Tools must help to concentrate on such properties. Direct code generation from specification and design is more efficient than just to automate the transition from design to manual coding. Code generation is a very good area where tools can do the job, no corrective action is needed any more when by specification and design the behavior is already fixed and tests are defined which prove the desired capabilities.

Hence, producing reliable or say trustworthy real-time systems is a complex task that calls for numerous means to be applied during its construction. Fundamentally, two elements are involved in this scheme, namely

(1) the real-time development process, and

(2) the real-time software products to be delivered via the activities of the process.

Accordingly, higher software quality can be achieved by

(1) improvement of the real-time development process itself, by for instance dictating a certain data and control flow of the activities in the process model thereby contributing also to a better sub-product but also to optimization and risk reduction.

(2) improvements of the quality of the sub-products generated during the route of development activities by requesting appropriate methods, standards, and tools. Ffor instance, dictating a specific method/tool to be used in the design activity leading to a quality design document (i.e the sub-product).

In the past, ESA/ESTEC initiated a number of activities investigating new approaches and introducing new techniques, especially for on-board software that should support the above two main elements and lead to a more reliable end-product being the “executing on-board software”.

Firstly, such activities covered feasibility of specification, design or implementation like schedulability [1] and performance analysis [2]. Secondly, they addressed architecture and stability of design [3]. Thirdly, optimisation of development lifecycle was initiated [4]. The experience gained in the past by separate projects [2,5,6] has now to be synthesized towards an efficient integrated approach, integrated via the lifecycle.

The challenge is to consider software development as a real "engineering discipline" rather than a matter of "fine arts". Such a rigorous approach requires more precise rules similar to "cooking rules" for system / software development and its management rather than weak guidelines leaving a lot of freedom for ending in inefficiency. It requires a rigorous procedure for streamlined implementation. Validation shall be a matter right at the beginning. Consequently, this leads to specification-based implementation: deriving an implementation directly from specification in a staggered approach allowing at each stage to verify that one is still on the right way to the envisaged goal.

2. CURRENT REAL-TIME SYSTEM DEVELOPMENT

We will in this section give a short description of the activities and the phases of current real-time software development arising from results of previous ESA/ESTEC activities together with experiences from the domain of avionics.

Fig. 2 presents the basic structure of the current software life-cycle. It represents the classical "waterfall approach" and requires hidden iterations between a number of activities and procucts. Typically, each of the development phases is applied to the whole system. This yields lifecycle products like User Requirements, Software Requirements, Test Specifications, Architectural Design, Source Code and finally the Executable Code.

For each product a consistency check has to be performed in order to ensure compliance between input and output of each phase. Fig. 3 shows in detail the sequence of the phases, compliance checks and method and tool support. "S" means "Software Requirements Specification", "V_e" represents "Verification, Consistency Checks", "D" stands for "Design", "C" for Coding", "MT" for "Module Testing", "IT" for Integration Testing" and "AT" for "Acceptance Testing". Finally, the validation "V_a" of the system against user requirements is performed.

Typically, during the specification phase methods and tools like HOORA [7], CORE [8], Yourdon (SA) [9], Coad [10], SDL [11,12,13], RAISE [14]), VDM [15] or Z [16] are used. In view of correctness, reliability and safety formal notations like SDL, RAISE, VDM or Z would be preferable.

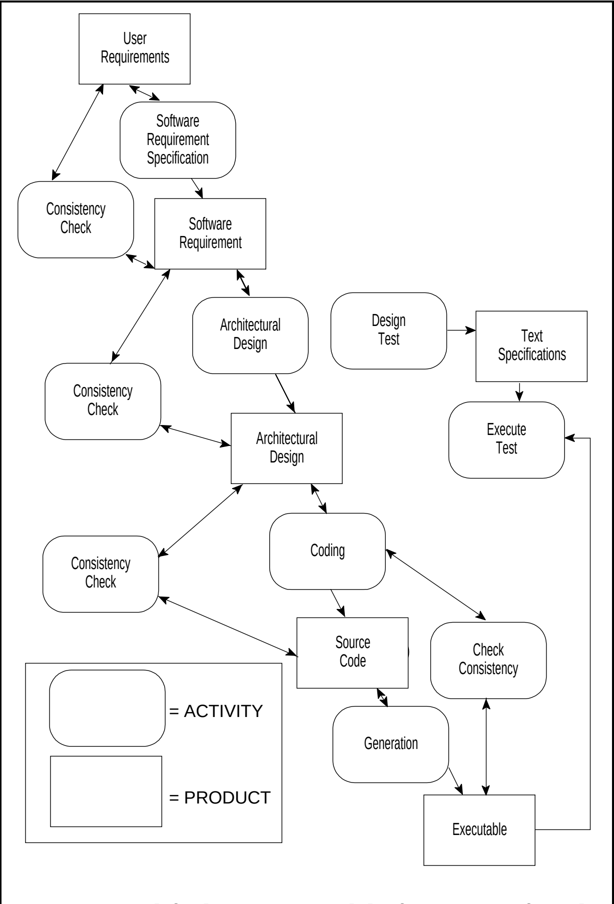

Fig. 2: Simplified Process Model of Current Lifecycle

During the design phase(s) HOOD and HRT-HOOD (for hard-real time systems), Yourdon (SD), SDL and RAISE may be applied. In general, we propose formal notations to be used.

For coding languages like C, C++ or Ada are used. Ada should be the choice for safety-critical systems.

Testing is supported by a lot of tools and test software. Ada has significant advantages for module testing e.g. test of subprograms against their specification in an automated manner [17], and for integration testing due to its checking capabilites across library units.

Tests of real-time software are carried out in two different environments: the host environment and the target environment. Again, the target environment may consist of two different types: the real target and of what is known in the space environment to be a Software Validation Facility (SVF). Such a facility encloses a board with the target processor, memory, and I/O together with a fan of services making it possible to

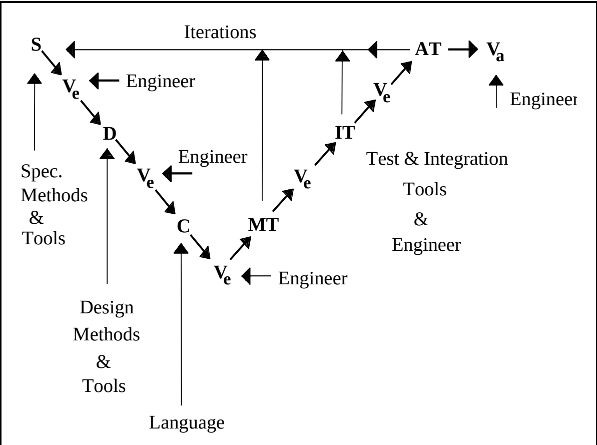

Fig. 3: The Sequential Lifecycle

control the tests (especially the clock) and having the environment (seen from the real-time software) look like the "real thing". This also offers the potential to apply fault injection techniques may be applied thereby providing a strong means for approaching verification of non-nominal conditions.

A prominent SVF configuration allowing for the above, is the SVF used in the Cluster and SOHO projects [18]. This kind of configuration makes it possible for every developer to have a high fidelity test platform on table.

In the traditional development approach methods and tools as mentionned above do not care about performance of (real-time) software, i.e. if the timing constraints can be fulfilled. Performance not only addresses processor power, but also performance of a (on-board) data network.

Currently, checks on performance are mostly done during testing and validation phases in a target envrionment which is rather late in the lifecycle.

Verification between lifecycle phases is done by engineers, methods or tools do not support a consistent and coherent transition between the phases.

3. COST ISSUES

To summarize, in the current development approach a number of verification activities have to performed, mostly by engineers and not well supported by methods and tools. This creates significant overhead. Moreover, validation cannot be done early in the lifecycle as the needed means like executable code or the target system / SVF are not available. This increases the risk to end up with insufficient system capabilites which is especially a problem for real-time systems.

Iterations are a consequence due to better understanding of a system when feedback is given by system execution. They are needed because nobody is able to have the full overview on final needs and one only can approximate them by smaller steps. This is true especially for complex software systems. However, as a system cannot be executed before end of coding phase, such improvements are not well supported in early phases of the current development approach.

To be prepared for the future firstly identification of the weakness of current system and software development methods is needed concerning remaining risks and effort. Secondly, one has to provide an alternative. It is essential to consider optimisation of the total system lifecycle and not only to concentrate on its dedicated phases. Also, task sharing between an engineer and tools must be reconsidered.

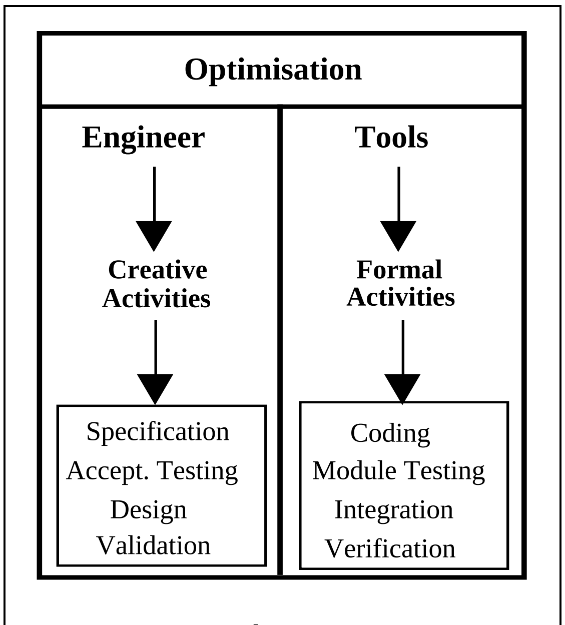

Fig. 4: Towards Cost Optimization

To achieve higher efficiency optimisation of task sharing between engineers and tools is a "must" (Fig. 4). An engineer shall concentrate on the creative part of system development, whilst tools shall take that part which can be formalised and automated. Therefore an engineer shall be more involved in early development phases and less in the later ones: this leads to early validation and risk reduction by simulation, hierarchical and stepwise refinement with execution of nominal and non-nominal conditions and increased degree of automation by use of formal methods and code generation (Fig. 5).

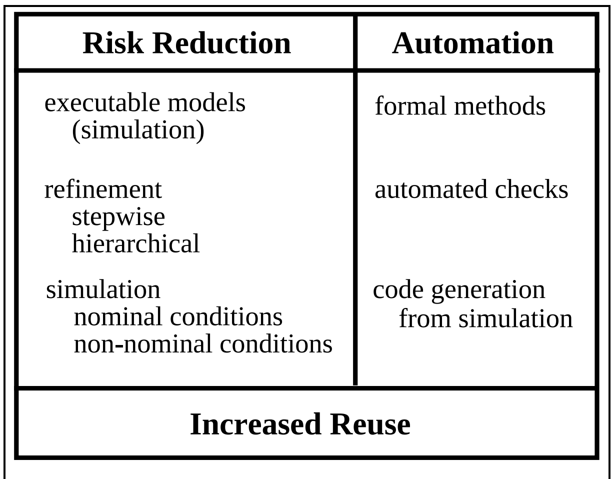
Risk Reduction
Automation
executable models (simulation)
refinement stepwise hierarchical
simulation nominal conditions non-nominal conditions
formal methods
automated checks
code generation from simulation
Increased Reuse

Fig. 5: Keypoints of an Alternative Lifecycle

Currently, it is just the other way around. An engineer spends a lot of time for coding, module testing and integration during rather late development phases.

In consequence, for such an optimised lifecycle effort is decreased for the following reasons:

- By incremental and early validation the expected system properties are known and continously be refined. The system is validated functionally and for performance. By executable models nominal and non-nominal scenarios can be simulated and evaluated. This strategy decreases the risk not to meet the desired goal.
  The earlier a non-compliance is detected, the less costs are required to remove it.
- By use of formal methods tools are able to detect automatically discrepancies in a system like incompatible interfaces or wrong behaviour. Also, automated code generation becomes possible.

  Efficiency of development is increased by the higher degree of automation and costs are reduced.

The higher the degree of formalisation is, the higher is the percentage which can be covered by tools.

Quality is increased because more development faults are detected earlier and automatically to a higher percentage by optimisation of tool usage. By automated code generation from already validated models no faults are introduced like in case of manual coding according to a separately available design.

The percentage of reuse increases because by standardisation and formalisation similarities can be better and earlier identified. If needed, properties, behaviour and interfaces of the new components can be harmonised with such ones of the already existing components.

Due to formalisation the verification steps can automatically be performed by tools, whilst the validation is an engineering task supported by tools for execution and result analysis.

In consequence, the possibilites to correct development in early phases are better. Tools are used for such tasks for which their usage gives highest benefit. This allows to save costs.

4. AN ALTERNATIVE LIFECYCLE

The former chapters have gone through the stages of a development lifecycle focusing on the approach currently applied and identifying potential improvements.

Now, a process model shall be defined which glues together a system's hierarchical decomposition and the previously described lifecycle phases. These phases are taken in a generic manner and applied to each decomposition step. The activities of each such phase are harmonized and optimized by introduction of appropriate methods and tools.

Considering the lifecycle as a whole, two important issues need to be considered:

- the transfer from one stage to another and
- the required iterations.

The transfer issue has to do with how well the linkage is between the methods used at one stage and at the following one. Clearly, to go from one stage to another

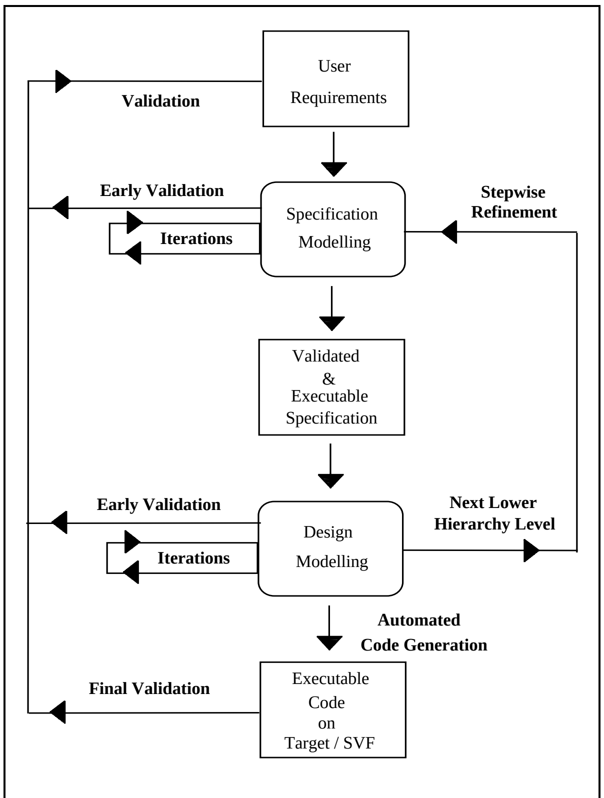

Fig.6: Simplified Process Model of an Optimized Lifecycle

should be achievable in a controlled manner. Notations and notions used on two connected stages should be compatible. For instance, if the underlying concurrency model of the design method is not compatible with the one embedded in the programming language, the activities of the coding stage would be both troublesome, and likely to be error-prone.

The method and the techniques used at one stage of the lifecycle should be built on an understanding of what is feasible at the next stage. Only in this way, the construction of real-time software may be viewed as a series of transformations. A view, that will make it possible for managing and controlling the process.

Fig. 6 shows for the proposed alternative lifecycle the view equivalent to Fig. 2. The keypoints are:

(1) Stepwise refinement.

(2) Combination of specification with coding and of design with coding.
This leads to executable models in both phases with the option for production of target code (if supported by a tool).

(3) Validation against user requirements by execution of the models.

(4) Iterations in each step if non-compliances or improvements are identified.

Starting bottom up, the plateau that glue together the top-level software requirements with the executing system, is the programming language Only by understanding what can be achieved at this plateau, appropriate design approaches can be taken. Consequently, if from the very beginning, the programming language is given, one should choose a design method that in fact could be glued to the possibilities offered by the programming language.

In the scope of real-time systems, it becomes mandatory that iterations are introduced, giving early feedback on how performance and resource constraints may be met. Some early hands-on experiences may be required to know what is possible and how this may best be achieved. Consequently, rather than addressing design horizontally (i.e. design in breadth), where the complete system is approached and refined by a stage-by-stage single-shot waterfall approach, design should enclose some vertical activities (i.e. design in depth).

Hence, a lifecycle should be arranged according to early and incremental validation and delivery. We have in the previous sections highlighted a set of methods and tools that should be used today in each of the mandatorial development phases. These are tools that brings a high degree of automation into the development and in combination will increase the quality of the on-board software.

The lifecyle shown in Fig. 6 will encourage to spend more effort on the specification and design phases. It considers the iterative nature of system development, and it identifies problem-oriented methods and tools and when they are to be applied.

The quality is increased for the following reasons:

(1) Having executable incremental models nominal and non-nominal scenarios can be simulated/animated and thus evaluated. This will decrease the danger of not achieving the desired goal.

(2) By using formal methods, tools are able to automatically detect discrepancies in a system such as incompatible interfaces or wrong behaviour. Automated (target) code generation becomes possible and development efficiency is increased by

this high degree of automation. Clearly, the higher the degree of formalization the higher the percentage of development which can be covered by tools

(3) More development faults are detected earlier and automatically . When using automated code generation (either fully (which is the ideal case) or partially) ensures that no faults are introduced as might occur in the case of manual coding.

(4) The percentage of reuse increases, because by standardization and formalization, similarities can be readily identified early. Moreover, needed properties of new components can be harmonized with ready existing ones.

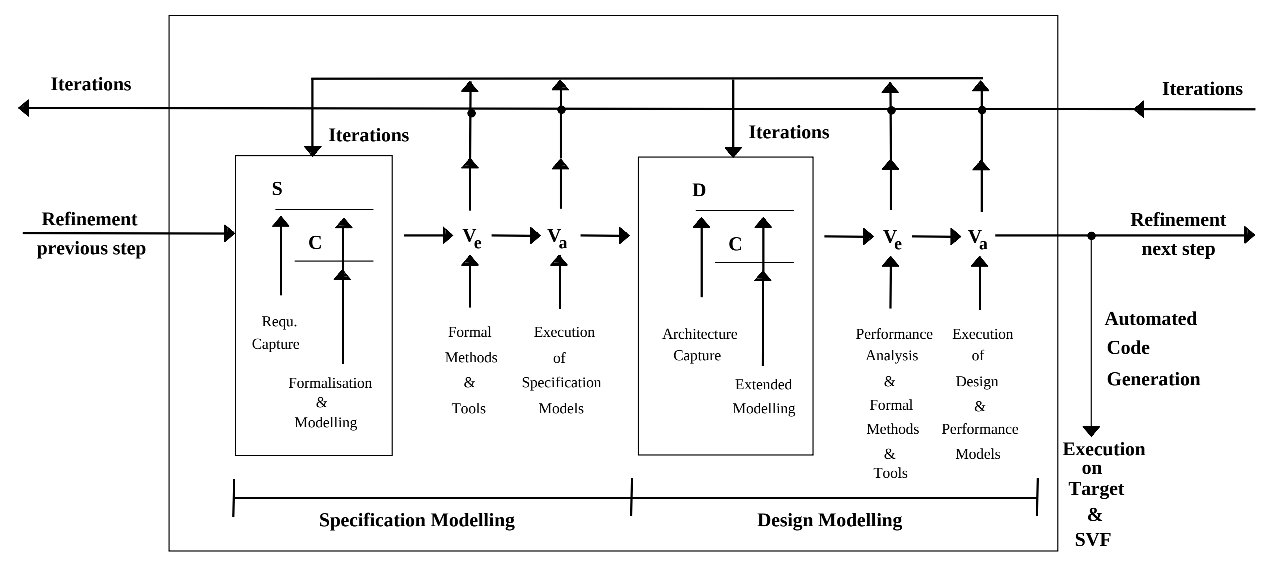

Fig. 7: The Incremental Lifecycle

Fig. 7 shows in more detail what is performed during each of successive development steps.

For each such step "Specification Modelling" and "Design Modelling" are performed. Specfcation Modelling consists of four and Design Modelling of six sub-steps. Fig. 8 identifies the actionees for the sub-steps and relates them to example tools. It is divided into two parts: specification and design. The six sub-steps are:

(1) A brainstorming sub-step for "Requirements Capture" and "Architecture Capture" during which methods like HOORA and OMT for specification and (HRT-)HOOD and performance analysis tools for design may be applied.
Such an activity may be needed especially for the top-level to get unstructured textual requirements in a form which allows formalisation by models.
Schedulability and performance constraints may be identified in case of design.

(2) The following "modelling" activity is the cornerstone of the approach. It combines specification and design with formalisation and modelling. Hence executable code is produced which allows animation and an immediate transition to the target system or SVF in case of leaf nodes of the hierarchy.
Specification models represent the functional and behavioural view. Design models take the interface of specification models and refine their internals with an architecture and additional (internal) models which specifiy the requirements for the next level.
In many cases one will find that the structuring of textual requirements (sub-step 1) is actually part of sub-step 2 by directly applying the modelling tool.
It is also possible to generate code from specification models for target or SVF, if desired.

(3) The produced code represents also a formal description of functionality, behaviour and performance. This allows

tools to verify properties like consistent interfaces, correctness of behaviour and state transitons, schedulability.
Most languages and tools can check consistency of data interfaces and types. Only a few tools are on the market which check behaviour, state transitions and schedulability.

(4) By execution of the models under nominal and non-nominal conditions the actual design and code is validated for functionality and behaviour against user requirements. In case of design modelling validation of performance is done in addition. Hence, at the end of each decomposition step possible non-compliances are identified and not at the end of system development.

(5) The fifth sub-step is not really an activity, because it is done automatically by tools: code for target or SVF may be generated automatically from the validated models, if tools support it.

(6) Finally, the sixth step deals with execution of the already applied nominal and non-nominal (if possible on the target or SVF) scenarios on the target or SVF and shall confirm the previous results of simulation.
This sub-step may not necessarily be performed at the end of a decomposition step. It depends on the criticality of a designed component and the availabilty of target and SVF whether it can be earlier.

Fig. 8 has identified different tools each supporting the sub-steps of the presented alternative lifecycle. These tools could serve as building blocks for an implementation of a toolset for this lifecycle.

For instance, by integration of GEODE, SES/workbench [19] and additional software the resulting tool environment EaSySim [20] covers most of the steps of the alternative lifecycle. Other alternatives can be retrieved from the example list of Fig. 8.

5. CONCLUSIONS

The presented lifecycle recommends

- incremental development,
- a coherent transition between specification and design,
- modelling as a combination of specification/design and coding,
- early validation against user requirements by execution of models,
- higher tool support for verification by increasing degree of formalisation.

Due to risk reduction, automation and combination of specification/design with coding a reasonable potential for cost saving is identified. The tool environment EaSySim is available which covers most of the activities of the alternative lifecycle.

This software lifecycle can easily be extended towards a system lifecycle for embedded systems. Then a unique approach for hardware-software co-design becomes possible which allows a late hardware-software trade-off together with early validation of system properties.

REFERENCES

[1] Hard Real-Time System Kernal Operating System, ESTEC contract no. 9198/90/NL/SF, Final Report 1993, Noordwijk, The Netherlands

[2] HRDMS (Highly Reliable DMS and Simulation), ESTEC contract no. 9882/92/NL/JG(SC), Final Report 1993, Noordwijk, The Netherlands

[3] HOOD (A Design Method for Ada), ESTEC contract no. 6887/86/NL/MAC, Final Report 1987, Noordwijk, The Netherlands

[4] Real-Time System Development Strategy, ESTEC itt no. AO/1-2935/94/NL/JG , SOW, Noordwijk, The Netherlands

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

[6] Advanced Methods and Tools in the ESSDE, ESTEC contract no. 9598/91/NL/JG(SC), Final Report, Noordwijk, The Netherlands

[7] HOORA - Hierarchical Object-Oriented Requirements Analysis Reference Manual, E2S/OORA/WP1/REF, Issue 1.4, E2S, 6/10/1993

[8] G.P.Mullery: CORE - A Method for Controlled Requirement Specification, 4th ICSE 79, pp.126-135

[9] E.Yourdon, L.L.Constantine: Structured Design, Englewood Cliffs, New Jersey, Yourdon Press, 1979
E.Yourdon: Modern Structured Analysis, Englewood Cliffs, New Jersey, Yourdon Press, 1989

[10] P.Coad, E.Yourdon: Object Oriented Analysis, Engelwood Cliffs, New Jersey, Yourdon Press, 1990

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

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

[13] SDT, TeleLogic AB, PO Box 4128, S-20312 Malmö,Sweden

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

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

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

[17] R. Gerlich, G.Fercher: A Random Testing Environment for Ada Programs, Proceedings of EUROSPACE Symposium "Ada in Aerospace", Brussels, Belgium, November 1993

[18] S. Mejnertsen, K.Hjortnaes, P.Holiday, S.Ekholm, J.Gujer: Software Validation Facility for On-Board Software

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

[20] 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

Phase	**Sub-Step**	**Actionee**	**Activity**	**V&V Subject**	**Tool Examples**
specification	requ. capture	engineer	informal, structured requirements		HOORA, OMT
	formalisation & modelling	engineer	building of specification models		EaSySim, GEODE, RAISE, SDT, VDM, Z, LOTOS Ada, C, C++
	verification	formal methods & tools	checks by formalisation static properties	interfaces types	EaSySim, GEODE, RAISE, SDT, VDM, Z, LOTOS Ada, C++
			dynamic properties	behaviour state transitions	EaSySim, GEODE, RAISE, SDT,
	validation	engineer	model execution nominal conditions	operational scenario	EaSySim RAISE-Ada translation
			non-nominal conditions	fault tolerance concept by fault injection	EaSySim

Fig. 8/1: Specification Steps

Phase	Sub-Step	Actionee	Activity	V&V Subject	Tool Examples
design	architecture capture	engineer			EaSySim (HRT)-HOOD
	extended modelling	engineer	refinement to design models		EaSySim, GEODE, RAISE, SDT, VDM, Z, LOTOS Ada, C, C++
	verification	formal methods & tools	checks by formalisation static properties	interfaces types	EaSySim, GEODE, RAISE, SDT, VDM, Z, LOTOS Ada, C++
			dynamic properties	behaviour state transitions schedulability	EaSySim, GEODE, RAISE, SDT, HRT-HOOD toolset
	validation	engineer	model execution & performance analysis nominal conditions	operational scenario	EaSySim
			non-nominal conditions	fault tolerance concept by fault injection	EaSySim
	code generation	tools	automated code generation		EaSySim, GEODE, RAISE, SDT
	testing	engineer & tools	acceptance testing	actual implementation	EaSySim, GEODE SVF

Fig. 8/2: Design Steps