An Experiment with the LOTOS Formal Description Technique on the Flight Warning Computer of Airbus 330/340 Aircrafts

Hubert GARAVEL
INRIA Rhône-Alpes / VERIMAG*
Miniparc-ZIRST
rue Lavoisier
38330 MONTBONNOT ST MARTIN
FRANCE
E-mail: `hubert.garavel@imag.fr`

René-Pierre HAUTBOIS
AEROSPATIALE
Avionics and Systems Division
A/DL/EP M8621
316, route de Bayonne
31060 TOULOUSE cedex 03
FRANCE

Abstract

This paper presents the main results of a two-year study concerning the introduction of formal methods in the life cycle of avionics software. This study was done in the framework of the EUREKA European project AIMS (Aerospace Intelligent Management and development environment for embedded Systems).

The ISO language LOTOS was used to describe a significant subset of the Flight Warning Computer of Airbus 330/340 aircrafts, which is a typical representative of Embedded Computer Systems. Six LOTOS descriptions were developed, (using both the abstract data types and the process algebra features of LOTOS) which are probably among the largest algebraic specifications written today. The CÆSAR/ALDÉBARAN toolset for LOTOS was used to support the description and analysis process. The LOTOS descriptions were automatically translated into executable prototypes, and then validated by means of simulation and testing. The paper presents the techniques used and the results obtained. It ends with a critical evaluation of the ISO language LOTOS as far as other applications with similar characteristics are concerned.

Key-words: Abstract Data Types, Aerospace, Airbus, Algebraic Specifications, Avionics, Design, Embedded Computing Systems, Formal Description Techniques, Formal Methods, LOTOS, Process Algebras, Real-Time, Simulation, Software Engineering, Specification, Testing, Validation, Verification.

Introduction

Aircraft, spacecraft, and helicopters contain increasingly complex, high integrity systems that oversee flight control, avionic and cockpit systems. The current trend is to develop *Embedded Computing Systems* (ECSs). ECSs provide significantly more functionality than traditional electro-mechanical systems, and do not suffer from the same size and weight penalties; however, the size of on-board

*VERIMAG is a joint laboratory of CNRS (Centre National de la Recherche Scientifique), INPG (Institut National Polytechnique de Grenoble), UJF (Université Joseph Fourier, Grenoble-I) and VERILOG SA. VERIMAG is associated with IMAG (Institut d'Informatique et de Mathématiques Appliquées de Grenoble). SPECTRE is a project of INRIA (Institut National de Recherche en Informatique et Automatique).

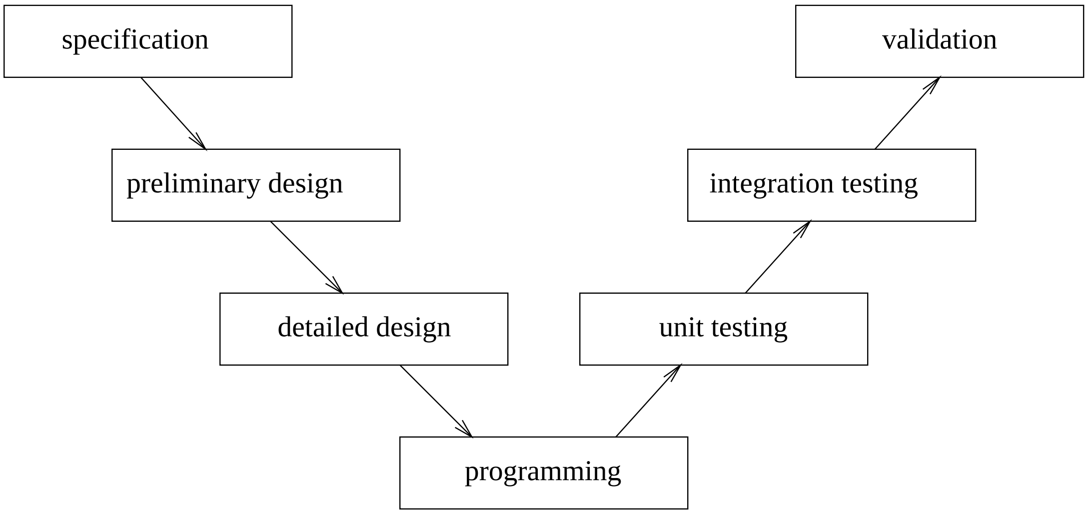

Figure 1: Traditional Software Life Cycle

software increases (for instance, 20 Mbytes in the case of the Airbus 340).

It is a major challenge for the aerospace industry to develop large complex ECSs, on time and within budget, while still satisfying high integrity requirements. Experience has led the European aerospace industry to notice that methods, tools, and environments offered by commercial suppliers to support the ECS development process do not currently address all the aerospace requirements.

For this reason, three major European aerospace companies — AEROSPATIALE (France), ALENIA (Italy), and BRITISH AEROSPACE (United Kingdom) — decided to cooperate in a joint research project, so as to compare and improve the way these companies actually develop and maintain their ECSs. This project, called AIMS[1], was initiated in 1988 with the strategic objectives of enhancing cooperation between European aerospace companies, improving productivity, and stabilizing time-schedules for the ECS development process, while ensuring that the required quality level for ECSs is achieved.

This paper presents the results of a two-year study conducted by the Aircraft Division of AEROSPATIALE as a part of the AIMS project [Hau92]. This study, named FLAIR[2] was conceived to reduce the high cost of ECSs by introducing so-called "formal methods" in the ECS development process. The FLAIR study lasted from 1991 to 1993 and involved the Aircraft Division of AEROSPATIALE, the French certification authority CEAT[3], and two research laboratories: ONERA/CERT[4] and VERIMAG.

The approach proposed and tried out in FLAIR differs from the traditional software life cycle shown on Figure 1. Assuming an ideal situation, the principles of the FLAIR approach are the following:

- The Preliminary and Detailed Design are developed using an executable formal description technique.
- The correctness of the Detailed Design is established using computer-aided verification tools.

[1] *Aerospace Intelligent Management and development environment for embedded Systems*, EUREKA project 112

[2] *Formal Language Approach Improving Requirements*

[3] *Centre d'Essais Aéronautiques de Toulouse* (Toulouse, France)

[4] *Office National d'Etudes et de Recherche Aérospatiales/Centre d'Etudes et de Recherche de Toulouse* (Toulouse, France)

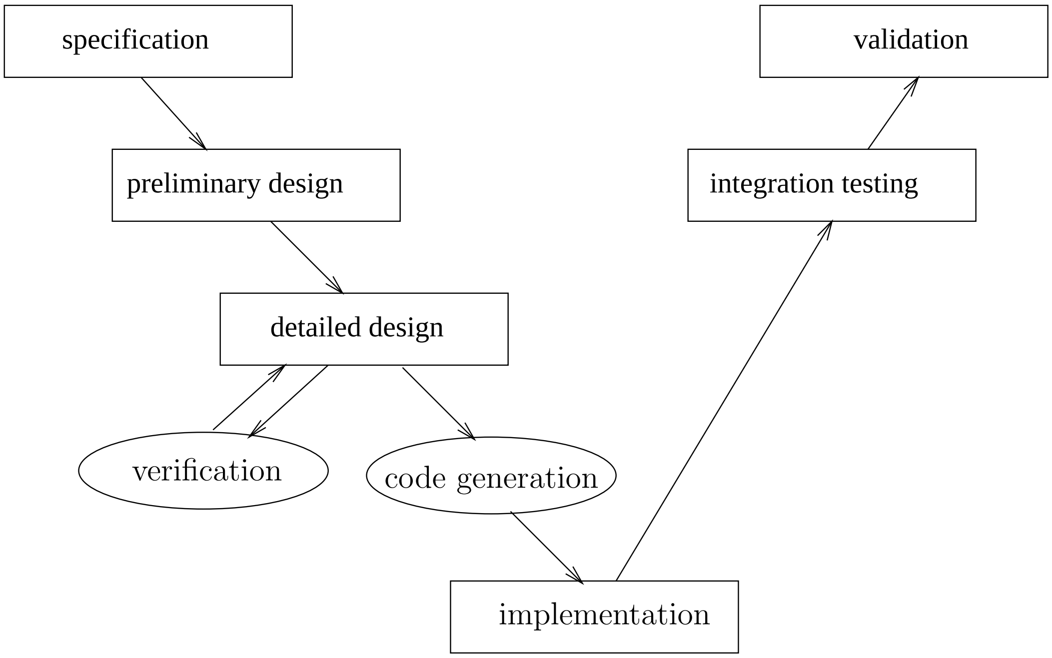

Figure 2: FLAIR Software Life Cycle

- The Detailed Design is automatically translated into executable code, using a compiler to perform the translation; the Manual Programming step is therefore suppressed.
- The Unit Testing phase is no longer necessary and can be suppressed, provided that the correctness of the compiler is proven. Since Unit Testing counts for 20% to 30% of the total software cost, significant savings can be obtained.
- The Integration Testing and Validation steps are kept.

The approach is summarized in Figure 2. The main goal of the FLAIR study was to assess the feasibility of these principles on a real-life ECS. The *Flight Warning Computer* (FWC) of the Airbus 330/340 was chosen as the target ECS for this study because its size and complexity are representative of on-board systems, and because an operational version of the FWC software (written in Ada, PLM, and assembly language) was already available for comparison.

This paper is organized as follows.

Section 1 outlines the major characteristics of the FWC and gives insights about both certification policies and current industrial practices regarding embedded real-time systems.

Section 2 briefly presents the LOTOS language that was selected as the formal description technique to be used for the Preliminary and Detailed Design of the FWC, as well as the reasons for the choice of LOTOS.

Section 3 describes the experience with the formal design of the FWC using LOTOS, as well as the problems encountered and the solutions found.

Section 4 presents the computer-aided verification tools CÆSAR, CÆSAR.ADT, and OPEN/CÆSAR, which were used to check the correctness of the FWC Detailed Design in LOTOS. The analysis strategy and the results obtained are discussed.

1 The Airbus 330/340 Flight Warning Computer

The Airbus 330/340 FWC is a system dedicated to the computation of warnings occurring on the ground or during the flight. A warning is supposed to notify the pilots of a danger or a system failure. The FWC behaves as a reactive system that continuously acquires information from the embedded systems of the whole aircraft, processes this information, and send reports — in digest and synthetic form — to the pilots. The FWC cooperates with other embedded computers: it takes its inputs from the *System Data Acquisition Concentrator* (SDAC) and delivers its outputs to the *Display Management Computer* (DMC).

The inputs of the FWC consist in analog/discrete data and digital messages (consisting of 32-bit words obeying the ARINC[5] 429 standard).

The outputs of the FWC consists of *warnings*. There are different types of warnings, such as detection of a defective system, altitude information, dangerous flight configuration, etc. A warning is a combination of different elements, which can be textual (lines of text displayed on a screen), visual (small lights), audio (sound or synthetic voice emitted through a loudspeaker), and/or digital (ARINC 429 words emitted on a bus). To each warning is associated a *severity level* (information, caution, master caution, or red warning, the latter being the most severe).

The FWC consists of several hardware cards communicating via a 32-byte parallel bus. The simplified architecture of the FWC is represented on Figure 3. The FWC is divided into three main functions, each being implemented on a specific card (respectively CPU 1, CPU 2, CPU 3) equipped with an Intel 80386 microprocessor:

The Acquisition function acquires data from ARINC and discrete cards, checks the validity of these data, extract and stores signal values into the Mailbox RAM (this shared memory is used by the Acquisition, Warning Computation, and Warning Presentation functions to communicate), and emits ARINC outputs through the QAT[6].

The Warning Computation and Presentation functions receive signals from the Mailbox RAM (there are about 11,000 possible signals, which are registered in the FWC database implemented in an EPROM), translate these signals into arrays of 2,000 boolean alarms, emit textual and visual outputs, send requests to the Sound Generation function, and put status messages in the Mailbox RAM to be consulted by the Acquisition function. The determination and the delivery of outputs obey complex and imperative rules, which take into account the severity level, the warning type, the current flight phase, and the actions of the pilots.

The Sound Generation function executes the requests of the Warning Presentation function. It is responsible for producing sounds and synthetic voice.

In the software of each CPU, there is a clear separation between real-time aspects, algorithms, and data structures.

Also, it is worth noticing that, for the FWC and other ECSs in general, the real-time aspects only represent a very small fraction of the embedded code (less than 5% in the case of the FWC). Moreover, the real-time concepts used are very simple: they follow the coroutine model, with foreground and background tasks, hardware interrupts, and timers.

Both the separation and the simplicity are a consequence of the rigorous regulations concerning embedded software[7]. The general principle can be summarized as follows: the real-time part of a

[5] *Aeronautical Radio INCorporated*
[6] *Quadruple ARINC transmitter*
[7] especially the joint document EUROCAE/RTCA DO 178

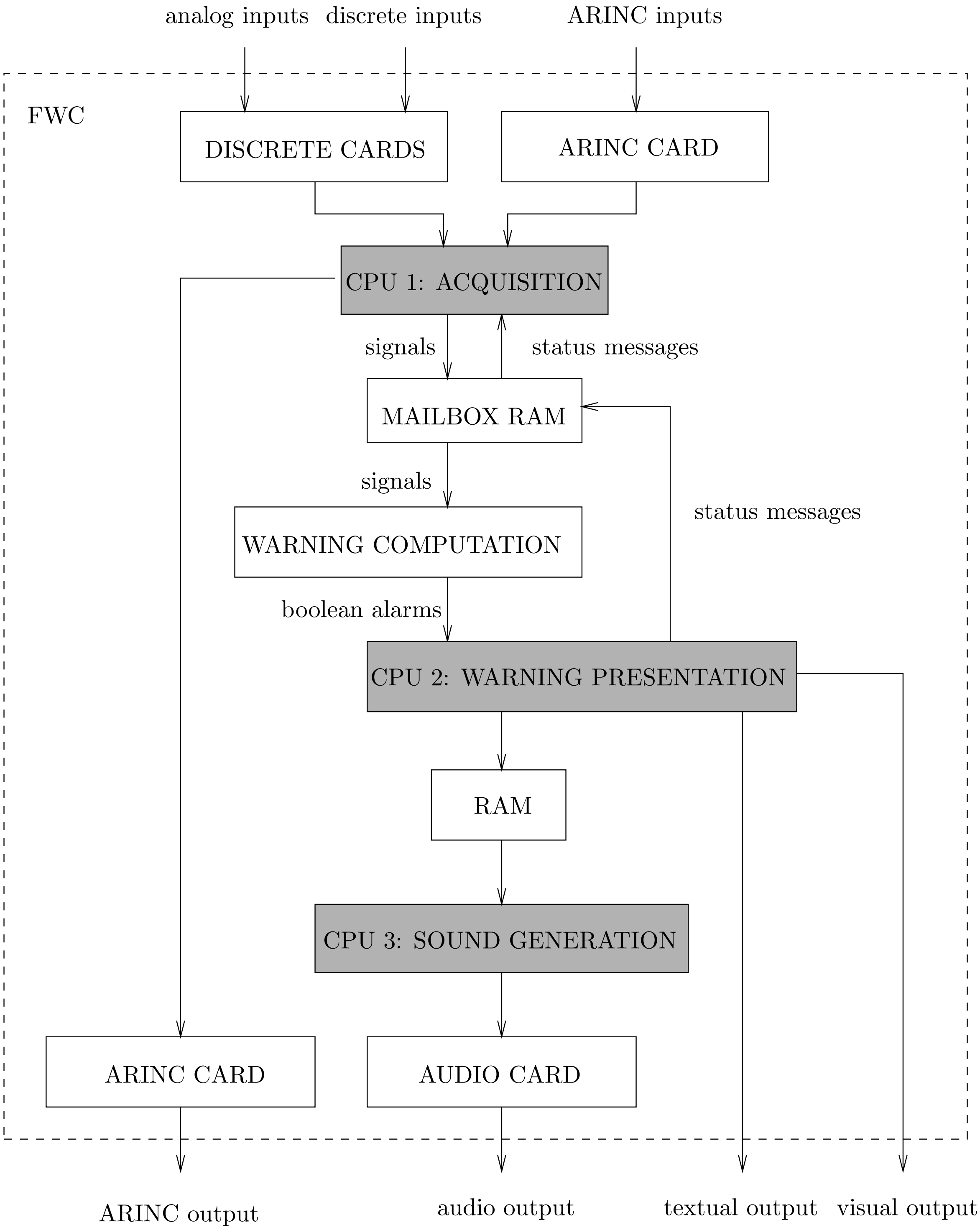

Figure 3: Simplified Architecture of the Flight Warning Computer

critical system must be simple, otherwise it will not be certified, since complex real-time designs inspire little confidence. Many important requirements stem from this general principle: for instance, in any Ada program to be certified, use of tasking is prohibited.

Regarding real-time aspects, the three CPUs of the FWC share common properties:

- Each CPU has two functioning modes: boot mode, which is activated when the computer is started or in case of power failure recovery, and operational mode, which takes place after the boot is completed.
- Each CPU has a Built-In Test Equipment (BITE) function, which is an auto-test program that executes in operational mode and checks the computer functioning. If the BITE of CPU 3 detects a problem in CPU 3, this problem is reported to CPU 2. Similarly, the BITE of CPU 2 reports to CPU 1 the problems it encounters. CPU 1 has the master BITE that is in charge of confirmation/correlation and takes appropriate decisions such as warning emission, halting, rebooting, or continuation.

However, the real-time behaviors of the three CPUs differ:

- In CPU 1, the BITE is running as a background task. Every 15 milliseconds (ms), the BITE is interrupted by a timer interruption. This interruption triggers the Acquisition Function, which runs for 10 ms. Afterwards the BITE resumes for 5 ms, until the Acquisition Function is triggered again.
- CPU 2 integrates two functions (a Warning Computation function involving combinatorial logics, and a Warning Presentation function based on a real-time monitor), which run continuously. Periodically, a timer interrupt provides 10 ms to the BITE.
- CPU 3 works in slave mode with respect to CPU 2: the BITE executes continuously, until CPU 3 receives a request from CPU 2 for sound generation. In that case, the BITE is interrupted in order to execute the request and resumes after the request is processed.

The remaining 95% of the FWC software consist in algorithms and data structures. The main difficulties in the development of this software arise from the complexity of the algorithms and the size of the data structures.

The FWC software obeys another important certification requirement: embedded software must be fully deterministic in order to be certified. Determinism means that, for given input sequences, the computer must always produce the same output sequences and, furthermore, always follow the same execution paths. The assumption of determinism greatly simplifies the testing phase, by allowing the observed events to be reproduced.

2 The Formal Description Technique LOTOS

As stated above, the principles of the FLAIR approach rely on the use of a formal description language, supported by computer-aided verification tools and provably-correct compilers. Therefore, it is a prerequisite for this language to have a mathematically-defined and well-founded semantics.

At the beginning of the FLAIR study, several languages were evaluated with respect to various criteria, including the existence of a formal semantics, the language expressiveness (and, especially, the ability to deal with complex data structures), the level of use of the language, and the availability of support tools. Finally, LOTOS was selected as the language to be used for the Preliminary and Detailed Design of the FWC.

LOTOS[8] is a Formal Description Technique for communication protocols and distributed systems. It was developed during the years 1981–88 in the framework of the SEDOS[9] project and standardized by ISO[10] in 1988 [ISO88, BB88]. The design of LOTOS was motivated by the need for a language with a high abstraction level and a strong mathematical basis, which could by used for the description and analysis of complex systems. As a design choice, LOTOS consists of two distinct and "orthogonal" sub-languages:

the data part of LOTOS is dedicated to the description of data structures. It is based on the well-known theory of algebraic abstract data types [Gut77], more specifically on the ACTONE specification language [EM85].

In this approach, data structures are described by *sorts*, which represent value domains, and *operations*, which are mathematical functions defined on these domains. The meaning of operations is defined by algebraic *equations*. Sorts, operations, and equations are grouped in modules called *types*, which can be combined together using importation, renaming, parametrization, and actualization. The underlying semantics is that of initial algebras [EM85].

the control part of LOTOS is based on the process algebra approach, which has been proposed for the study of concurrent systems. Examples of process algebras are CCS [Mil80, Mil89], CSP [Hoa85], ACP [BK84], and LOTOS, which seems to merge the best features of CCS and CSP.

As a process algebra, LOTOS relies on a small set of basic operators, which represent primitive concepts of concurrent systems (sequential composition, non-deterministic choice, guard, parallel composition, rendez-vous, etc.) These operators are used to build algebraic terms, which express concurrent system behaviors; the approach is compositional, since complex behaviors can be obtained by combining elementary ones.

The formal semantics of process algebras is usually defined in terms of *labelled transition systems* (LTSs) [Par81, Mil80, ISO88], i.e. directed graphs whose vertices denote global states of the system and whose edges correspond to the evolutions of the system (transitions). However, other concurrency models, especially Petri Nets, seem to be appropriate when dealing with process algebras [Old91, GS90, Tau90, BvB90, ML88].

Due to its characteristics, LOTOS has received considerable attention from the research community. Many prototype tools have been developed for this language, including simulators, compilers, test generators, analysis and verification tools, etc. LOTOS has been used in various industrial applications but, to the authors' knowledge, the FLAIR study is probably the first attempt at using LOTOS to develop avionics software for a real-life ECS. There are some related experiments with other formal methods: for instance TLA [Lam93] was used to describe and analyze the Airbus 320 braking system [Sin94].

3 Formal Description in LOTOS of the FWC

The starting point for developing a formal description of the FWC was the specification elaborated by AEROSPATIALE for the existing FWC. This specification consists in a collection of requirements

[8] *Language Of Temporal Ordering Specification*
[9] *Software Environment for the Design of Open Distributed Systems*, ESPRIT project 410
[10] *International Organization for Standardization*

(thousands of pages, divided in several books) expressed in both natural language and semi-formal diagrams.

At the beginning of the FLAIR study, the applicability of LOTOS to the formalization of these requirements was investigated, but difficulties were faced:

- The requirements are extremely diverse. Some of them are outside the scope of computer science, for instance mechanical requirements ("*The FWC rack must contain at least 50% spare space to allow future extensions*") and electrical requirements ("*The FWC must resist to any power failure, whatever its duration*"), etc. Other requirements deal with software, but cannot be formalized easily: ("*The FWC is parametrized with a signal database and must be able to accept any signal database for configuration*").
- Because it is a formal description technique, LOTOS requires from the designers a clear knowledge of their intentions. However, during the Specification phase, most design decisions are not available, since they are the concern of subsequent design phases.

It was therefore decided to use LOTOS at the Preliminary Design and Detailed Design levels, for the three main functions of the FWC (Acquisition, Warning Presentation, and Sound Generation).

During the Preliminary Design phase, the architecture of each function was determined in a classical way. Each CPU was decomposed into elements, which were mapped to LOTOS types and/or processes. Three LOTOS descriptions were elaborated; the definitions of some types and processes were left empty (deferred to the subsequent Detailed Design phase).

During the Detailed Design phase, the three descriptions were refined so as to obtain complete LOTOS descriptions. Algorithms and data structures were defined formally, in order to allow automatic analysis and code generation.

The Preliminary and Detailed Design descriptions were written in an "extended LOTOS" dialect, i.e. an extension of LOTOS with a set of macro-notations, intended to simplify and to improve the quality of the design. A pre-processor tool was developed during the FLAIR study, which automatically translates "extended LOTOS" descriptions into standard LOTOS ones. This pre-processor was systematically applied before using analysis tools; on the other hand, the code generation tools were designed to operate directly at the "extended LOTOS" level, in order to take advantage of the additional informations provided by LOTOS extensions (see Figure 4).

First of all, the pre-processor accepts a "`#include`" directive[11], which introduces a limited form of modularity by allowing the LOTOS description to be split into several files. The other extensions are detailed below, as the three main description issues (data, algorithms, and real-time) are presented in turn.

3.1 Description of the FWC Data Structures

A real-life embedded system like the FWC involves a variety of data types, and could not be adequately described using a language with a limited type system. Indeed, the complete FWC Detailed Design contains approximately 250 types, which can be divided into several classes:

Usual types such as bits, booleans, characters, bytes, integers, hexadecimal numbers, 16-byte and 32-byte words, character strings, bit strings, etc. From a developer's point of view, the standardized LOTOS library [ISO88, appendix A] is not as helpful as it could be, since some usual types must be extended, while other ones have to be created from scratch.

[11] same as in the C language

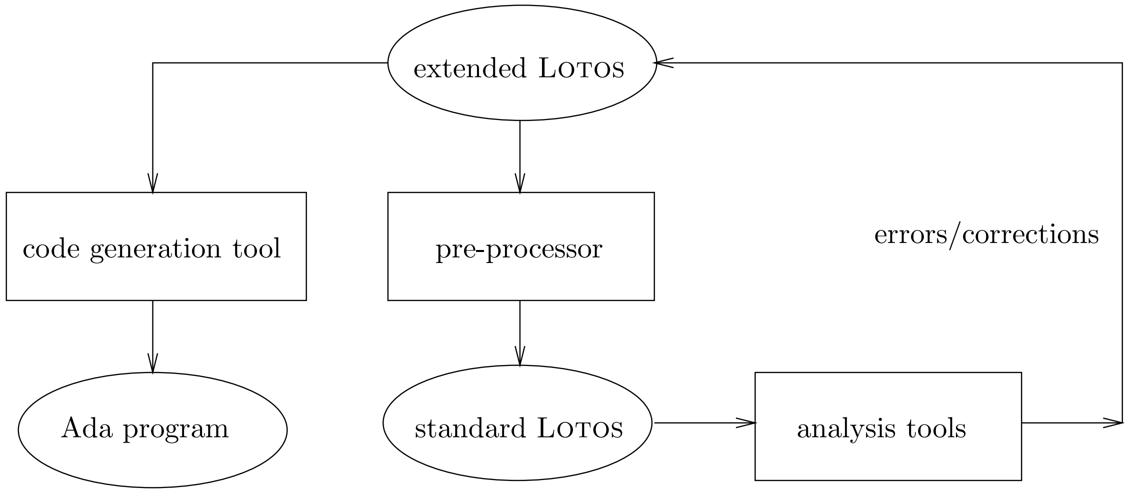

Figure 4: Development Environment of the FLAIR Study

Enumerated types are heavily used. Although the "pure" abstract data types mechanism of LOTOS allows the description of arbitrarily complex data structures, it makes no provision for particular classes of data types, such as enumerated types. Therefore, the FLAIR pre-processor recognizes an abbreviation "`enum...endenum`" which allows to define enumerated types concisely. For instance, the following declaration:

```
enum ALARM_STATUS is
    ABSENT, PRESENT, INHIBITED, PENDING, CANCELED, TRAPPED
endenum
```

expands into a standard LOTOS type definition containing one sort and six enumerated values. Additional operations are generated: comparison functions, integer conversion functions (from enumerated values to ordinal numbers and vice-versa), etc.

Record types are frequently used. The FLAIR pre-processor provides a "`struct...endstruct`" clause to define record types in a compact way. For instance, the following declaration:

```
struct AUDIO_DATA is BOOLEAN, TIME, AUDIO_TYPE, AUDIO_COMMAND
     DURATION : TIME,         (* sound duration, in 100 ms *)
     TYPE     : AUDIO_TYPE,   (* sound type, brief or long *)
     CODE     : AUDIO_COMMAND (* sound emission code *)
endstruct
```

declares an abstract data type containing one sort with three fields. Auxiliary operations are also defined, such as equality and inequality functions, and access functions, which extract or modify the contents of the field.

Memory types allow the FWC resources other than the three CPUs to be modeled: EPROM, EEPROM, Mailbox RAM, RAM, ARINC memory, analog/discrete ports, QAT, etc. All these resources can be described as memories, some of them are of large size (the RAMs and EPROMs of the existing FWC range from 32 Kbytes to 512 Kbytes). The FLAIR pre-processor implements an appropriate "`mem...endmem`" construct. For instance, the declaration below:

```
mem EPROM is TEXT, ALARM, AUDIO_DATA, HYBRID_DATA
readonly
     PAGE_WARNING, PAGE_STATUS, T4962100L1, T7125210L3, ... : TEXT,
     W4962100, W7103070, W7125130, W7325214, ... : ALARM,
     SORTED_ALARM_LIST : ALARM_LIST,
     NO_SOUND, CRC, CRICKET, CAVALRY, BUZZER, CLICK, ... : AUDIO_DATA,
     NO_SYNTH_VOICE, FOUR_HUNDRED, THREE_HUNDRED, ... : AUDIO_DATA,
     WIND_SHEAR, PRIORITY_LEFT, HUNDRED_ABOVE, LATE, ... : AUDIO_DATA,
     NO_HYBRID, CRICKET_STALL : HYBRID_DATA,
endmem
```

generates a collection of LOTOS types, which define a memory consisting of a set of locations. Each location may either be undefined, or contain a value of a given sort (`TEXT`, `ALARM`, ..., or `HYBRID_DATA` in the example above). It is accessed using an address, which is an enumerated value (`PAGE_WARNING`, `PAGE_STATUS`, ... `CRICKET_STALL` in the example above).

Basically, a memory is represented as a list of pairs (a_i, v_i), meaning that the location at address a_i contains value v_i. This representation may seem unnecessarily complex and somewhat artificial, but it is intended to detect non-initialized variables (i.e. variables used before set). In fact, the chosen representation is slightly more complex, because it avoids polymorphism on location contents, thus allowing type-checking to be done at compile-time.

When describing the data part of each CPU, only a restricted subset of LOTOS was used. Parametrized types were avoided, as the Ada code of the existing FWC does not rely on genericity. Because the development of large abstract data types description needed a methodology, the so-called "constructor-oriented" style was adopted: equations are oriented (i.e. considered as term rewrite rules), the set of operations is split into *(free) constructors* (primitive operations) and *non-constructors* (derived operations, defined in terms of constructors), constructors are "marked" explicitly, and equations between constructors are prohibited.

3.2 Description of the FWC Algorithms

The algorithmic aspects probably constitute the most important fraction of the FWC code. Regarding algorithmics, the three LOTOS descriptions for the FWC Detailed Design were developed using common design principles:

- Each description represents a set of parallel processes corresponding to the CPU and all the resources accessed by the CPU (RAM, EPROM, etc.) These processes synchronize and communicate using the rendez-vous mechanism provided by LOTOS. For instance, the overall architecture of CPU2 is described as follows:

```
START >> accept
     INIT_RAM:RAM,
     INIT_DP_RAM:DP_RAM,
     INIT_MB_RAM:MB_RAM,
     INIT_EPROM:EPROM,
     INIT_PORT:PORT,
     INIT_QAT:QAT
in
(
```

```
        CPU_2 [INTERRUPT.
               READ_RAM, WRITE_RAM,
               READ_DP_RAM, WRITE_DP_RAM,
               READ_MB_RAM, WRITE_MB_RAM,
               READ_EPROM,
               READ_PORT, WRITE_PORT,
               READ_QAT, WRITE_QAT]
        |[
        READ_RAM, WRITE_RAM,
        READ_DP_RAM, WRITE_DP_RAM,
        READ_MB_RAM, WRITE_MB_RAM,
        READ_EPROM,
        READ_PORT, WRITE_PORT,
        READ_QAT, WRITE_QAT
        ]|
        (
             RAM_MANAGER [READ_RAM, WRITE_RAM] (INIT_RAM)
             |||
             DP_RAM_MANAGER [READ_DP_RAM, WRITE_DP_RAM] (INIT_DP_RAM)
             |||
             MB_RAM_MANAGER [READ_MB_RAM, WRITE_MB_RAM] (INIT_MB_RAM)
             |||
             EPROM_MANAGER [READ_EPROM] (INIT_EPROM)
             |||
             PORT_MANAGER [READ_PORT, WRITE_PORT] (INIT_PORT)
             |||
             QAT_MANAGER [READ_QAT, WRITE_QAT] (INIT_QAT)
        )
   )
```

In this architecture, the CPU (process `CPU_2`) has a central position. It accesses all six resources (processes `RAM_MANAGER`, ..., `QAT_MANAGER`), either for reading (gates `READ_RAM`, ..., `READ_QAT`) or writing (gates `WRITE_RAM`, ..., `WRITE_QAT`). The resources evolve independently and only communicate with the CPU. There is an initialization phase (process `START`), which loads values into the memories and computes their initial contents (variables `INIT_RAM`, ..., `INIT_QAT`).

- Since LOTOS is a purely functional parallel language (based on the value-passing paradigm), it has no built-in concept of shared memory. It is possible, however, to describe a memory using the constructs provided by LOTOS. As shown above, each resource is encapsulated in a LOTOS process (e.g., the `RAM_MANAGER`) which allows read and write access by means of rendezvous; in the case of read-only memories like EPROMs and EEPROMs, write access is not offered. For each "`mem..endmem`" declaration, the FLAIR pre-processor automatically builds the corresponding resource manager process.

- The LOTOS processes describing the behaviors of CPU 1, CPU 2, and CPU 3 are written by hand. The behavior of each CPU is sequential. Basically, each CPU acquires input data by reading the memories, checks for the validity of these input data, computes output data, and delivers these output data by writing into the memories; however, the details of the algorithms are actually more complex than suggested here.

  In the existing FWC, the CPU algorithms are programmed in Ada, PLM, and assembly language. To describe them in LOTOS, it was necessary to change from a classical programming style (based on assignments, "**if**", "**case**", "**while**", and "**for**" constructs) to a functional style (involving sequential composition, guarded commands, and recursion), which proved to be possible, but tricky. An example of a simple LOTOS process is shown below:

```
process PUT_LABEL_357 [READ_RAM, WRITE_RAM] (L:QAT_WORD_LIST) : exit :=
     [L = {} of QAT_WORD_LIST] ->
          exit
     []
     [L <> {} of QAT_WORD_LIST] ->
          (
          [BIT (8, HEAD (L)) = 0 of BIT] ->
               READ_RAM !FIFO_357 ?FIFO:QAT_WORD_LIST;
               WRITE_RAM !FIFO_357 !ADD (FIFO, HEAD (L));
               PUT_LABEL_357 [READ_RAM, WRITE_RAM] (TAIL (L))
          []
          [BIT (8, HEAD (L)) = 1 of BIT] ->
               PUT_LABEL_357 [READ_RAM, WRITE_RAM] (TAIL (L))
          )
endproc
```

A difficulty was found when describing the algorithms in LOTOS: new variables had to be introduced during the Detailed Design phase, since it was not possible to determine completely the set of relevant variables during the Specification and Preliminary Design phases. Because LOTOS is a purely functional language, the only way to specify global variables is to declare them as process formal parameters. Therefore, introducing a new variable often brings changes to many process headers and process calls.

3.3 Description of the FWC Real-Time Aspects

As explained above, the real-time part of the FWC and other ECSs is clearly separated from the data structures and algorithms, and only counts for 5% of the FWC embedded software. An attempt was done at using LOTOS to formalize these aspects, but this revealed not to be feasible.

Timeouts, watchdogs, and task context switchings could not be expressed, since LOTOS has no concept of "urgent action". The disabling operator "`[>`" of LOTOS is not adequate for this purpose, for two reasons:

- Its effects are non-deterministic: a disrupted process may or may not continue its execution. To express the FWC real-time features, preemptive disruption is necessary [Ber93].
- When a process is disrupted, its execution aborts ("interrupt-and-terminate" paradigm). On the opposite, FWC tasks behave as coroutines ("interrupt-and-resume" paradigm).

For this reason, the real-time parts of the FWC were not described in LOTOS, but expressed as chronograms and left to be manually programmed in Ada (using hardware interrupts, not tasks).

3.4 Results and Assessment

The table below gives quantitative informations about the LOTOS descriptions developed during the FLAIR. For each description, L represents the number of lines of LOTOS, L' the number of lines of

LOTOS (blank lines and comments excluded), T the number of types (libraries included), and P the number of processes.

Function	*Design phase*	L	L'	T	P
CPU 1 (Acquisition)	Preliminary	6,706	5,432	117	84
	Detailed	7,869	6,098	107	60
CPU 2 (Warning Presentation)	Preliminary	5,348	4,048	82	67
	Detailed	13,506	10,652	145	101
CPU 3 (Sound Generation)	Preliminary	3,427	2,454	75	53
	Detailed	6,165	4,865	73	87
TOTAL	—	43,021	33,549	599	452

These descriptions are certainly among the largest LOTOS programs developed today. Their data parts are also among the largest existing abstract data type specifications (according to the overview of case studies given in [EC90]). Indeed, the Detailed Design of CPU 2 has not less than 145 types, 145 sorts, 515 constructors, 908 non-constructors, and 1432 equations!

It was concluded that LOTOS could be used to describe parts of the Preliminary Design of a typical ECS like the FWC. However, the description is only partial, because real-time aspects cannot be expressed using LOTOS.

As for the Detailed Design, it was concluded that LOTOS could be applied. But in doing so, LOTOS was mostly used as a functional language, with a top-down approach, to describe data structures and sequential algorithms. Indeed, the LOTOS description of CPU 2 has approximately the same size as the corresponding Ada code for CPU 2, and both descriptions are more or less at the same abstraction level.

A quantitative assessment concluded that the effort spent in developing the CPU 2 description in LOTOS was about 30% higher than its counterpart in Ada. Three main reasons were identified: the formation time to learn LOTOS was longer than for Ada; to manage large descriptions, the FLAIR pre-processor had to be developed (about 3,000 lines of C); the debugging phase was longer since the diagnostics provided by the LOTOS tools were not explicit enough.

4 Analysis of the Detailed Design

After completion of the Detailed Design, the AEROSPATIALE team evaluated various LOTOS tool environments for assessment of their capabilities and performances. As the result of this evaluation, the CADP[12] toolbox [FGM+92] was selected for the rest of the FLAIR project.

4.1 The CADP Toolbox for LOTOS

The CADP toolbox is an integrated set of tools dedicated to the efficient compilation, simulation, testing, and verification of descriptions written in LOTOS. It has been developed since 1985 and currently contains several closely interconnected components[13]:

- ALDÉBARAN [Fer90, FM91b] is a tool for verifying communicating systems represented by labelled transition systems (LTS). It allows the reduction and the comparison of LTSs modulo various equivalence and preorder relations. The verification algorithms used in ALDÉBARAN

[12] CÆSAR/ALDÉBARAN *Distribution Package*

[13] The CADP toolbox can be obtained by sending an e-mail request to `caesar@imag.fr`

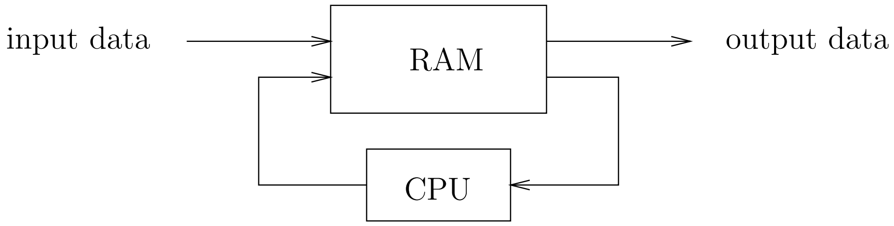

Figure 5: The CPU as a dynamic system

are based on the Paige-Tarjan algorithm [PT87] for computing the relational coarsest partition, on *on-the-fly* techniques [FM91a], and on symbolic verification techniques based on *Binary Decision Diagrams* (BDDs) [FKM93].

- CÆSAR.ADT [Gar89, GT93] is a compiler that translates the data part of LOTOS specifications (assuming the constructor-oriented style) into libraries of C types and functions. Each LOTOS sort is translated into a C type and each LOTOS operation is translated into a C function.

- CÆSAR [GS90] is a compiler for the behavior part of LOTOS. It translates LOTOS descriptions into LTSs that can later be checked, using either verification tools for LTSs (like ALDÉBARAN) or temporal logic evaluators. The translation algorithms of CÆSAR proceed in successive steps, using various intermediate forms, namely a Petri Net model extended with variables, which provides a compact and structured representation for both the control and data flows. From this model, a C program is generated; once compiled with the implementation produced by CÆSAR.ADT for the data part, it is executed, giving the LTS corresponding to the LOTOS description.

- OPEN/CÆSAR [Gar92] is an programming environment that allows user-defined customized programs for simulation, execution, verification, testing, etc. to be created in a simple and modular way.

4.2 The Strategy Used for Correctness Checking

In a rough approximation, the behavior of each CPU could be modelled by a function taking input data and returning output data. The situation is more complex in fact, since some resources are both read and written, and since the input data are not all collected at the same time, but at different dates determined by the bus clock, the acquisition period, and the memory refreshment period. Hence, the CPU behaviour is closer to a dynamic system (see Figure 5).

Given the current state of the art, it seems that the FWC complexity is beyond the capabilities of existing automated proof techniques:

- The algorithms are complex and involve large data structures. They cannot be simply expressed as boolean or algebraic functions. The CPU 2 description has more than 10,000 lines of code, which probably exceeds the limitations of current theorem provers.

- Also the state space is extremely large. For instance, CPU 1 must deal with combinations of more than 11,000 different signals. Similarly, the inputs of Warning Presentation of CPU 2 consist in an array of more than 2,000 boolean alarms (a state space of $2^{2,000}$ elements is currently too large, even for the best existing boolean decision methods).

Due to these intrinsic limitations, full verification was untractable. It was therefore agreed to do partial verification only. The approach followed during the FLAIR study was a "black-box" approach combining simulation and testing techniques:

- For each CPU, a corresponding implementation in C was generated using the CÆSAR and CÆSAR.ADT compilers.
- This implementation was run on different sets of input data; for each of them, the validity of the set of emitted output data was controlled visually by an AEROSPATIALE engineer. Obviously, this is not complete verification, since all sets of input data could not be tried.

  The various sets of input data to be tested were not determined randomly, but according to a precise methodology. In the case of CPU 1, the 11,000 possible signals were arranged in 32 sets, since the time cycle for the whole FWC is 480 ms and the period of the Acquisition function is 15 ms (32 = 480/15). The same signal may occur in several sets, depending on its periodicity.

  In the case of CPU 2, the input data of CPU 2 consist of both the boolean arrays produced by the Warning Computation function and the values computed by CPU 2 during the previous cycles and stored in the RAM. The output data of CPU 2 consist of all the values stored in the Mailbox RAM (which interfaces CPU 1 and CPU 2), in the DP RAM (which interfaces CPU 2 and CPU 3), in the RAM used by CPU 2, all 8-byte and 16-byte words emitted on the discrete ports, etc.
- Because of determinism, the same inputs yield the same outputs. The LTS corresponding to the CPU evolution for any given set of input data is a *finite chain* (i.e. a transition system where each state has a single successor state, except the last state state, which has no successor). Given this remarkably simple LTS structure, the usual distinctions between simulation, random execution, and testing disappear, since these three notions coincide. In this particular case, it was not necessary to use sophisticated verification tools for LTSs (such as ALDÉBARAN), which are intended for the verification of complex systems, where concurrency and non-determinism play an important role.
- The OPEN/CÆSAR environment was used to interface and to monitor the implementation generated by CÆSAR and CÆSAR.ADT. The input data were read from various files (each corresponding to a given resource) and the output data were written on other files to be examined later. The exploration of the LTS was done by writing a small traversal program checking for the absence of non-determinism. The skeleton of this program was the following:

```
var s1, s2 : state
var l : label
var L : set of ⟨label, state⟩
begin
s1 := initial_state()
repeat
  begin
  L := {⟨l, s2⟩|there exists a transition s1 --l--> s2}
  if card(L) = 0 then
    print "execution terminated"
  else_if card(L) ≥ 2 then
    print "non-determinism found at state s1; execution aborted"
  else_if card(L) = 1 then
    begin
    let l, s2 such_that L = {⟨l, s2⟩}
    s1 := s2
    if l concerns an input resource then
      read the value of l from the corresponding file
    else_if l concerns an output resource then
      write a value of l to the corresponding file
    else_if l concerns an internal action then
      do nothing
    end
until card(L) ≠ 1
end
```

4.3 Results and Assessment

During the FLAIR study, the CÆSAR and CÆSAR.ADT tools were used to perform syntax analysis, static semantics checking (identifier binding, type checking, etc.), and C code generation on both the CPU 1 and CPU 2 Detailed Design descriptions. Due to a lack of time, CPU 3 was not fully analyzed.

For the largest description (CPU 2), quantitative measurements were done on a SUN SparcStation 10. Using CÆSAR.ADT (version 4.1) for the data part, this description compiled in 30 seconds, producing a C program of 1,074,158 bytes. Using CÆSAR (version 4.8) for the algorithmic part, this description compiled in less than 5 minutes, generating a C program of 6,029,040 bytes. By compiling and linking together these two C programs, a object program of 2,446,712 bytes was obtained. The speed of the generated code was reasonable, since a set of input data could be processed in some minutes. As for memory consumption, 1,728 bytes were needed to store the global state of CPU 2, not including the memories and other length-varying data structures.

The "black-box" approach allowed the detection and correction of various design errors, namely:

1. **erroneous or missing algebraic equations.**
2. **occurrence of non-determinism,** which would have been difficult to discover by non-automated methods due to the intrinsic complexity of the CPUs: for instance, the behavior of the CPU 2 is equivalent (as modelled by CÆSAR) to an extended Petri net with 1,377 places, 3,411 transitions, 7 tokens, and 665 state variables!

3. **premature termination,** which was caused either by inappropriate boolean guards or by impossible rendez-vous synchronizations;
4. **non-initialized variables,** which were discovered when trying to access memory locations that were not assigned previously.

At the beginning, diagnosis problems were faced in using the tools: it was sometimes difficult for the users to understand the reason of an error. During the FLAIR study, CÆSAR and CÆSAR.ADT were improved to incorporate debugging and tracing facilities.

Conclusion

The aerospace industry is currently confronted to the high cost of Embedded Computing Systems (ECS), as these systems become more and more complex due to the increasing need of integrated functions. In the framework of the AIMS project, the FLAIR study focused on embedded software (the primary cost factor for ECSs) by trying to reduce the time and effort spent in the Unit Testing phase (which is estimated to be 20–30% of the total software cost for an ECS). The FLAIR approach is based on the introduction of formal methods in the software life cycle, especially at the Preliminary and Detailed Design levels.

This approach took place with the ISO language LOTOS on a representative case study, the Flight Warning Computer (FWC) of the Airbus 330/340. The FLAIR experience is original in the sense that LOTOS was not used in its native application field (telecommunication protocols), but to design a significant subset of the FWC embedded software, traditionally developed in Ada, PLM, or assembly language.

Six large LOTOS descriptions were developed for the FWC, totaling more than 33,000 lines of LOTOS. Using the CÆSAR and CÆSAR.ADT tools, two of these descriptions (including the largest one) were successfully compiled and partially verified.

However, from an industrial point of view, and given the current state of the art, the FLAIR approach is not operational today, and cannot replace the traditional ECS development process.

The main reason for this is the lack of a key component in the FLAIR software development chain. To be effective, the FLAIR approach relies on a compiler, which should be provably-correct and capable of translating LOTOS descriptions into Ada or PLM code to be certified and embedded. During the FLAIR study, an attempt was made to develop such a compiler, based on an implementation by ONERA/CERT of the LF[14] formalism developed at the University of Edinburgh [HHP87]. However, the results are not conclusive, since the Ada code generated by this compiler does not meet the performance requirements for embedded software. Moreover, this code is highly recursive, and consequently rejected by the certification authority (recursion is forbidden, unless it is formally established that the program stack does not overflow). Therefore, the development of efficient compiling techniques for algebraic languages like LOTOS remains an open problem.

Regarding the kind of language suitable for on-board software design, the FLAIR study concluded that a single language, to be effective, should have a formal semantics and provide high-level constructs to express (1) data structures, (2) algorithms, (3) real-time aspects as well. As far as LOTOS is concerned, it is expected that the undergoing revision at ISO of the LOTOS standard will introduce desirable extensions to the language, which would ease the development of large ECS descriptions:

1. Regarding data structures, the library of predefined types should be extended. Notational

[14] *Logical Framework*

facilities should be available for defining enumerated types, records, discriminated unions, etc. Such facilities have been suggested, for instance, in the ESPRIT projects LOTOSPHERE and SPECS, and, more recently, in the European-Canadian project EUCALYPTUS [Pec93].

2. Regarding algorithms, classical structured programming constructs (such as "**if**", "**case**", "**while**", "**for**", etc.) are currently missing and should be added if possible;

3. Regarding real-time aspects, ECSs must be kept simple in order to meet certification requirements. It is therefore likely that the real-time aspects can easily be expressed using timed process algebras such as ATP [NS91, NS94] or timed versions of LOTOS [BL91, LL92, QFA93].

Although the objectives of the FLAIR study were probably too ambitious for the current state of the art in formal methods, the authors still believe that the FLAIR principles are sound: formal methods, verification techniques, and automatic code generation could be profitably introduced in the ECS development process.

Acknowledgements

The authors are grateful to the members of the FLAIR group and the partners of the AIMS project, and especially the British team for sharing their experience in the field of formal methods. Acknowledgements are due to Florence Baudin, Laurent Mounier, John Plaice, and Joseph Sifakis for their helpful comments on this paper.

References

[BB88] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, January 1988.

[Ber93] Gérard Berry. Preemption and Concurrency. In *Proceedings of FSTTCS 93*, volume 761 of *Lecture Notes in Computer Science*, pages 72–93. Springer Verlag, 1993.

[BK84] J. A. Bergstra and J. W. Klop. Process Algebra for Synchronous Communication. *Information and Computation*, 60:109–137, 1984.

[BL91] T. Bolognesi and F. Lucidi. LOTOS-like Process Algebras with Urgent or Timed Interactions. In Kenneth R. Parker and Gordon A. Rose, editors, *Proceedings of the 4th International Conference on Formal Description Techniques FORTE'91*. North-Holland, 1991.

[BvB90] Michel Barbeau and Gregor v. Bochmann. Deriving Analysable Petri Nets from LOTOS Specifications. Publication 707, Département IRO, Université de Montréal, January 1990.

[EC90] Harmut Ehrigh and Ingo Claßens. Overview of Algebraic Specification Languages, Environments and Tools, and Algebraic Specifications of Software Systems (part 3). *Bulletin of the EATCS*, 41:145–153, June 1990.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1 — Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer Verlag, 1985.

[Fer90] Jean-Claude Fernandez. An Implementation of an Efficient Algorithm for Bisimulation Equivalence. *Science of Computer Programming*, 13(2–3):219–236, May 1990.

[FGM+92] Jean-Claude Fernandez, Hubert Garavel, Laurent Mounier, Anne Rasse, Carlos Rodríguez, and Joseph Sifakis. A Toolbox for the Verification of LOTOS Programs. In Lori A. Clarke, editor, *Proceedings of the 14th International Conference on Software Engineering ICSE'14 (Melbourne, Australia)*, pages 246–259. ACM, May 1992.

[FKM93] Jean-Claude Fernandez, Alain Kerbrat, and Laurent Mounier. Symbolic Equivalence Checking. In C. Courcoubetis, editor, *Proceedings of the 5th Workshop on Computer-Aided Verification (Heraklion, Greece)*, volume 697 of *Lecture Notes in Computer Science*. Springer Verlag, June 1993.

[FM91a] Jean-Claude Fernandez and Laurent Mounier. "On the Fly" Verification of Behavioural Equivalences and Preorders. In K. G. Larsen and A. Skou, editors, *Proceedings of the 3rd Workshop on Computer-Aided Verification (Aalborg, Denmark)*, volume 575 of *Lecture Notes in Computer Science*, Berlin, July 1991. Springer Verlag.

[FM91b] Jean-Claude Fernandez and Laurent Mounier. A Tool Set for Deciding Behavioral Equivalences. In *Proceedings of CONCUR'91 (Amsterdam, The Netherlands)*, August 1991.

[Gar89] Hubert Garavel. Compilation of LOTOS Abstract Data Types. In Son T. Vuong, editor, *Proceedings of the 2nd International Conference on Formal Description Techniques FORTE'89 (Vancouver B.C., Canada)*, pages 147–162. North-Holland, December 1989.

[Gar92] Hubert Garavel. The OPEN/CÆSAR Reference Manual. Rapport SPECTRE C33, Laboratoire de Génie Informatique — Institut IMAG, Grenoble, May 1992.

[GS90] Hubert Garavel and Joseph Sifakis. Compilation and Verification of LOTOS Specifications. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Proceedings of the 10th International Symposium on Protocol Specification, Testing and Verification (Ottawa, Canada)*, pages 379–394. IFIP, North-Holland, June 1990.

[GT93] Hubert Garavel and Philippe Turlier. CÆSAR.ADT : un compilateur pour les types abstraits algébriques du langage LOTOS. In Rachida Dssouli and Gregor v. Bochmann, editors, *Actes du Colloque Francophone pour l'Ingénierie des Protocoles CFIP'93 (Montréal, Canada)*, 1993.

[Gut77] J. Guttag. Abstract Data Types and the Development of Data Structures. *Communications of the ACM*, 20(6):396–404, June 1977.

[Hau92] René-Pierre Hautbois. FLAIR : une Application des Techniques Formelles en Conception. In *5th International Conference on Software Engineering and its Applications*, pages 821–836. EC2, December 1992.

[HHP87] R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. In *Proceedings of the 2nd Annual Symposium on Logic in Computer Science (LICS 87), Cornell*, 1987.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[ISO88] ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1988.

[Lam93] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, November 1993.

[LL92] Guy Leduc and Luc Léonard. A Timed LOTOS Supporting a Dense Time Domain and Including New Timed Operators. In Michel Diaz and Roland Groz, editors, *Proceedings of the 5th International Conference on Formal Description Techniques FORTE'92 (Lannion, France)*, pages 99–114. IFIP, October 1992.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[ML88] Saturnino Marchena and Gonzalo Leon. Transformation from LOTOS Specs to Galileo Nets. In Kenneth J. Turner, editor, *Proceedings of the 1st International Conference on Formal Description Techniques FORTE'88 (Stirling, Scotland)*, pages 217–230. North-Holland, September 1988.

[NS91] Xavier Nicollin and Joseph Sifakis. An Overview and Synthesis on Timed Process Algebras. In K. G. Larsen and A. Skou, editors, *Proceedings of the 3rd Workshop on Computer-Aided Verification (Aalborg, Denmark)*, volume 575 of *Lecture Notes in Computer Science*, Berlin, July 1991. Springer Verlag.

[NS94] Xavier Nicollin and Joseph Sifakis. The Algebra of Timed Processes ATP: Theory and Application. *Information and Computation*, 114(1):131–178, 1994.

[Old91] E. R. Olderog. *Nets, Terms, and Formulas*. Cambridge University Press, 1991.

[Par81] David Park. Concurrency and Automata on Infinite Sequences. In Peter Deussen, editor, *Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer Verlag, March 1981.

[Pec93] Charles Pecheur. VLib: Infinite Virtual Libraries for LOTOS. In A. Danthine, G. Leduc, and P. Wolper, editors, *Proceedings of the 13th IFIP International Workshop on Protocol Specification, Testing and Verification (Liège, Belgium)*, pages 1–16. IFIP, North-Holland, May 1993.

[PT87] Robert Paige and Robert E. Tarjan. Three Partition Refinement Algorithms. *SIAM Journal of Computing*, 16(6):973–989, December 1987.

[QFA93] J. Quemada, D. Frutos, and A. Azcorra. TIC: A TImed Calculus. *Formal Aspects of Computing*, 5(3):224–252, June 1993.

[Sin94] Amanda J. Sinton. A Safety Analysis of the Airbus A320 Braking System Design. Master's thesis, University of Stirling, Department of Computing Science and Mathematics, Scotland, UK, March 1994.

[Tau90] Dirk Taubner. *Finite Representations of CCS and TCSP Programs by Automata and Petri Nets*, volume 369 of *Lecture Notes in Computer Science*. Springer Verlag, 1990.