Development of SDL-Based Software for an Embedded System – Practical Experiences

Total 13 pages

S.Bläsius, J. Maier, G. Kohler, S.Karg (now with VIAG Interkom)
Tenovis GmbH & Co KG, Kapellenstr. 13 , D-82008 Unterhaching, Germany
Tel.: +49 89 66 53 10, Fax: +49 89 66 53 12 40,
{Stefan.Blaesius, Josef.Maier, Guenther.Kohler}@ tenovis.com
Stefan.Karg@viaginterkom.de

ABSTRACT

Requirements addressed at software development for telecommunication systems are highly demanding. On the one hand, new products in this sector are to be presented at the market in a minimum of time. On the other hand, a high quality standard is to be ensured, thus combined with an ever increasing amount of services and features, need to be supported.
The software development for embedded systems normally consists of two steps. First, software is developed on the host system. This software is than tested under non-real-time conditions in a simulated environment. After that the software is integrated in the target system, where it is exposed to all influences of the "real-time-world".
Hence, a test concept is needed, that both fulfils the requirements of software development and supports testing at host and target.

This paper provides an overview of the SDT-based development process for PBX-systems at Tenovis GmbH, with a special focus on the test concept. Moreover, it reports the practical experiences that have been made by using the SDT-tool chain from design phase up to software integration at the target.

KEYWORDS

SDL, MSC, industrial application, development process, testing, embedded systems

1 INTRODUCTION

As we all know high product quality together with time-to-market are requirements, that are addressed to software development for telecommunication services. This demand is difficult to satisfy. Taking that into account, using SDL-based tools allow a structured design of even complex software architectures. Tenovis GmbH (former product division

Private Networks of Bosch Telecom GmbH) is developing software for private branch exchanges (PBX) supporting ISDN- and DECT-standards.
A test concept based on the SDT has been designed to fulfil the following requirements:

- achieve high quality at host test level
- achieve a close relationship between host test and target test, i.e. the results drawn from the host test can be directly transferred to the target test or vice versa.

The use of SDL-based tools for the development of telecommunication systems is already recommended by the international standardisation (ETSI, ITU, CCITT). At Tenovis GmbH we have collected wide experience in the Specification and Description Language SDL by using a self-developed SDL-Editor for over 10 years.
Section 2 describes this first approach of software development using SDL and states reasons for the migration to the SDT-tools. Furthermore, it provides an overview on the target system (PBX) and its hardware restrictions. In section 3, the focus is on the test concept on the host system. Additional tools are described which improve the correlation of host and target test. While Section 4 deals with testing the system on target, section 5 compares our experiences with SDT and our self-made SDL-tool.

2 ACTUAL SITUATION

The target platform is a multiprocessor system based on the INTEL-microprocessor 80186/188 as shown in Fig. 1. Either units contain a microprocessor that communicate with each other by using a LAN-alike interface. Up to 6 non-intelligent units access telephony interfaces such as digital and analogue lines which are supported by one intelligent unit. A real-time operating system (RTOS) guarantees basic communication functions for the application software located at each processor unit.

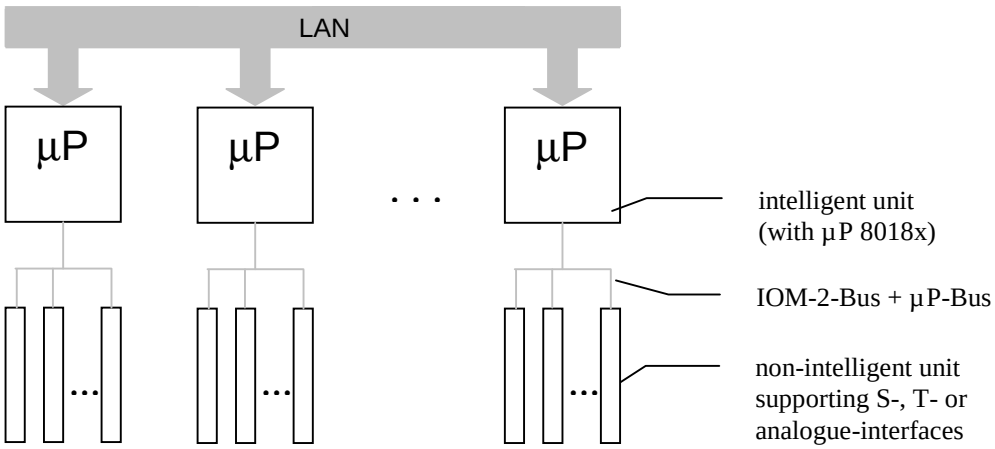


Fig. 1) The target platform of the PBX

Tasks that have been developed using SDL – for example -are ISDN-layer 3 or DECT-Mobility Control or a task providing CTI-functions.
Due to the architecture of the processor in use, the following restrictions on RAM and

ROM sizes apply: A task must not exceed 1MB of ROM and 512 KB of RAM although sophisticated measures are taken to increase the addressing capabilities of the microprocessor.

The PBX-software is developed on the host system, a SUN-workstation. The first approaches using SDL were conducted with a tool developed in-house, which supported graphical editing of SDL-88. This tool - called SDLDG - was capable to store SDL-Graphic-Representation (SDL-GR) in a non-readable private file format as well as to produce standard textual Phrase Presentation (SDL-PR). In addition, a code generator was developed, that automatically generate C-code by combining the PR-format of the specified process with a so called list file. The latter had to be provided manually and was used to map the symbols applied in the SDL-process to C-functions. Compared to traditional development (C/Pascal) we soon realised the significant advantages of generating SDL-based code e.g. a measurably quicker development and a considerable lower rate of errors.

However, limitations of this SDL-environment were recognised with both the process communication and the handling of SDL-instances. One of the drawback was the private SDLDG-tool did not support the design of inter-process communication or instantiation. It had to be implemented manually in C-code and – as we realised – was very inflexible.

Given this fact in 1997 we decided to use Telelogics SDT for the development of any future SDL-systems. This SDT provides the design of the process communication, the part of the SDL-design we have missed with our self-developed tool. Furthermore a powerful tool chain has been offered to support the entire process from the design to the integration tests at the target. The big challenge was now to integrate the new SDL-code at the restricted target system and to fulfil the system requirements such as:

- high quality and reliability
- high performance
- use of automatic documentation
- support of error analysis

The next section describes, using these new tools, with a focus on quality assurance and testing.

3 SOFTWARE DEVELOPMENT AND QUALITY ENSURANCE USING SDL & MSC

Fig. 2 shows a survey of the methods and tools used in the process of developing software for the PBX-System.
Documentation as well as test- and coverage reports have to be exact, detailed and up-to-date. We are firmly convinced that the use of SDL and the corresponding chain of tools has helped a great deal to speed up development and to reduce the efforts needed for documentation purposes.

3.1 THE DEVELOPMENT PROCESS

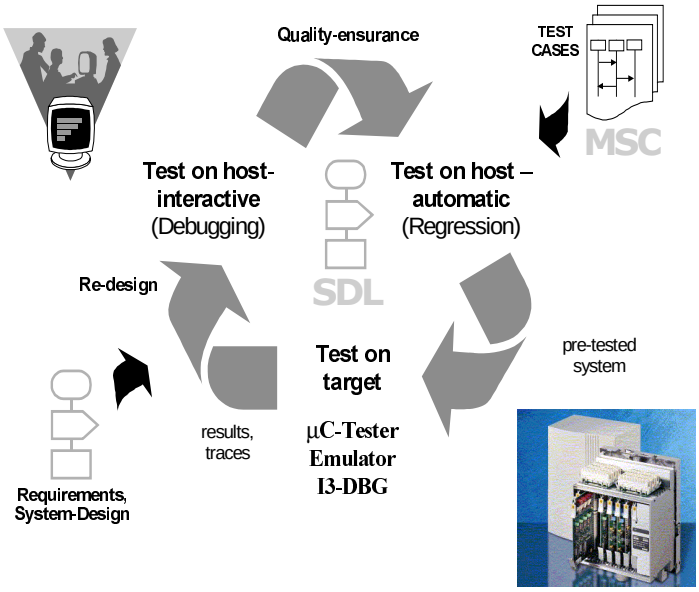


Fig. 2) The development-process

After a thorough requirements analysis and a first draft of the systems' design – done in discussions at the drawing-board, - the functional design is performed by using SDL.
A top-down strategy is used here: first, the functional blocks, their tasks and the communication channels are defined. Then each developer on the team is assigned a block and is to design the internal structure. As to keep the system consistent, reviews concerning the design are called in before the actual implementation is actually begun.
Each developer is given the ability to simulate the system and to test single components and functionality interactively by using our testing set-up. To simulate the system advantage is taken from the SDL-Runtime-Simulation library C-basic integrated in the SDT thus providing ample support for testing and debugging. The simulator had been extended with our testing environment (called TUSSI, an acronym derived from the German equivalent of 'testing environment for SDL Systems, interactive'). TUSSI simulates the basic functions of the target system's RTOS, such as memory-management and task-communication. It also provides a capable user-interface for sending messages into the simulated system.
The development of functional entities of a complex system can be easily paralleled by this approach. Since all components can be tested individually, the effort for testing the integrated system is drastically reduced. To detect flaws in process-interface specifications at the earliest possible stage, the integration of TUSSI is as early as possible.

3.2 SPECIFYING TEST-CASES BY USING MSC

Each developer has to provide test-specifications for his/her area of responsibility – signifying how and what to test. Those documents have to be provided as MSC, since they will directly be used as sources for regression testing.

Fig. 3 shows an example of a test-case:
The IUT in the diagram is tested against two tasks named ‘L2’ and ‘L3’. The specified signals are automatically converted into the testing inputs.

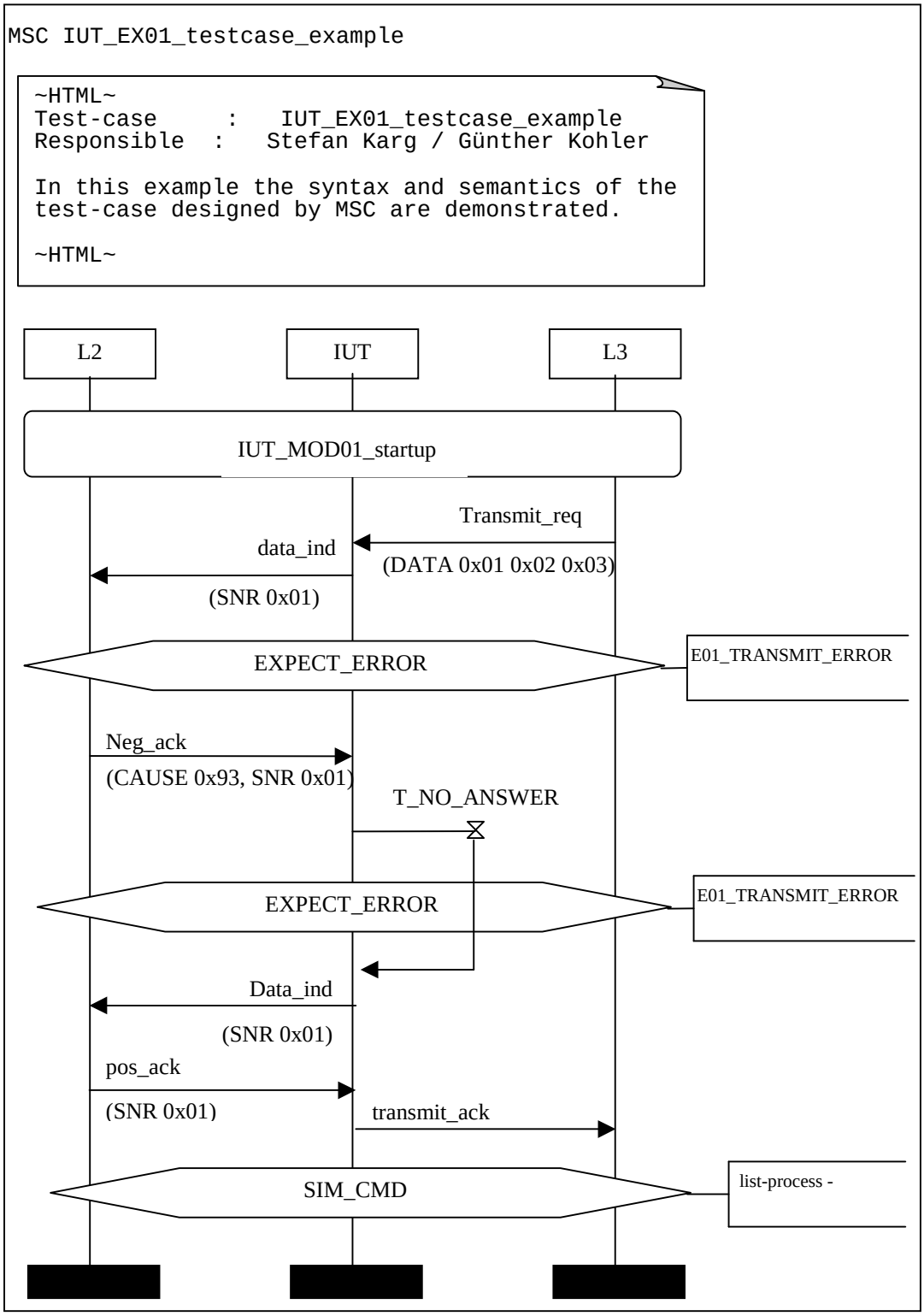


Fig. 3) Testcase definition using MSC

However, there's one part that hasn't been automated yet: the contents of the individual messages to be sent into the system have to be provided manually. These will be used for both, debugging and regression testing, ensuring that there are exactly the same preconditions for both methods. To reduce the number of message files to write, variable information elements, such as party numbers or call reference can be coped with. Therefore only a description of every different message type has to be provided.
The variable data source is provided as parameters of the signal, thus specifying the values and which element to modify.

If needed, reoccurring situations can be included by using the MSC-Symbol 'reference', whilst the symbol 'condition' is used to provide information for special situations, e.g. specifying an 'Expected Error' or sending commands to the simulator.
An 'expected error' is used to test the behavior of the system in case of failure. On execution the signal specified immediately after the 'expected error' condition has to lead exactly to the error specified by it. If another error occurs or no exception at all is thrown, at least the test-case will be marked "passed with warnings".

Another item of interest is the automated documentation of the test-case's purpose. The part of the test-case's description given in the text-field - embraced in the tags '~HTML~' - is extracted automatically and is stored in a HTML-Page for documentation purposes.
Most of the tools converting the MSC-specifications into inputs for testing and the tools analyzing the results have been written in PERL, thus taking advantage of the powerful text-analyzing and -extracting methods underlying this language.

3.3 REGRESSION TESTING

Fig. 4 shows our strategy which is to begin with regression testing as soon as one function is ready for testing. The benefit of this concept lies in the fact that one can ensure a level of functionality once reached is kept throughout the whole development
process.
By invoking a regression test, the system is tested against all test-cases.
For this purpose, a tool converts the test-cases, specified by MSCs, into input-files suitable for the testing environment and a description of the expected system behavior. Those are then run against the Implementation Under Test (IUT).

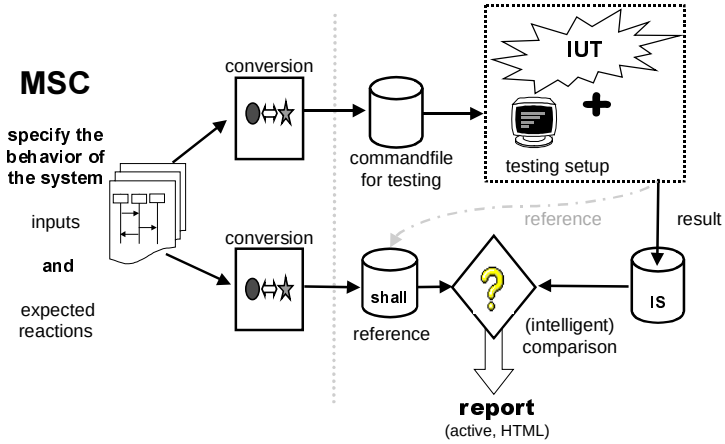


Fig. 4) Regression testing set-up

After completion, the system output is compared automatically with the specified reactions. This process is conveniently automated taking advantage of the UNIX-make mechanism.

From a test results an active report in HTML is generated automatically and distributed by mail to the development team members. This report contains information about test coverage, a statistical survey of the test-run's results, recently changed modules and a table of individual results. The outcome of a test is referred to in three categories: "passed", "passed with warnings", or "failed". Since a black-box strategy for testing is used, a test has failed when the system's behavior on the external interfaces has changed. This is determined by checking the test-cases' output against a reference. A developer viewing the report, can click on the verdict and is shown the differences between reference and individual result in the Web-browser.
If the external behavior is correct, there can nevertheless be some implementation flaws or internal errors, e.g. lost signals, or handled exceptions. These can be detected by analyzing the log-file generated by the simulator. These exceptions and implicit signal consumption are indicated by characteristic strings in the simulators output. Recognize them lead to a verdict of "passed with warnings", if the exception was not specified in the test case description beforehand (this is essential to distinctly test the occurrence of an exception in a certain situation). By clicking the verdict the recognized internal errors in the log-file is shown.

3.4 TAKING IT ONE STEP FURTHER: FEEDBACK FROM TARGET TESTING

When the implementation has reached a stage, where a test on the target platform is possible, a first attempt of integration is performed.
To further increase the possibilities for debugging, we have developed a means of catching the messages received by the system on target and to convert them to inputs for the host-testing system. This is executed by tracing the inter-process communication on the internal LAN of the PABX. By using these messages it is possible to simulate the system behavior on the host with exactly these messages received on target. This process has the advantage of getting reproducible situations in a highly capable debugging environment – without having to administrate the code on target. Furthermore, recordings of a detected error on target can be converted directly into test-cases for regression testing on the host. By doing this, the coverage is increased and with it, the meaningfulness of the regression test in a constant fashion while testing.

4 ON TARGET TESTING WITH THE MICROTESTER

A highly capable debugging and testing environment on the host side is used. However what is the situation on target, where testing and debugging under real-time conditions is a necessity?
As the severe memory restrictions and real-time conditions, the C-basic or C-advanced code generator cannot be used for target implementations. Not to loose the benefit from automatically generated code derived from the SDL descriptions, we decided to use the C-micro code generator from Telelogic's tool chain. This code generator is highly scaleable and copes much better with our memory restrictions. However, this advantage

is gained at the cost of some language restrictions to SDL-96, e.g. one must not use most object oriented structures.

We decided to use the 'light integration' approach of the C-micro library (refer to Fig. 5). Thus being the easiest approach for getting a SDL-based system integrated with a legacy RTOS environment:

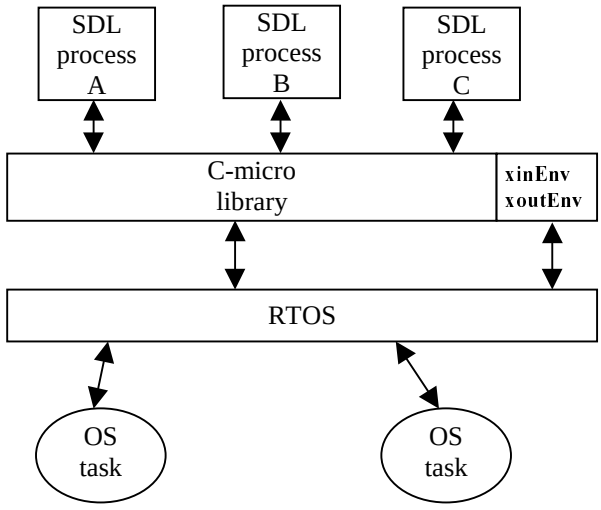


Fig. 5) „Light integration" of SDL-Systems into the RTOS

In this set-up, the complete SDL system is running as one OS process. It uses the SDL mechanism for SDL internal communication and the OS mechanism for communication with the environment, so do the tasks, programmed in a conventional way. This approach also implies that the C-micro library is a part of every OS-task realized in SDL. If there is more than one task using SDL, redundancy of the library code is the trade-off for an easy integration. Due to this fact, this solution is only practicable if the minority of tasks on one board are realized using SDL.

Another feature we use successfully is the optional available so called "microtester", allowing convenient tracing and debugging at SDL level in the target but at the loss of memory. The tool is highly scaleable and allows to turn single features on and off at will by conditional compilation.

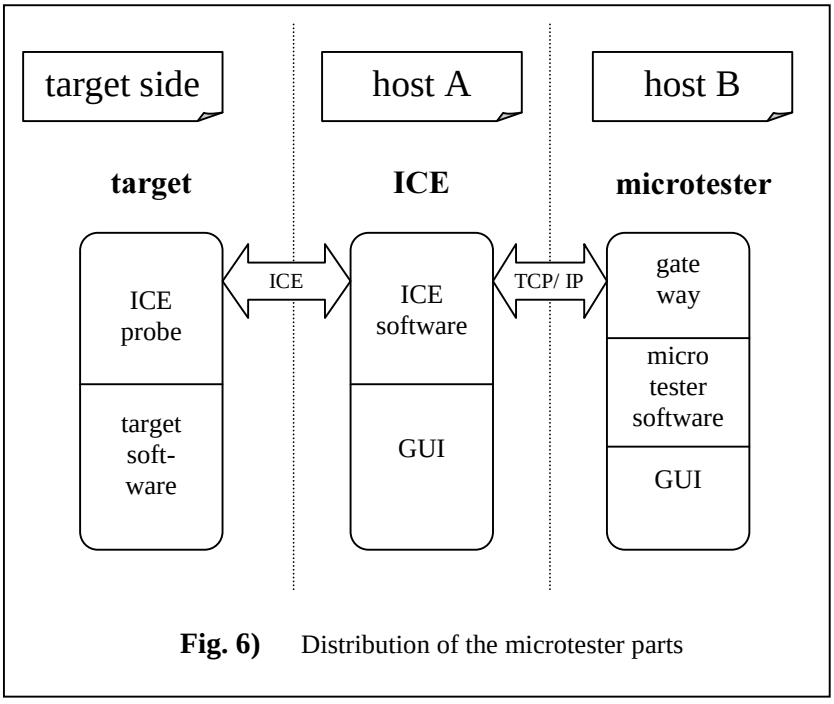


Fig. 6) Distribution of the microtester parts

The microtester operates in distributed mode (Fig. 6): Some parts are compiled into the target software, while others run on the host machine. Functions which allow control and usage of the data link, are compulsory on the target implementation. Additionally

procedures are required to retrieve the internal information on system states, variable values, trace information and the like. The host is connected via data link - either a V.24 or TCP/IP or the “Lauterbach Gateway” - to decode and present the extracted information.

The In-Circuit-Emulator (ICE) software and the microtester host software may run on different host machines, dependent on the socket connections used. The “Lauterbach - Gateway”, a connection of the SDT microtester to the ICE of Lauterbach Datentechnik GmbH, uses the ICE’s functionality as a communications link between the microtester parts on the host and target. Within the ICE the dual ported RAM is used for information interchange between both parties, ICE and the gateway client, reside on the host part of the microtester and are connected by a socket link.
Using this special gateway has the advantage of reducing development efforts and testing side effects. Since the gateway is provided by Telelogic no extra communication mechanism has to be implemented. Furthermore no additional interface hardware is necessary on the PBX-boards. In addition, the implications on the real-time behavior are reduced by using the emulator-probe as communication link.
By using the microtester it is ensured to obtain a detailed textual trace of the system running in ‘real-world-conditions’. Thus including all information provided by the SDL-System, such as states reached, signals sent and received or decision paths chosen to mention only a few.

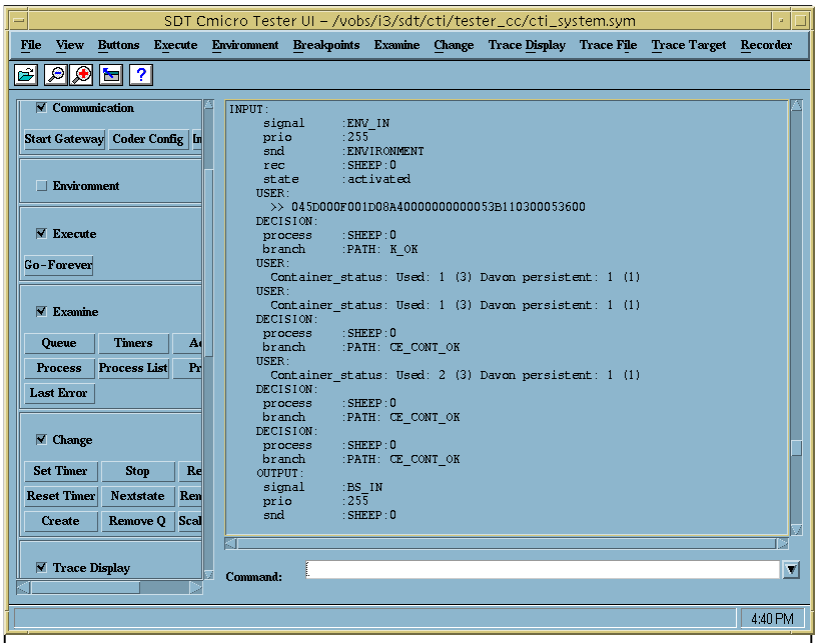


Fig. 7) GUI of the microtester (Screenshot)

Fig. 7 shows the Graphical User Interface (GUI) of the microtester. The textual trace in the right part of the window can be recognized together with the User Interface as well as the controls for controlling and enquiring the system running on target.
Moreover, the microtester allows features for debugging usually only available in a development environment, e.g. setting breakpoints at SDL-level.

One of the most useful features is the possibility to draw MSC-diagrams of the system in target; thus being a great help in debugging and verifying the dynamic behavior of the system. It is possible to check the expected behavior of test-cases

on the host against the real situation measured on target using the same methods and tools.

All available features cannot be used, due to the restrictions embedded systems impose on ROM and RAM. For example a graphical SDL trace - though available in the microtester - can not be used, because the additional amount of code needed to provide the trace information would create a memory overflow.
The microtester also offers a record and play mode. Here SDL signals are written in a file and can be injected to the target on demand. With this feature it is possible to reproduce testing scenarios. However, this 'nice feature' cannot be used on our telecommunication target because firstly of the multiprocessor system and secondly as not all tasks are designed with SDT. Therefor recordings are always incomplete as not all events occurring on target are visible to the microtester.

5 DEVELOPMENT EXPERIENCES USING SDL

This section focuses on anticipated problems and memory consumption when a SDT-environment is employed.

5.1 TARGETING ASPECTS

As we use different code generators and kernels on the host and target, some problems are caused by the different approaches. One obstacle is that the scheduling strategy differs between the C-micro kernel applied on the target and the C-basic kernel used on the host . In the C-micro kernel exists a signal queue, which stores the signals in incoming sequence. No difference is made between internal signals and those from the environment. On the contrary, the C-basic-kernel on the host provides one queue for each process with the option of setting process priorities.
It is necessary for the communication within the target that the order of incoming signals from any other tasks is adhered to. This means that a signal from the environment has to be fully processed before accepting the next, a behavior known as "run to completion". To achieve this, the original task-loop of the C-micro-kernel has to be changed by setting priority of internal signals over environmental signals. Only than it is guaranteed that an external signal is only processed in the system idle state, i.e. all internal signals (except saved signals) are handled. On the host this behavior is forced by using only process priorities without changing the task-loop of the kernel.

Furthermore different of data structures on target and host are faced. The processors on host (SUN SPARC) and on target (INTEL 80186) use different byte alignment and byte order. This means, that messages received from other tasks, which are represented by byte-streams in our system, could not be used for testing in the host simulation. In addition, byte-streams generated from our system could not be verified against the target requirements. This lead to the development of a central module - called container - for translating byte-streams to SDT-internal data structures and vice versa. The byte-stream is completely hidden from the actual implementation, which proved to be highly useful. With that the messages for host testing

are exactly identical to the messages on target.

Another aspect is, that the internal data structures of C-micro code and the C-basic code are completely different. Normally this creates no drawback, as these internal structures are hidden by SDL. However, problems can occur if such data structures (e.g. process identification PID) are used in C procedures outside SDL. In this case conditional compiling must be used.

5.2 MEMORY USAGE AND TIMING IMPACTS

A well known problem in testing is that using test equipment can change the behavior of the tested object. This also occurs when the microtester is used. The first critical element is the increased software. Hence the memory layout (mapping) is different. This may have effects on debugging, because some mapping dependent errors (e.g. overwriting data by false pointers) cannot be observed when a different layout is applied.
Fig. 8 shows the SDL related memory usage of our current SDT project. In both diagrams 100% always refer to the maximum memory usage.

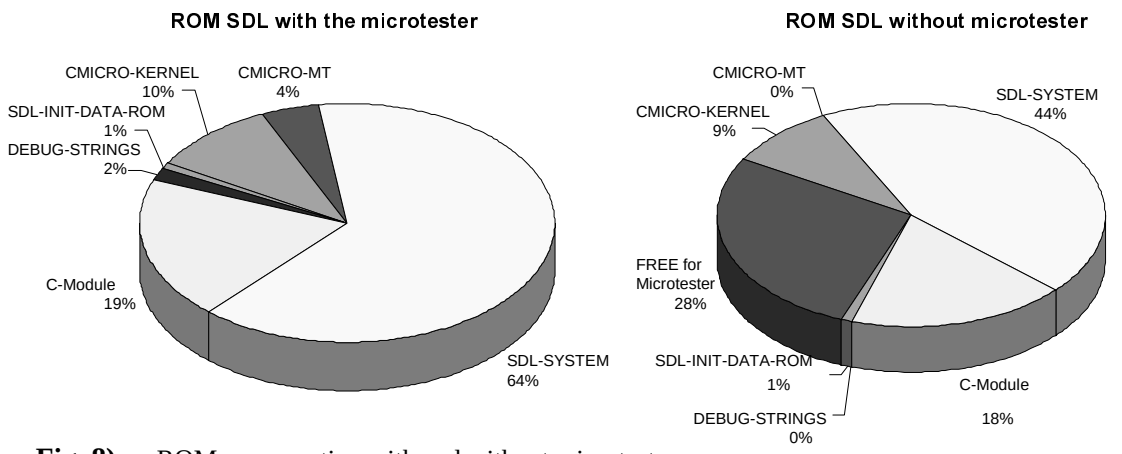


Fig. 8) ROM consumption with and without microtester

In general, the ROM needs roughly 30% for the microtester usage. The microtester library is relatively small; therefore it has no great influence on the ROM. The influence on the code generated from the SDL system is more significant due to the additional instrumentation of the code, for example trace procedure calls. The relatively small size of the debug-string segment can only be achieved by post-processing the generated code. Whilst the C-micro code generator produces a string constant for each occurrence, hereby wasting lots of memory, a tool substitutes reoccurring strings addressed by references.

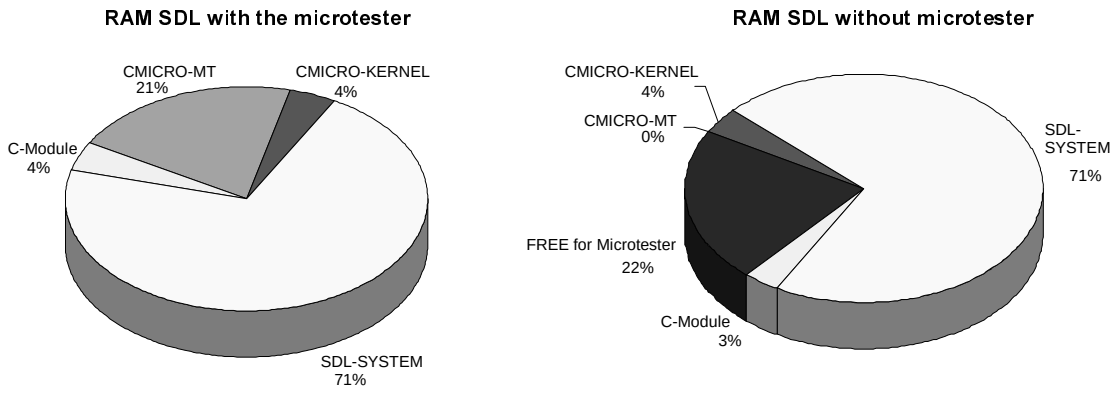
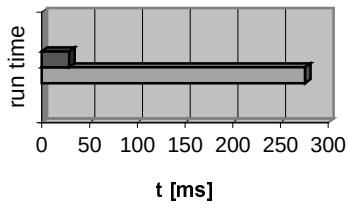


Fig. 9) RAM consumption with and without microtester

In contrast to ROM consumption, the generated code uses no additional RAM. But the microtester needs a huge buffer for communication via the gateway with the host: Fig. 9 shows that this fact leads to a 21% increase in the RAM.
As to ensure work with the microtester on the target later the implications of applying the microtester on memory usage have to be considered up front when planning a SDT-based system.

Another interesting point is the testers' influence on the execution time. When testing with the microtester is embedded, each transaction takes longer. That may cause problems if the target has strict real-time requirements e.g. there may be timeouts or synchronization failures.



	run time
without microtester	28,810
with microtester (best case)	275,980

Fig. 10) Influence of the microtester on run time

The timing data in Fig. 10 is taken form our current SDT project according to the ICE's timing analyzer. As our implementation consists of SDL and non-SDL parts, the major non-SDL part (ASN.1 coding) is listed separately. The timing data given for SDL includes the complete running time; i.e. starting with receiving the signal to all internal

signals are processed and the system is ready to fetch another signal from the environment. As a real-time operating system is used, times measured correspond with the actual processing times. That means the time measured consists of the execution time of our task plus the run time added by task interruptions.
The size of the microtesters communication buffer has an immense influence on the run time. If the buffer is designed to small a frequent buffer overflow will block the task until the buffer is cleared by the host. Thus real-time is completely lost.

5.3 THE RIGHT TOOL FOR THE RIGHT JOB

On the whole, the first SDL project set up required much time for the basic design and the adaptations to the C-micro library. However, if these initial problems are solved, the SDT tool chain is most effective with SDL based software development.
With any following project the prior project can be used as a template, with only minimal adaptations to be made. Furthermore, functionality and bug-fixing of existing SDT projects is much easier compared to the conventional approach.
Although the main part of our project is developed in SDL, it is not advisable to try and implement everything in SDL. While SDL is ideal for protocol handling and state machines, software parts with loops or small data bases should be implemented in conventional C code. An ADT mechanism provides the connection between C code and SDL.

6 CONCLUSION

By employing the SDT tool chain together with our adaptations and extensions, we found a highly capable environment which facilitated and accelerated the development process.
Albeit, using the strategies and methods described in this paper enables the user to conduct highly paralleled software development for PBX systems; together with the possibility of integration and regression testing throughout nearly the whole process. This together with automated documentation, testing and report generation has led to a measurable increase in development speed and product quality.
Due to restrictions on our target system, drawbacks are that the SDL-Library used on the host and target is not inline. This may cause differences in the behavior (e.g. scheduling). Also the use of the testing tools has impact on real-time behavior and memory requirements. Finally our experience suggests, that those impacts are minor squibbles and can be coped with.

REFERENCES

[1] Telelogic: “TAU 3.6 Reference Manual”, Telelogic AB, Malmö, Sweden, 1999
[2] Bosch Telecom GmbH: “I3 Technisches Handbuch”, Bosch Telecom1996
[3] ITU-T Recommendation Z.100: “Specification and Description Language (SDL)”, ITU-T, Geneve 1993
[4] ITU-T Recommendation Z.120: “Message Sequence Chart (MSC)”, ITU-T, Geneve 1993