Organising Incremental, Reusable and Automated Software Development

'DASIA 99'

- Data Systems in Aerospace -

Lisbon, Portugal

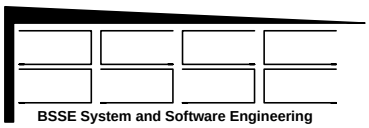
May 17-21, 1999

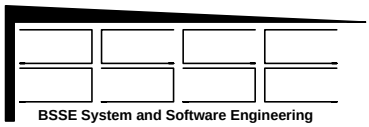
Rainer Gerlich

BSSE System and Software Engineering

Auf dem Ruhbuehl 181
D-88090 Immenstaad

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





Organising Incremental, Reusable and Automated Software Development

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 internet: http://home.t-online.de/home/gerlich/

Abstract: Saving of development costs and time has been an issue in the past and still will be a future challenge. Amongst others the following key points will help to approach this goal: (1) early reduction of risks, (2) reuse and (3) automation. The organisation scheme described here harmonises risk reduction and reuse and allows for automation of development steps. At start of a project it allows for immediate setup of an inexpensive executable prototype which can be extended by incremental development to the final version. Reuse is supported in twofold manner: across projects and across lifecycle phases. As an executable prototype is immediately available (within about 15 to 20 minutes after provision of the engineering information) an early identification of risks is possible. The approach is based on formalisation of the software structure and the organisation of development steps, and standardisation of internal and external software interfaces. It is driven by ideas coming from SDL [1] and related verification techniques, object-oriented and formal methods and experience with a number of projects investigating early system validation [2-4] and follow-on projects by which the idea was refined and improved [5-7]. The standardisation of interfaces supports easy change of system topology, redundancy switching and transparent distribution of system components across a network independently of the physical media.

Keywords: automation, incremental development, reusable software, early system validation, standardisation, software interfaces, software integration, distributed systems, software development process

1. INTRODUCTION

Since a couple of years activities have been performed for ESA/ESTEC to improve system development and especially software development by early system validation [2-4].

The EaSyVaDe approach as defined by [3] is based on incremental development starting with a representative prototype which is more and more refined towards the final version. Such a prototype is either executed by a simulator or on the target system to get an immediate feedback.

During all these projects attempts have been made to establish components which may be reused from project to project and valueable feedback was collected which could be used to improve the reusability of the components.

By the project HRDMS [2] a first standard for module interfaces was introduced but it was too much bounded to the simulation environment and could not be reused for target implementation in the course of OMBSIM [3]. OMBSIM concentrated on the verification and validation means as provided by SDL tools [8,9]. As the SDL verification tools require the use of dedicated interfaces between the components for documentation purposes, no standardised interfaces were introduced. During DDV [4] a step towards interface standardisation in SDL was made but it yielded a graphical representation which was difficult to read, and the code needed to be instrumented heavily.

During these exercises it was recognised that a system's topology has to be hardcoded in a tool's environment[1] which made it difficult to deal with reconfiguration of hardware after a fault like switching to a redundant bus. Such hardcoded communication lines also require a significant effort during development in case of a change and nearly prevent reuse of software from project to project.

[1] This is true for all graphical tools, in general, which allow to draw data channels between components.

Moreover, it turned out to be very inefficient to implement procedures for redundancy switching for all the dedicated lines.

These problems lead to a new approach based on standardisation of data exchange formats, input and output interfaces, aqnd data-driven definition of the topology. In consequence, only a minimum of formalised inputs on engineering level is needed. The related tool environment allows for an automation of system generation and testing. As programming environments SDL/C (ObjectGEODE [8]) or pure C are supported.

2. THE IDEA

The idea for standardisation of interfaces is not new because it is already used in other areas like hardware, bus protocols or packet telemetry [10], and a similar idea was proposed in general already more than 10 years ago [11,12].

However, it seems to be a new approach concerning communication between software components like processes and subroutines. The principal idea is to introduce a "SoftBus" for intercommunication based on a standard format which carries all the information needed for data exchange and distribution, and which can be applied to every project.

This standard data exchange format includes information about sender, receiver and the type of required data processing at the receiver.

Due to the information about the receiver a routing software package can transparently forward the data across a network. This is similar to CORBA [13] but also supports a heterogeneous set of physical channels like RS232, files, screens, IPC and TCP/IP and fulfills the needs of real-time processing. A description of the data format by a user like for CORBA/IDL is not needed.

Consequently, a system and its capabilities can be defined by a set of commands, a number of components (like processes or hardware devices), a set of communication channels and resources (like CPU, buses).

This allows to formalise (a) the specification of a system's properties, (b) the organisation of a system's structure and (c) the development process, and to efficiently support a user when he is going to start a new project.

The standardisation and organisation scheme allows to introduce a generic, reusable architecture for transparent distribution which also supports management of redundant components. An I/O layer decouples (in object-oriented manner) the data transfer from data processing and allows to implement any communication topology by data.

The organisation allows for (a) automated generation of a software system for which only high level engineering information is needed (Fig. 2-1), and (b) for incremental system and software development.

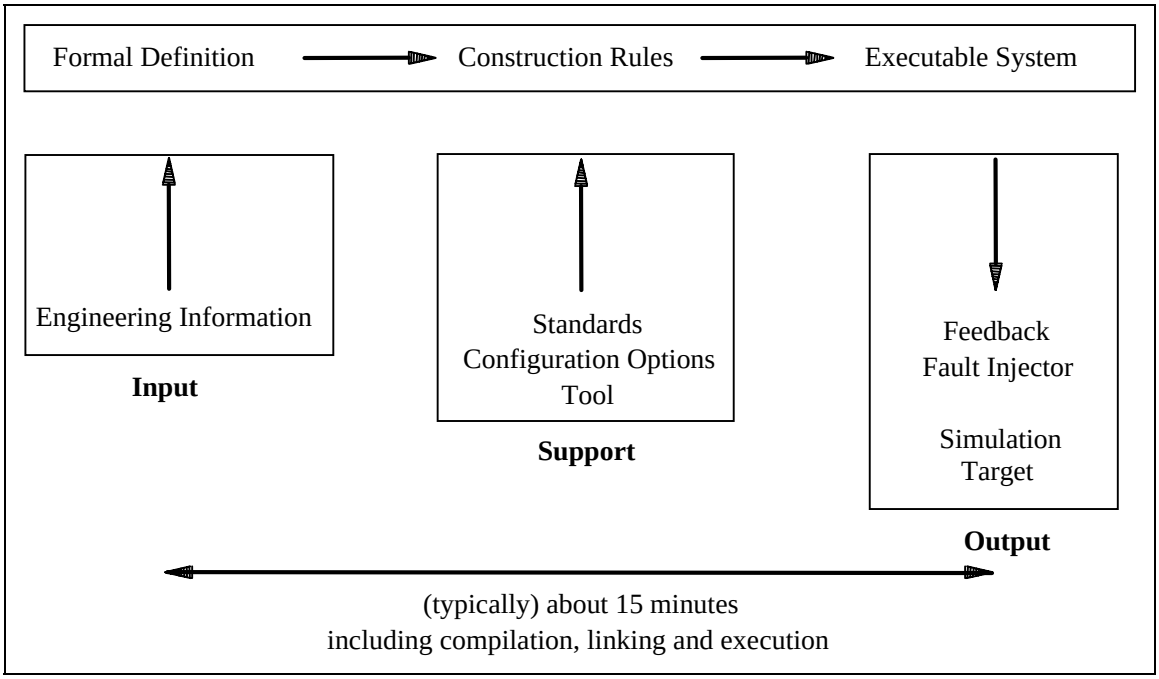


Fig. 2-1: Automated and Instantaneous Construction of a System

2.1 The Data Exchange Format

Two types of formats are introduced to exchange messages:

- a "short" format which only carries a command together with a few data only
- an "extended" format which includes additional ("attached") data in a user-defined format.

The "short format" includes the following information:

- information about the sender
- information about the receiver
- information about what the receiver shall do on reception of the message ("command")
- priority of the message
- short information contents (some data)
- format type (short, extended, format of attached data/information)
- length of the complete message (including the attached data)
- timing information
  e.g. when the message was sent initially or actually (in case it passes several modules).

The user-defined information is added as byte stream after the end of a short message.

2.2 The Communication Channels

For communication between modules like processes "logical" channels are defined which are mapped onto physical channels like buses, RS232, RS422, TCP/IP, IPC, UDP, files, screens etc.

A number of physical channels may be grouped together to form a logical channel. Hence, two (not necessarily equivalent) physical channels like two buses may form a redundant pair of channels. Arbitrary grouping of physical channels is allowed. Redundant channels are identified by adding some grouping information.

A standard transfer procedure takes the message and forwards it through all the physical channels to the destination, selectively or in broadcasting mode. This approach decouples the application specific routines from the system's topology and allows to introduce generic subroutines for data distribution which are driven by data. This results in 100% reuse of the communication functions.

The message formats may also be used for module-internal communication, e.g. to be passed from function to function as parameter. This way it is very easy to switch from internal to external communication because no interfaces need to be changed. This prevents that changes of the topology or of software-hardware partitioning will seriously impact the implemented software.

The standardised format allows also to introduce a single interface to receive messages within a process. Similar to the output procedure the receiving functions convert the (physical) input into the same logical format. This allows to start processing of data or of other events from a single location within an application process (Fig. 2-2).

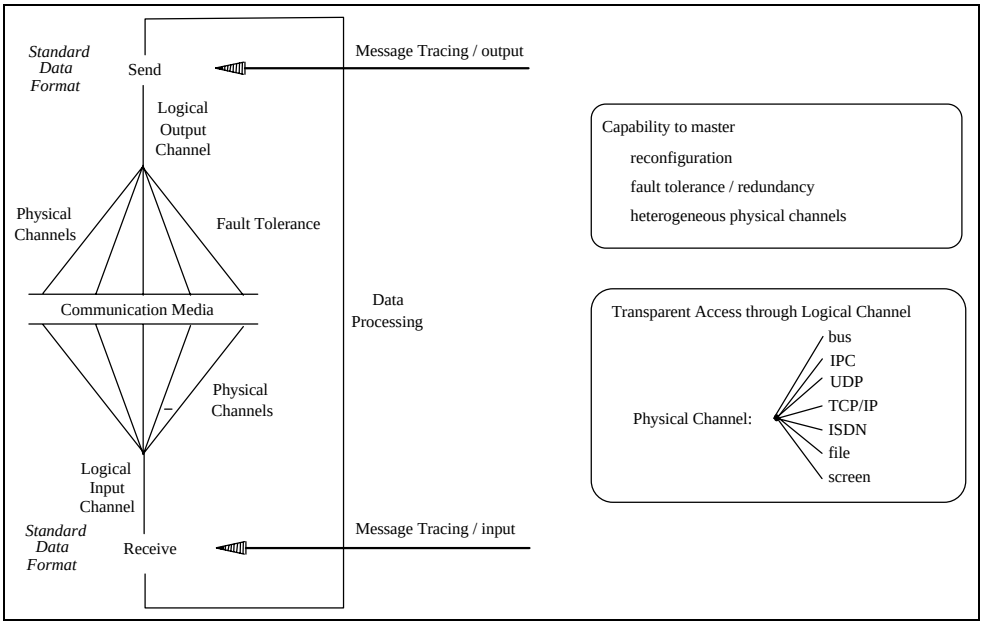


Fig. 2-2: I/O Interface Standardisation

2.3 Data Processing and Commanding

The message format includes information ("command") about what the receiver shall do on reception of the message. A sender may advise the receiver to execute some actions on reception of a message, or the command specifies which data are included and how to process them.

This allows to introduce a standard structure for a module like a process and leads to 100% reuse by formalised construction of application modules.

Such a module identifies the incoming comands and then branches to the related processing sequence. Some of the commands may be common to all applications like to open or close communication lines. Some others may be specific and are (automatically) added to the common branches in the course of incremental development.

The principal structure will be based on a case (a "switch" in C) or a set of cases which covers all the possible incoming or internally generated commands. Each branch of this case includes the corresponding actions.

This allows to provide (1) templates which can be reused for 100% by every application and (2) fixed rules how to expand the template to make it specific for an application.

The result is an extended Finite State Machine (FSM): a FSM for dedicated, state-based processing and another command processing facility for state-independent inputs.

2.4 Formalisation

By the commands which are included in each message the communication between each module is formalised in the sense that

- a set of commands can be defined which are to be processed by a module and which are legal input
- a set of output commands (if any) can be correlated with each input command.

Consequently, a system's activities are described by the data flow between the modules and the associated commands. This allows to perform automated testing based on the input-output mapping of commands and validation of the data and command flow. It is even possible to execute tests in an early development phase when the functionality is not fully implemented, but the principal functionality and the resources are known.

2.5 Outlook

The basic ideas as described above were stimulated by an SDL implementation as performed in the course of an exercise [14] related to the ESTEC round-table on "Executable Specifications" and improved from project to project. The principal organisation of a SDL process in terms of states and state transitions has driven the organisation.

In SDL a process is structured into states and state transitions. However, the same structure can be obtained in any other language. This leaves it up to the user which language he wants to apply to his project.

The benefit of SDL is that the available SDL tools provide good verification support which is not available when applying another programming language. However, by EaSySim II it is possible to interface with SDL verification tools from other languages like C.

3. THE ORGANISATION

Having already executed a number of projects the current status of organisation is:

- at start of a new project a user receives templates (currently in SDL or C), a set of library routines and sample data files which drive system configuration and define system properties,
- a user defines all the internal and external data exchange and data processing by literals,
- a user defines all the resources and their properties which are needed for performance analysis by literals, resource consumptions and timing constraints,
- the tool instantiates the modules from the provided templates and a user's data inputs,
- the tool generates stubs for user-defined (application-specific) functions. Such stubs cover resource consumption and data transfer during the first steps of devlopment.

- possibly, a user complements the automated installation procedure for the simulation and target envrionment by defining which additional software (not included in the formal definition) is to compile and to link
- finally, the tool runs a utility which generates template data files for the application which a user can adapt if required.

The needed inputs from engineering level are provided by three principal files:

- A file describing behaviour and performance of the system (per process: initial state, incoming command, outgoing command, destination, channel, final state, resource/CPU, amount of resource consumption and data to be transmitted).

  By this file the elements of the architecture like processes and devices and their instances, resources, channels, states and commands are implicitly declared by literals.

- A file including general options and rules for automated generation of the system.
- A C include file containing compilation switches which define the functionality and the instrumentation to be included.

Having executed such steps the system is immediately executable by the simulator or on the target system and a first feedback can be obtained.

Depending on the complexity of the system it typically takes about 10 to 20 minutes (including compilation, linking and execution) to generate everything from scratch what is needed for execution. Graphical figures of data flow (like MSC's), propagation of data over time (timing diagram) and reports (coverage, exceptions, injected errors) are provided after execution. On request the generator inserts automatically actions for fault injection.

Then, the user can incrementally add functionality in each of the processes. Per action he can call user-defined functions. If such functions are missing during the first iterations the system generator provides means to emulate the missing logic.

After each extension or modification the system may be subject of validation either by the simulator or on the target system.

Validation support is provided by SDL tools like ObjectGEODE, and by EaSySim II. In latter case EaSySim II adds own functionality to cover the validation needs.

Also, EaSySim II provides library routines by which the source code can be instrumented for resource consumption and performance analysis, generation of Message Sequence Charts (MSC's) and timing diagrams.

Moreover, automated testing as described in 2.4 above is avaliable and will be applied to the validation of MSL [15].

4. BENEFITS

The benefits observed so far are:

- A first iteration of an executable for a new application can be obtained just during a coffee break due to tool support which builts automatically the specific system from a generic structure and engineering inputs.

  The system is immediately executable on platforms such as UNIX, Linux or VxWorks.

- The organisation principles, the code generator and the large number of library routines can be reused from project to project.
- Parts of an application can incrementally be refined or modified with little impact on other modules.
- Validation can be performed after each such incremental step in the simulation and the target environment.

4.1 Incremental Development

The initial system is built automatically based on information about commands, devices, resources and existing software which may be added.

A user has to define the commands, the devices and the resources in terms of literals. This information will be inserted into source code templates (SDL/C or pure C) from which the initial version is derived.

Also, a user has to define by data the topology, the groups of redundant channels, properties of the communication devices, other performance data and so on. This is a standard procedure to be applied for every project. For definition of the

topology simple mapping rules may be given, even wild cards are allowed.

By a next step a user expands his application by adding new commands. There is no need to define all commands right at the beginning, but they may be added as needed. Also, more communication channels may be added when needed. The library routines just process what is defined and they accept every change of the number of channels or of channel properties. Most of the properties can also be changed during execution, if needed (this is usally called "on-line maintenance").

After each upgrade the software may be validated by simulation or on the target system. This way a system can easily be expanded towards its final version due to automatically generated new versions.

As data transfer is transparently handled by a routing layer, components (processes) can easily be migrated (off-line or on-line) between processors. This allows to built transparently distributed systems for which functionality may be allocated according to actual performance needs (if allocation is not constrained by the hardware configuration).

4.2 Reuse

Reuse is multiply achieved: there is (1) reuse of a system's software structure, (2) reuse of the organisation of developement (the software development process), (3) reuse of the library, and (4) reuse of user-provided (existing) software which may be attached at well-defined interfaces.

The provided library routines for data communication are generic and cover a number of transfer media like IPC, TCP/IP, RS232, ISDN, modem, Tcl/Tk screens and a protocol for safe data transfer.

Support is available to store or retrieve data into/from buffers (also circular buffers are supported), to generate and evaluate statistics, and to handle status messages.

Also, communication with an external (SQL) database in a safe manner (regarding loss of data) or with graphical user interfaces (GUI) is already supported.

Hence, when starting with a new project a user has already available a large number of well-tested routines. Moreover, due to the standardised data exchange format, the generic structure and provided organisation scheme the user will be able to generate himself more reusable routines for his application domain.

4.3 Integration Platform

Due to the standard format for data exchange and standardisation of time management (not described here) an integration of asnychronous with synchronous software like code generated by the tool SCADE [16] was possible within a rather short time during the ESPRIT project CRISYS [17].

5. NEED FOR CUSTOMISED TOOLS

Most tools force a user to define dedicated and hardcoded lines to express the data flow between two components: one line per command (or a subset of commands) is required instead of one line associated with a set of commands like it is needed in case of a standardised interface.

With current tools, the visual representation of the functionality as expressed in the first case mentionned above is better, but the degree of reuse and the productivity are significantly decreased. As the commands may change from project to project such lines need to be redrawn for each project.

In the second case the documentation of the system's architecture is not well supported by current tools, but development is more efficient.

Below some problems are dicussed which are valid in general, but have been identified when a generic approach was applied.

5.1 Problems with Topology and Data Flow

To conclude, a generic approach is not well supported a priori by a tool: the tool assumes that the data flow is defined regarding documentation purposes rather than efficiency of development and reuse. It was observed that engineers intend to express an architecture in view of readabilty of the tool-generated figures rather than to tune the efficiency of the implementation process.

Also, the coherent transition from an early to the final development phase by incremental refinement is not well supported. Most of the current tools are based on development methods which do not assume such coherent transitions.

When introducing rules which shall ensure higher degree of reuse (during the implementation phase) this is in conflict with traditional development approaches where a specification will never be used for later implemenation: it just serves to express the requirements and this makes it reasonable to adjust the structure and the architecture according to documentation needs.

When taking a generic approach which shall apply to all development phases this causes problems because the capabilities for expressing functionality or tracing of data flow are not appropriate.
Usually, a dedicated interface line is introduced according to the command to be documented. Hence, the communication lines are chosen according to documentation needs.

However, the generic data format as described above covers possibly a large number of commands which usually are not (cannot be) displayed by a tool along a line.

Moreover, in case of the standardised format it may happen at execution time that a number of data are displayed by the the tracking tool which are not of interest for the user. But if the tool is not supporting formating / filtering of the information, such data cannot be suppressed and this makes it difficult to read the graphical diagrams.

Another problem relates to documentation of data transfer through physical channels which are not inherently supported by tools. TCP/IP or RS232 channels are examples in case of SDL. SDL itself only supports one type of a physical channel for communication.

When getting data from different channels (like shown by Fig. 2-2) (after extension of the tool) it would be rather helpful to document through which channel the data arrived or were sent. This information is not needed if only the SDL channels are available. But it is missing if other channels are introduced by a user.

Such problems related to an incremental, generic and innovative development approach require some tool extensions by add-on software guiding the (commercial) tool to provide what is needed.

5.2 Limitations by the Class Concept

Automated reuse of classes from project to project is not possible in most cases due to structural differences between the project-specific software. Manual instrumentation is needed to implement such structural changes.
This limits the advantage of the class concept significantly regarding automation and efficiency of software development. Hence, automated generation can only be applied to some parts of the system, but not to the whole system.

5.3 Possible Solutions

5.3.1 Tracking of the Data Flow

The most efficient solution is to (re)use a (commercial) tool which is already available and to upgrade and extend it in a manner which ensures compatibility with its future versions.

As the SDL verification means are based on description of data flow by Message Sequence Charts (MSC), routines have been established which generate output compliant with the MSC standard [18] from pure C. Hence, the output can be processed by SDL toolsets [8,9] and the SDL verification capabilities are available for a pure C environment.

EaSySim II instruments the automatically generated code, but a user may instrument his application code (SDL and/or C) by the EaSySim II routines in addition.

The EaSySim II routines allow to format and to filter the displayed information, and to display the name of the transfer channel as discussed above.

This instrumentation is not only limited to SDL, but may be applied to every source code written in a programming language supporting a C interface.

By adding such interface routines the tracing also is extended towards timing and performance: a separate file is generated including the timing information. From this file timing diagrams are derived which allow to display the MSC information in a format known from logic analyzers or performance simulation tools like SES/workbench [19] (Fig. 5-1). For debugging an event may be selected to display the contents of the included data message (Fig. 5-2).

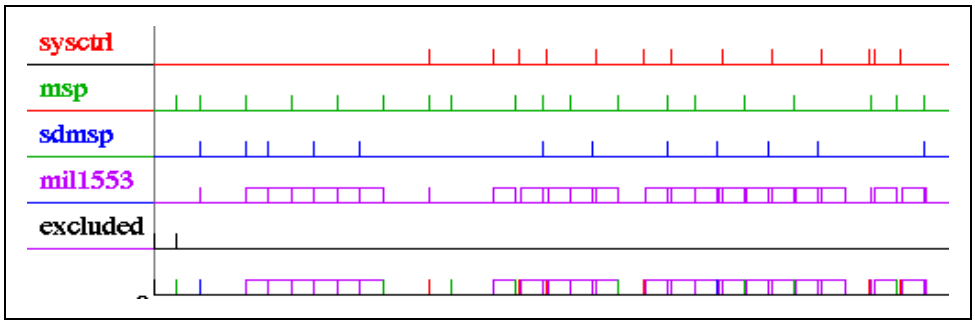


Fig. 5-1: Feedback by a Timing Diagram

StartTime: 0.802 sec
FrameDuration: 98.000 ms
Medium: SDLch
Device: bus1553
Source: bus1553
SourceInst: 1
Dest: sdmsp
DestInst: 1
DestSite: 0 Priority: 255
DataStatus: short
Command: msdae_txcmd
Inst: 1 Length: 0
Index: 1 Data: 0

Fig. 5-2: Contents of a Data Message

5.3.2 Extending the Class Concept

The class concept was replaced by a concept for construction of an instance from a template. In case no structural differences exist the construction rules are based on "copy/paste" and "replace".

When a new structure needs to be built for an instance the given rules can automatically add e.g. branches to an instance or data declarations.

This way all software can be generated automatically and no manual intervention is needed for instantiation.

6. CONCLUSIONS

An approach to standardise and formalise the system and software development process has been described which supports incremental development and validation and allows fully automation of a system. The benefits for software development and the observed problems with existing tools have been discussed. The approach has been and will be applied to a number of projects and refined continuously based on the obtained feedback.

The organisation of developement and of the software itself allow to automate the generation process. Only a minimum of high-level engineering nformation is needed. The type of information to be provided relates to the system itself, detailed knowledge on software engineering and software implemetation is not necessarily required.

This opens the possibility that a system engineer and not a software engineer can perform the early activities during system development because the engineer just has to provide information about the system: the processes, resources like processors, buses and network and their performance, commands defining the data exchange and data processing.

The formalisation of the inputs allow for automated testing and fault injection.

Due to the support of automated testing a system engineer can perform a first validation of the system. Then a software engineer may continue with details by applying an incremental development approach.

This approach does not require a number of methodological (graphical) diagrams due to automation. Such diagrams are ususally needed to simplify the manual development steps. The absence of such diagrams is one reason why costs are decreased because manual processing requires a lot of human resources.

The goal to allow a system engineer to experiment with rapidly available and inexpensive system configurations at the very beginning of a project was already addressed by the HRDMS [2] and OMBSIM [3] projects but it took a lot more time to end up with a solution: the feedback from a number of exercises was needed to get an

appropriate organisation and a generic system architecture for automated construction of software.

This work was funded in part by the ESPRIT project CRISYS [17].

Team members are: Schneider/SES (F, prime), Elf (F), Siemens Electrocom (D), Verilog (F), Verimag (F), Prover Technology (S), GMD (D), University of Grenoble (F), CEA (F), BSSE (D)

7. REFERENCES

[1] Specification and Description Language, Z.100 (03/93), ITU General Secretariat - Sales Section, Place de Nations, CH-1211 Geneva 20

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

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

[3b] OMBSIM (On-Board Mangement System Behavioural Simulation), ESTEC contract no. 10430/93/NL/FM(SC), Final Report Nov. 1995, Noordwijk, The Netherlands

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

[5] EaSySim II, 1996-1999, Rainer Gerlich BSSE System and Software Engineering, Auf dem Ruhbuehl 181, D-88090 Immenstaad, Germany

[6] Protocol Validation, BSSE, 1997, unpublished

[7] CADIS (Central and Remote Data Acquisition and Distribution Integrated System), 1997-1999, Rainer Gerlich BSSE System and Software Engineering, Auf dem Ruhbuehl 181, D-88090 Immenstaad, Germany

[8] ObjectGEODE SDL-Tool, Verilog, 150 rue Vauquelin, F-31081 Toulouse Cedex, France

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

[10] Packet Utilisation Standard (PUS), ESA PSS-07-101

[11] Rod Allen, "The SoftBus Approach", ESA/ESTEC, Noordwijk, private communication, 1986

[12a] AESOD I, Study on AOCS Embedded Software Design, ESTEC contract no. 6534/85/NL/AN(SC), Final Report, 1987, Noordwijk, The Netherlands

[12b] AESOD II, ESTEC contract no. 7433/87/NL/AN(SC), Final Report, 1990, Noordwijk, The Netherlands

[13] Common Object Request Broker Architecture (CORBA), Object Management Group (OMG)

[14] Executable Specifications with particular application to spacecraft control and data systems, 2nd Round Table Proceedings, December 10-11, 1996, ESTEC, Noordwijk, The Netherlands

[15] A SDL Model for Behavioural Validation of MSL, ESTEC contract no. 13309/98/NL/MV, Noordwijk, The Netherlands

[16] SCADE, Verilog, 150 rue Vauquelin, F-31081 Toulouse Cedex, France

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

[18] Message Sequence Chart (MSC), Z.120 (10/96), ITU General Secretariat - Sales Section, Place de Nations, CH-1211 Geneva 20

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