Static analysis of multi-threaded applications by abstract interpretation

CARLOS LEDEZMA

Master of Science Thesis
Stockholm, Sweden 2013

TRITA-ICT-EX-2013:218
KTH Royal Institute of Technology

Department of Software and Computer Systems

Degree project in Computer Science

Static analysis of multi-threaded applications by abstract interpretation

Author: Carlos Ledezma (cglr@kth.se)
Supervisor: David Delmas (david.delmas@airbus.com)

Examiner: Prof. Mats Brorsson, KTH, Sweden
There exist currently in production an immense number of applications that are considered *safety critical*, meaning that the execution of them is directly related to issues concerning the well-being of people. A domain where these applications are particularly present is in the aeronautics industry. A piece of critical software that’s embedded into an airplane’s calculator cannot, under any circumstance, fail while the aircraft is in-flight. And this restriction becomes more and more severe when the priority of the application escalates. This situation also poses an inconvenient at the moment of testing software. Since for applications to be tested on their real environment (flight test) it is necessary to have certain guarantees that it won’t fail, other methods such as unitary tests and simulations have to be used. But none of these methods are sound, meaning that if some particular case is unintentionally left out of the executions, then the behavior of the program in such scenario is not contemplated in the performed analysis. But when we are talking about *safety critical* applications, these small cases could mean a very big difference.

This is why more and more companies that produce this kind of software are starting to include in their verification process sound techniques to validate the absence of run-time errors on their programs. Particularly Airbus, one of the main aircraft manufacturers of the world, uses AstréeA, a static analyzer based on abstract interpretation, to prove that the programs embedded in their calculators cannot possibly fail.

In the following report an investigation will be presented were AstréeA was used at Airbus to prove the absence of run-time errors on the ATSU. The introductory chapter presents a description of the software analyzed, an explanation of the objectives set for the project and its scope. Then, on chapter 2 all the necessary theoretical concepts will be presented. Sections 2.1 - 2.3 give an overview of the basics of abstract interpretation, while section 2.4 presents the analyzer used. Then chapters 3 and 4 describe in depth the solution given and how the investigation was carried out. Finally chapters 5 and 6 enter into the presentation and analysis of the results obtained in the period of study and the current state of the solution.
Acknowledgment

There are way too many people I’m thankful to for supporting me in this project and, in general, during my whole career. Not to subtract importance from them, but to exalt the relevance of the following, I’ll only mention three persons:

My father, Carlos Ledezma,
my mother, Cecilia Rondón and
my brother, Carlos Alberto Ledezma.

You have and will always be there for me.

Toulouse, August 2nd, 2013.

Carlos Guillermo Ledezma
5. Results

5.1. Summary of the alarms .......................... 37
5.2. Suppositions for correctness .......................... 40
5.3. Recommendations to ENS .......................... 42
5.4. Recommendations to Airbus .......................... 43
5.5. Resources produced .......................... 43

6. Conclusions and recommendations .......................... 45

6.1. Conclusions .................................. 45
6.2. Recommendations for future work .......................... 46

A. Examples of some implemented stubs .......................... 51

B. LynxOS 2.4 stub library .......................... 57

C. Log search examples .......................... 59
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Structure of the ATSU</td>
<td>6</td>
</tr>
<tr>
<td>2.1</td>
<td>Galois connection between partial trace semantics and reflexive transitive</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>closure</td>
<td></td>
</tr>
<tr>
<td>2.2</td>
<td>The widening operator for the interval domain</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>The narrowing operator for the interval domain</td>
<td>12</td>
</tr>
<tr>
<td>2.4</td>
<td>Example code for intervals abstract domain</td>
<td>12</td>
</tr>
<tr>
<td>2.5</td>
<td>Example code for decision trees abstract domain</td>
<td>15</td>
</tr>
<tr>
<td>3.1</td>
<td>Simulation of the reception of a signal in a stub</td>
<td>24</td>
</tr>
<tr>
<td>3.2</td>
<td>Example code for partitioning I/O</td>
<td>25</td>
</tr>
<tr>
<td>3.3</td>
<td>The prologue and epilogue to a partitioned I/O</td>
<td>26</td>
</tr>
<tr>
<td>3.4</td>
<td>Example of the utility of interval division (with alarm)</td>
<td>27</td>
</tr>
<tr>
<td>3.5</td>
<td>Example of the utility of interval division (no alarm)</td>
<td>27</td>
</tr>
<tr>
<td>3.6</td>
<td>Initial approach to file stubs</td>
<td>27</td>
</tr>
<tr>
<td>3.7</td>
<td>A solution that allows different unrolling on the same array</td>
<td>28</td>
</tr>
<tr>
<td>4.1</td>
<td>Example of a stub for strlen</td>
<td>30</td>
</tr>
<tr>
<td>4.2</td>
<td>The support for several processes</td>
<td>31</td>
</tr>
<tr>
<td>4.3</td>
<td>Structures used to implement the stubs for file I/O</td>
<td>32</td>
</tr>
<tr>
<td>4.4</td>
<td>Structure used to represent shared memory in the stubs</td>
<td>33</td>
</tr>
<tr>
<td>4.5</td>
<td>Specification of an absolute address</td>
<td>34</td>
</tr>
<tr>
<td>5.1</td>
<td>Command line arguments for the analyses</td>
<td>38</td>
</tr>
<tr>
<td>5.2</td>
<td>Reasons for the invalid pointers to functions</td>
<td>39</td>
</tr>
<tr>
<td>5.3</td>
<td>Assumptions of names and values for environment variables</td>
<td>41</td>
</tr>
<tr>
<td>6.1</td>
<td>Two ways of implementing a condition on a while loop</td>
<td>45</td>
</tr>
</tbody>
</table>
# Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOC</td>
<td>Airline operational control.</td>
</tr>
<tr>
<td>ATC</td>
<td>Air traffic control.</td>
</tr>
<tr>
<td>Astrée</td>
<td>Real-time embedded software static analyzer (For its initials in French).</td>
</tr>
<tr>
<td>AstréeA</td>
<td>Real-time asynchronous embedded software static analyzer (For its initials in French).</td>
</tr>
<tr>
<td>ATSU</td>
<td>Air traffic services unit.</td>
</tr>
<tr>
<td>DCDU</td>
<td>Datalink control and display unit.</td>
</tr>
<tr>
<td>ÉNS</td>
<td>École Normale Supérieure.</td>
</tr>
<tr>
<td>FIFO</td>
<td>First in first out.</td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output.</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of electrical and electronics engineers.</td>
</tr>
<tr>
<td>IPC</td>
<td>Inter process communication.</td>
</tr>
<tr>
<td>MCDU</td>
<td>Multifunction control display unit.</td>
</tr>
<tr>
<td>OS</td>
<td>Operating system.</td>
</tr>
<tr>
<td>PCB</td>
<td>Process control block.</td>
</tr>
<tr>
<td>POSIX</td>
<td>Portable operating system interface.</td>
</tr>
<tr>
<td>RAM</td>
<td>Random access memory.</td>
</tr>
<tr>
<td>SA</td>
<td>Host platform (For its initials in French).</td>
</tr>
</tbody>
</table>
Chapter 1.

Introduction

Today, there exists an amazingly big number of applications meant to do verification of a program’s execution behavior and notify if any bugs can potentially happen. Nonetheless, most of these applications use informal methods in order to do this verification. This means that they can indeed prove that under some conditions the program may fail, but those analyses are not sound, in the sense that their answer could lead to what is called a false negative. In these cases the analyzer would state that a program is bug free, when this is not actually true. The reason why these programs present this kind of behavior is because they are not based on any mathematical foundation that can prove that their analysis covers all possible cases.

Now, there are certain areas of the software development industry where a malfunction of software can have catastrophic effects, resulting in the loss of gigantic amounts of money and, more importantly, in the loss of human lives. One of these areas is, for example, the aeronautic industry. Here, in particular, software is embedded into the main control systems of airplanes, helicopters and even space ships. The safety critical systems developed in these companies cannot afford to have malfunctions, and for instance the researchers in this area have developed applications based on formal methods, that can soundly prove the absence of run-time errors in software of particular families. The mathematical foundation behind these software is called abstract interpretation, a theory that is thoroughly discussed in chapter 2.

One of such applications is the static analyzer called AstréeA, which applies the theory of program semantics and abstract domains to prove that a determined program is free of run-time errors (integer/float overflows, access out-of-bounds, implicit casting, etc.). It was made, in particular, to perform analyses over applications running on a real-time scheduling OS with fixed priorities and on a mono-processor architecture, which are very common in the avionic world.

AstréeA, as was just commented, has its bases on the theory of abstract interpretation, and it uses the full power of the theorems derived from it to perform it’s proofs. But it also counts on another, less mathematical, but equally important resource: a library of stubs. These are an abstract representation of the calls to the standard C library that the analyzed program performs. They are developed using the conformance documents of the OS that the application will use when run, in order to determine the correct behavior of the calls, and a series of primitives that AstréeA provides, that allow the user to manipulate the abstract domains calculated by the analyzer. These stubs are, clearly, not executable, since a regular compiler wouldn’t be able to interpret the directives introduced to manipulate the abstract domains. But AstréeA will indeed be able to process this code and perform the appropriate analysis actions. A more thorough explanation of what a stub is and how they were developed for this investigation is given on chapter 4.
The presence of a correctly implemented library of *stubs* could mean the difference between a correct and an erroneous analysis. This is because this model is the only way the analyzer can simulate the environment in which the software analyzed will be run. And of course, the development of these calls also implies the development of the appropriate structures that simulate the effects of the function calls. All of this poses a big challenge for the developer that has to set up the analysis, since the representations found have to be correct, but they also have to be efficient enough so the analyses can run in decent time.

The functionality of this analyzer is explained in more detail in section 2.4. A more detailed explanation of the development of the *stubs*, as previously mentioned, is contained in chapter 4.

1.1. Problem statement

1.1.1. Target system

The Air Traffic Services Unit (ATSU) is a subset of the embedded software that is included in the A320, A330 and A340 aircrafts produced by Airbus. This particular program is in charge of handling all the communications that occur between the airplane and the land operators that have all the control over the air traffic [26]. The ATSU’s development started in 1995, and has been since then a long-term development project at Airbus. It was certified by the European Joint Airworthiness Authorities in 1998 for A330 and A340 aircrafts, and in 1999 for the A320 [2]. It has been since then operating on the airplanes for major airlines such as Lufthansa, Air France and Emirates.

The system is based on the concept of a “Host platform”, which provides services to different external applications, and serves to mask the hardware elements that are manipulated by the ATSU. This host runs on a POSIX system which offers multi-tasking. It is partitioned into three main components [2]:

**Boot software:** Part of the hardware.

**Aircraft interface software:** It’s loadable software that provides the applications with a high level interface. It contains several functionalities but in particular it provides the following 4 services:

- **Air/Ground communication services.** Provides the access to communications sub-networks, management and status control of air/ground communications and routing of messages between the on-board peripherals and the ground stations.

- **Human machine interface services.** Manages the communications between the applications and the peripherals, and the MCDU, DCDU and PRINTER protocols.

- **On-board peripherals services.** Handles the communication exchanges between the peripherals.

- **System management services.** Receives and broadcasts the data to and from the peripherals, ensures the downloading and uploading of functions and manages power supply cuts.
1.1. Problem statement

Datalink applications software: Includes the Airline Operational Control (AOC) and the Air Traffic Control (ATC) applications. The first are applications dedicated to the communication between the aircraft and the airline facility. The second deal with the communication between the aircraft and the air traffic control centers. These applications are provided by third parties, but include important functionalities such as fuel calculations, extraction and storage of aircraft performance data and location services.

Apart from that structure, the ATSU provides an ensemble of modules that give the mentioned masking of the communication between the applications and the low level details. This software is called “Structure d’accueil” (SA), and is embedded in the hardware. It is composed of several different processes [3]:

GEST. Is the main process of the ATSU, and is in charge of setting up the execution environment and starting all the other processes of the system. After the initialization is over, this process keeps control and handles all the errors that can occur.

ACEM. Handles the acquisition and emission of labels, which are the parameters sent by the applications and the different modules.

PROT. Deals with the airplane’s peripherals’ access protocols.

ARF. Serves as router between the applications on the plane and the ground communications.

IHM. Allows the handling of the screens on the airplane (MCDU, DCDU).

ARF-MANAGER. Interface to handle the parameters of the router function of the ATSU.

TELE. Allows the download of applications and databases.

BITE. Deals with memorization and visualization of errors.

All the modules of the ATSU are written in C, using the parallelism defined by the POSIX standard. The application is then run on a calculator with LynxOS 2.4 as operating system. Since by the time LynxOS 2.4 was created the current POSIX standard was not valid, the ATSU also counts with a module that implements the used POSIX calls by the means of the respective system provided functions. This module is called the “BIB_COMMUNE”. The full structure of the ATSU can be seen on figure 1.1.

The ATSU was partly coded by hand and partly generated automatically using code generation tools.

1.1.2. Problem definition

It is now in the interest of Airbus to use AstréeA to prove the absence of run-time errors on the SA. This analysis shall be done in a modular fashion, by first proving the absence of run-time errors on each process and then performing an analysis that takes into account the interactions between the different modules.

So, the investigation presented in this report addressed the issue of starting with this analysis process, and defining the bases so in the future a full analysis can be achieved.

The necessary stubs to model the calls to LynxOS 2.4, both the ones that are provided by the operating system and the ones that have been provided to comply with the POSIX standard, were developed. Then a static analysis by abstract interpretation was made, which was tuned in order to avoid the false alarms that the analyzer gave. Any relationship that the analyzer
1. Introduction

The analysis started from the IHM module, and has left a scalable solution that can later be expanded in order to perform a full analysis of the whole SA.

1.1.3. General objective

The investigation aims at experimenting with AstréeA to prove the absence of run-time errors on the SA, a multi-process, multi-threaded application developed by Airbus under the POSIX and C99 standards. The conditions implemented by AstréeA that are too restrictive or too coarse, have to be found in order to propose modifications over them. Also, conditions that are not deduced by the analyzer have to be manually established. This way an analysis can be defined which can prove the absence of run-time errors on the target application, throwing the minimum number of false alarms possible.

1.1.4. Specific objectives

1. Familiarize with the scientific foundations of static analysis by abstract interpretation and the applicable standards: C99, POSIX.
2. Make experiments of static analysis on avionics software using AstréeA.
3. Develop the stubs that are strictly necessary in order to perform an analysis of the application.
4. By iterative refinement, achieve a precise analysis.
5. Characterize the invariants of false alarms that cannot be cleared by the analysis and propose refinements to AstréeA that would allow these proofs.
6. If a particular invariant cannot be expressed in AstréeA, due to the limitations imposed by the analyzer, propose the ENS the appropriate modifications that would allow a more precise analysis.

Figure 1.1.: Structure of the ATSU
1.1. Problem statement

7. If time and current state of the analysis allow it, update the POSIX model in order to get better precision on the analysis, reinforce politics on system resource usage and extend the analysis to more modules of the SA.

1.1.5. Scope of the solution

Due to the early stage at which this project was taken, requiring a full analysis of the SA and reaching a zero false alarm goal were completely unrealistic objectives. It is because of this that this project was limited to performing an analysis of the IHM module. The choice of this module was due to the fact that it is one of the smallest pieces of the ATSU. Its size and relatively lower complexity could guarantee that complete analyses could be made and results could be seen throughout the investigation.

But also, due to the complexity and the size of the IHM process (4 threads composing nearly 11,000 lines of code [12]), it would have been unrealistic to propose reaching a zero false alarm goal on the analysis of this module. So, the scope of this analysis reaches trying to prove the absence of run-time errors on the IHM module, having as few false alarms as possible.

On the other hand, it is part of the proposed solution the fully developed necessary stubs to make a complete analysis of the code of IHM. That is, all possible system calls made by the module have been stubbed with some appropriate level of abstraction to perform an analysis.

1.1.6. Contributions

Thanks to the project that was made, Airbus now counts with a library of stubs of calls to LynxOS 2.4 (extended with some additional POSIX compliant functions) sufficiently large to perform a full static analysis of the IHM module of the SA using AstréeA. This is a very important step in the evolution of an analysis of the SA, since this library can easily be extended to include some other system calls that may be particular to other modules. The development of the stubs was a troublesome and tricky labor, which took a considerable amount of time and dedication. It is then a good starting point for future continuations of this project.

Apart from that, the analyses themselves resulted in several recommendations to the ÉNS that improved the functionality of the analyzer, allowing more precision on certain fields that were way too general.

For Airbus in particular, several recommendations were made on some strange program behaviors which were discovered due to the presence of false alarms. These were notified to the correct instances, which will in due time perform the appropriate modifications to the code of the IHM. This way, Airbus shall now count with software of better quality, and that can be more easily certified using static analysis.

For the rest of the community that uses AstréeA or similar static analyzers in order to perform validation of software, this investigation gives them an important amount of documentation on methods for proving correctness of code and clearing false alarms. It also provides some examples on how stubs can be developed in order to model particular structures of the operating system in ways that can get precise analyses in acceptable execution times.
1.2. Related work

In the area of abstract interpretation there is a lot of work being done, in particular at the École
Normale Supérieure in Paris, France.

If some further research is to be done on this topic by a beginner in abstract interpretation, it
would be convenient to start by [7] and [9]. These deal with the most basic concepts and provide
explanations that are easily assimilated.

Afterwards, a more complete work on all the concepts associated to static analysis of avionics
and aerospace software can be found on [4].

Some more advanced theory on abstract interpretation, involving multi-threading, memory
management and real-time systems, and abstractions that can be used to handle all these are
present in [16], [17] and [18].

The more practical, industry oriented approaches to abstract interpretation, which directly involve
real life examples of enterprises that are using abstract interpretation to prove the correctness of
their software, are explained in [10], [11] and [24].
Chapter 2.
Background theory

2.1. Abstract interpretation

Abstract interpretation is “a theory of approximation of mathematical structures, in particular those involved in the semantic models of computer systems”[9]. The whole concept is based on the fact that one can create abstractions of program executions by defining mathematical structures called semantics. These semantics have to be precise enough so the whole program can be described, but sufficiently coarse so they can be represented by a computer and calculated in reasonable time.

Now, since the problem of finding all possible program executions is non-decidable, then it is never possible to find a computable semantics that will actually retain all the properties of every program. Also, the most precise semantics become sometimes in-computable, so it becomes necessary to create further abstractions of these, by defining a more coarse semantics, and demonstrating that anything that is proved in the new, more abstract case, is still true in the more concrete case. This proof can be made by showing that there exists a Galois connection between the 2 semantics [9].

Nonetheless, by abstracting away a semantics, since some properties are lost, the resulting analysis becomes sound, but not complete. This means that the analysis can result in false alarms, where the analyzer says that there may be a bug, when due to the actual execution contexts this is impossible.

The following two examples, and further explanation of how one is an abstraction of the other, will serve to make this theory more clear. A much larger set of semantics can be found on [4].

Partial trace semantics: is the most basic way of representing a program. This is done by building concatenations $s_0s_1\ldots s_n$, where each $s_i$ represents a program state, and where it is satisfied that $s_is_{i+1}$ is a valid transition of the program. The set of all valid traces built in that manner is called the collecting semantics [9], and it expresses all possible program executions. Now, the calculation of this set is a non-decidable problem, which means it is necessary to find a more general set.

Reflexive transitive closure: consists of all possible pairs $(s_0,s_n)$ such that there exists a valid trace that starts at $s_0$ and ends at $s_n$. This new semantics can be calculated more easily, but at the cost of some loss of information. Note that with it all reachable states of the program can be determined, but it is impossible to say which are the intermediate states.
2. Background theory

To prove that effectively the reflexive transitive closure is a correct abstraction of the partial trace semantics, the two functions on figure 2.1 are defined [9].

\[ \alpha^*(X) = \{(s_0, s_n) | (s_0s_1...s_n) \in X \} \]

Where \( X = \) partial trace semantics.

\[ \gamma^*(Y) = \{s_0s_1...s_n | (s_0, s_n) \in Y \} \]

Where \( Y = \) reflexive transitive closure.

Figure 2.1.: Galois connection between partial trace semantics and reflexive transitive closure

It can be seen that effectively function \( \alpha^* \) applied to the set of possible partial traces of a program will yield by definition the reflexive transitive closure of these traces. On the other side, function \( \gamma^* \) will calculate, for a given set of initial and final states, all possible partial traces that could generate each given pair. So this function over-approximates the partial trace semantics [9]: \( X \subseteq \gamma^*(\alpha^*(X)) \).

This relationship is what is called a Galois connection, which was mentioned before.

Having proved the existence of this relationship, it can be then guaranteed that every question that is answered in the concrete partial trace semantics can be answered in the reflexive transitive closure, and so the later represents a sound abstraction of the former.

2.2. Fix-point notation

In mathematics, a fix-point of a function is a point (from the domain of the function), that is mapped to itself.

This concept is very useful when developing programs that perform abstract interpretation, since they provide, as will be seen in sections 2.2.1 and 2.2.2, a mathematical way of approximating the values that a specific element will take on a particular domain, without actually having to perform all the calculations.

In this sense, it has been proved that semantics can be expressed as fix-points [8]. And in fact, when designing an abstraction for a particular domain, one of the key problems is to find the function whose least fix-point is the desired semantics. The existence of this representation allows to calculate semantics iteratively, reducing considerably the computation costs of it [20].

A clear example of this is the reflexive transitive closure explained in section 2.1. This semantics can be calculated as the smallest solution to the equation \( X = 1_S \cup (t \circ X) \). Where \( 1_S = \{(s, s) | s \in S \} \) is the identity on \( S \), and \( \circ \) is the composition of relations [4].

2.2.1. Widening

As explained before, having a semantics expressed in its fix-point notation allows to calculate its value iteratively by starting at the initial value of the variable and then performing the modifications indicated by the program until a fix-point is reached. This is indeed an effective way to perform the calculations, but intuitively one can deduce that for variables that are
modified a lot (like in a very long while loop), this process can take a considerably long time.

So an idea to accelerate convergence is to use an operator called widening \[4\].

This will imply solving, instead of \(X = F(X)\), the equation \(X = X \nabla F(X)\), which uses 2 iterates to extrapolate the next value of the equation. This extrapolation should over-approximate the values, in order to guarantee the soundness of the method, and should enforce convergence, to guarantee the termination of the method.

An example of widening will be presented on section 2.3.

Now, since the operator actually over-approximates the values of the semantics, then the result is not as precise as one would wish, and sometimes it’s too imprecise. That is why another operator is introduced, that will try to reduce as much as possible the imprecision generated by the widening. This operator is called narrowing.

### 2.2.2. Narrowing

Also called “downwards iterations” \[4\], consists on going back in the iterations of the widening while a correct semantics can be guaranteed. Again, as with the previous operator, this new one has to ensure soundness and termination.

Both the widening and narrowing operators are used by AstréeA when deducing the bounds of variables modified inside iterations. As will be seen further ahead this is sometimes good, since it brings a lot of speed into the analysis, but can be a problem when precision is needed on the iterates.

### 2.3. Abstract domains of interest

Now, it will be of interest to expand the information presented on abstract domains. It is necessary, in order to get a more clear view of the results presented further ahead, and the reasons why the chosen methods were used, to have a more thorough knowledge of certain abstractions implemented in AstréeA.

#### 2.3.1. Intervals

The interval domain maintains an upper and lower bound on every integer and floating-point variable used by the program \[4\]. When a particular variable is modified, its representation on this domain can be easily updated by performing the appropriate interval operation on the already deduced bounds for the value.

So, for example, when an arithmetic operation is performed on the variable, this operation is repeated on both bounds of the interval. Also, if two execution traces are joined at some point in the program, then the intervals representing a particular variable on each of those traces are joined using the union of intervals.

It is a very simple domain, but it gives a lot of useful information with which one can deduce different kinds of errors such as divisions by zero or out of bounds accesses.
This domain is very attractive since it has a very low memory and time cost per abstract operation. It is also commonly used because it’s widening and narrowing operators are rather simple, and can help solve very long loops in only a few iterations. These operators can be defined as seen on figures 2.2 and 2.3 [9]:

\[
[l, h][\nabla](l', h') = [a, b] \text{ such that:}
\]

\[
a = \begin{cases} 
\max\{r_i | r_i \leq l'\} & \text{if } l' < l \\
\text{l} & \text{otherwise}
\end{cases}
\]

\[
b = \begin{cases} 
\min\{r_i | h' \leq r_i\} & \text{if } h < h' \\
\text{h} & \text{otherwise}
\end{cases}
\]

Where \(-\infty = r_0 < r_1 < \ldots < r_n = \infty\) is a chosen finite ramp

Figure 2.2.: The widening operator for the interval domain

\[
[l, h][\Delta](l', h') = [a, b] \text{ such that:}
\]

\[
a = \begin{cases} 
\text{l'} & \text{if } (\exists i | l = r_i) \\
\text{l} & \text{otherwise}
\end{cases}
\]

\[
b = \begin{cases} 
\text{h'} & \text{if } (\exists j | h = r_j) \\
\text{h} & \text{otherwise}
\end{cases}
\]

Where \(-\infty = r_0 < r_1 < \ldots < r_n = \infty\) is a chosen finite ramp

Figure 2.3.: The narrowing operator for the interval domain

In order to get a better idea of how the interval domain and it’s iteration accelerators work, a small analysis will be made of the code presented in figure 2.4

```pseudocode
1 int x = 1;
2 int res;
3 while(x < 100)
4 { 
5     if(random_decision) break;
6     ++ x;
7 }
8 res = 1 / x;
```

Figure 2.4.: Example code for intervals abstract domain

It would be desirable to prove that the assignment to variable res on line 9 will not cause a division by zero without having to run the code.

As it was explained in section 2.2 every abstraction of a program can be defined as the least fix-point of a function. In order to get, then, the semantics value for x after the execution of the loop and until the assignment, it is necessary to deduce the function whose least fix-point represents these iterations. The logical procedure to do so is as follows.
2.3. Abstract domains of interest

When this code is analyzed it can be easily deduced that after the first instruction the value of \( x \) will be 1. But then, the program enters an iteration that performs an assignment depending on a random condition, and does this until another well established condition is satisfied.

So, when the loop is entered for the first time, it can be assured that \( x = \{1\} \), a singleton interval. When the flow reaches the \texttt{if} statement, a partition of the execution traces is produced, one trace follows the path where the condition is true, and the other where it is false. On the first case, nothing happens, and the variable keeps the same value. On the second case though, the variable gets incremented by one, so by performing this operation on both bounds of the interval, the resulting value is \( x = \{2\} \).

At the end of the first iteration then, there are two traces. As it is the most practical thing to do, these two traces will be merged into one again, in order to keep the analysis simple. To merge the values, an interval union is done, leading to the new value \( x = [1, 2] \). And since the set of values for \( x \) has changed, then a fix-point has not been reached, so another iteration has to be done.

On the second loop, a similar effect occurs, just that this time the first trace (where the condition is true), will have the value \( x = [1, 2] \) and the second trace will have the value \( x = [2, 3] \), due to the increment on both bounds. When the union of the two flows is done, the third iteration will have the value \( x = [1, 3] \).

Intuitively, but also formally proved by [9], it can be seen that the function that defines each iterate in base of the previous one (using the interval notation) is:

\[
F([l, h]) = \{1\} \cup [l + 1; \min(99, h + 1)]
\]

In order to get the bounds on the variable it is enough to find the least fix-point of this equation, since this will indeed represent the semantics of the program, as has been mentioned before. It is easy to deduce that this will be done until \( x = [1, 99] \), which will indeed be a fix-point for the iteration.

Having this approximation, it can be proved that the assignment \( res = 1/var \) will not produce a division by 0, since 0 is not included in the deduced values for \( var \). And it can be guaranteed that this analysis is sound because the interval domain is an abstraction of the trace semantics [9], which is a concrete representation of a program.

Now, some quick math tells that this process will perform the same analysis 99 times. And even though at the end of the loop a correct approximation for \( x \) will be given, if the loop would have more iterations, or perform much more operations, then these repetitive analyses might become a burden in execution times. Note that in particular if an infinite loop is analyzed, then this method will never finish. This is why it becomes necessary to use \textit{widening}.

As explained before, \textit{widening} implies using two iterates of the function in order to perform the calculation of the next iterate, and have a quicker convergence to the approximation of the final value. This will now be applied to the example.

Just as in the last time, at the end of the first loop the value of the variable is \( x = [1, 2] \). This result can be used, along with the value before the iteration, to perform a \textit{widening} step as follows:
2. Background theory

Let $-\infty < -1 < 0 < 1 < \infty$ a finite ramp

$$X_1 = X_0 \nabla F(X_0)$$

(Definition of widening)

$$= \{1\} \nabla [1, 2]$$

(Interval widening (2.2))

$$= [1, \infty]$$

Since the bound didn’t stabilize, the process must be repeated. By analyzing the previous steps, it can be easily deduced that the result of joining the traces after the second iteration will be $x = [1, \infty]$. This means that when performing the widening the result will be the same. Having reached a fix-point, it can then be established that this will be the value of $x$ after the loop.

Of course, this is a very over-approximated value, so let’s exemplify now how this value can be made more precise by using narrowing. So, on a first step the following is obtained:

Let $-\infty < -1 < 0 < 1 < \infty$ a finite ramp

$$Y_1 = Y_0 \Delta F(Y_0)$$

(Definition of narrowing)

$$= [1, \infty] \Delta [1, 99]$$

(Interval narrowing (2.3))

$$= [1, 99]$$

And this is a fix-point for the narrowing since $F([1, 99]) = [1, 99]$, which means the downwards iterations stop.

As it can be seen, by applying these mathematical concepts, a process that took 99 iterations to reach a conclusion, was reduced to only 2 widening steps and 2 narrowing steps. Even more so, independently of the size of the iteration, this second method will take the same 4 steps to calculate the bounds of the variable.

It is important to note here that this is only a demonstrative example of how widening and narrowing work. In this case AstréeA would perform an intersection between the loop condition $([-\infty, 99])$ and the widening conclusion $([1, \infty])$ to arrive to the same result but saving the extra operations required for narrowing.

Of course, since these steps actually calculate over-approximations, they are not always as precise as one would need. For example, if the random condition was not there on the code presented in figure 2.4 then the regular method would have been able to deduce that at the end of the loop $x = \{99\}$, while the iteration acceleration would have reached the same conclusion as with the conditional. These imprecisions are sometimes tolerable, but when not, additional measures have to be taken in order to avoid these generalizations.

2.3.2. Decision trees

The abstract values in this semantics consist of binary trees, which contain numeric invariants at the leaves and boolean variables at the nodes. This way a program execution can be partitioned into two cases depending on the value taken by the boolean [4].

These are particularly useful when certain program information is stored in boolean variables, and it is of interest to deduce properties that will be held depending on whether the booleans are true or false.

In order to have a better example of what this domain deals with, an analysis will be made of the code seen on figure 2.5 [1].
2.3. Abstract domains of interest

It is the objective here to prove that the assignment on variable \( Y \) cannot produce a division by 0, and thus a run-time error.

If this analysis is made using only the interval domains, as it was done on the previous example, then at the point of the assignment the only information one would have would be: \( B = \{ 0 \} \), since that is the guard of the condition; and \( Y = X = [0, \text{MAX}_\text{UINT}] \), due to the fact that neither of these variables has been initialized. So, under these conditions, since the value 0 is contained in the possible set of values of \( X \), an analysis would raise an alarm indicating the possibility of the error. But due to the way the value to \( B \) is assigned, this behavior is an impossible case.

Wanting to clear the false alarm then, it is necessary to have an abstract domain that establishes the relationship between the boolean variable \( B \) and the numerical value of \( X \). So, a boolean tree is created in order to establish this relationship. This abstract domain will create a binary tree which will establish the following (after the assignment \( B = (X == 0) \))\[1\]:

\[
\begin{align*}
\text{if } B \text{ then } X \in \{0\} \\
\text{else } X \in [1, \text{MAX}_\text{UINT}] \text{ and is not } 0
\end{align*}
\]

It is evident to see how this condition is represented in a binary decision tree where \( B \) is located on the node, and the values for \( X \) are on each of the leaves. And so, having established this, the analysis can be sure that the assignment of line 10 cannot lead to an error, since it is known that \( B \) is false.

2.3.3. Interleaving

The abstract domains presented previously work very well when analyzing a sequential succession of instructions. But it is evident that if we include the possibility of having multiple threads modifying a particular variable, then these abstractions become insufficient, because they don’t allow to represent the interactions between the different executing threads.

The interleaving semantics is the most natural model of a parallel execution. This is, it considers all possible interleaving of control paths from all threads in the program \[17\]. These correspond to all sequentially consistent executions.
2.3.4. Interference

Since considering all possible interleaving of thread executions can result in very long execution times for the analyzer (since it reasons on infinite sets of paths), then [17] proposes this new, more abstract, semantics.

The non-parallel semantics are enriched by the notion of intersection, which will denote the values that can be set to a particular variable by a particular thread. By doing this, the analyzer can calculate the effects that the execution of each thread will have over the others. So, the analysis will calculate the set of intersections over and over again, until it reaches a fix-point [10]. A much better explanation of how the analyzer performs these calculations is given on section 2.4.1.

These intersections give a much quicker way of converging to the possible values of the shared variables, but they have the inconvenient that they abstract away the relationships between the variables, therefore losing important precision.

2.4. The AstréeA analyzer

The Analyseur statique de logiciels temps-réel asynchrones embarqués (real-time asynchronous embedded software static analyzer), AstréeA, is a static analyzer based on abstract interpretation that addresses the issue of proving the absence of run-time errors on “multi-threaded embedded critical C software” [16]. It is currently developed by the École Normale Supérieure (ENS), based on Astrée, a static analyzer of the same kind; but without support for parallelism.

Even though AstréeA is currently on development phase, it’s predecessor, Astrée, is already a commercial software currently distributed by AbsInt, a company that “provides advanced development tools for embedded systems, and tools for validation, verification and certification of safety-critical software” [24].

Both Astrée and AstréeA are capable of analyzing structured C code, that meets certain restrictions that will be further discussed in section 2.4.2. They target embedded applications, as found particularly in the aeronautics industry [24].

2.4.1. Mode of operation

In order to have a correct understanding of how AstréeA performs an analysis, it is first useful to know how Astrée does it.

The Astrée analyzer has to be invoked on a piece of preprocessed C code, for which an entry point is defined. By default, as in most C compilers, this entry point is set to be the main function, but this value can be overridden. When this is set up the analyzer will go through the code, performing on each instruction the appropriate operations on the abstract domains activated for the analysis. These transformations greatly vary depending on each domain, but some general explanations about them, on reduced languages, can be found on [18]. The analyzer will then emit an alarm every time it reaches an instruction that would imply performing an invalid operation. These behaviors are deduced by looking at the sets of values calculated so far for the variables involved in the instruction, and making sure that none of them would cause an error. The type of alarm emitted will depend on the type of error encountered.
2.4. The AstréeA analyzer

If after encountering an alarm Astrée can deduce any possible traces that might still be valid, then it will notify the alarm and go on with the analysis of these other traces. If every trace would lead to a run-time error, then the analysis stops and emits an error message [1].

Currently Astrée can deduce the following run-time errors [1]:

**Errors terminating the execution**, such as floating-point exceptions.

**Errors not terminating the execution with predictable outcome**, such as overflows over signed integers.

**Errors not terminating the execution with unpredictable results**, such as memory corruptions.

Of course, since the analyzer uses abstractions, which represent an over-approximation of the sets of possible values for variables, it is possible that a particular emitted alarm represents a run-time error that, under the actual execution traces of the program, is impossible. But Astrée will still notify it, since in its abstraction, this error is possible. This is what’s called a *false alarm*, and the main objective while doing a static analysis is to reach an analysis as coarse as possible (for time sake) but that doesn’t throw any *false alarms*.

Having a better understanding of Astrée, understanding the behavior of AstréeA should be easier. What this second version of the analyzer does is to perform for each thread a sequential analysis, just as if it were using Astrée. The major difference is that this analysis also calculates the interferences that the thread has on the execution of the other threads.

After each thread is individually analyzed, all interferences are merged. If the set of interferences didn’t change from the previous execution, then a fix-point has been reached, and the analysis can stop. But if the set of interferences has indeed changed, then the threads are individually analyzed once again, but this time taking into account the new sets of interferences [10].

AstréeA also keeps track of which variables are accessed after having locked which synchronization structures, allowing it to deduce also possible *data races*. Unfortunately this is currently the only parallelism associated error that the analyzer can deduce. Other complications like *deadlocks*, *livelocks* or *priority inversions* are not yet detected.

For performing the analysis, AstréeA currently supports, among others, the traces, intervals, octagons, decision trees and pointers abstract domains [1].

Apart from that, AstréeA needs to be provided with a library of *stubs*. These are an implementation of the system and library calls performed by the analyzed application, but written using AstréeA directives. This means that the resulting functions are not actually executable, but serve to provide the analyzer with sufficient semantic information to know how the calls modify the parameters, and how subsequent calls on similar or related parameters would react [19].

2.4.2. Limitations

Even though the analyzer is very complete, it still has several limitations imposed by the abstract domains used. These limitations imply that sometimes it is harder to get rid of alarms, and that sometimes decisions have to be taken that make the analysis much slower.
Among the most serious limitations found during the investigation there are:

**Boolean decision trees** currently support on leaf nodes only numerical abstractions. This implies that, for example, if depending on a boolean condition a pointer can be NULL or point to a certain value, then the analyzer will not be able to deduce this relationship, even if the pointer is included in a boolean decision tree. This example in particular is very common on the target application, and this limitation implied the use of more complicated methods to allow the analyzer to deduce these types of relationships.

**Strictly static information.** AstréeA has not implemented any mechanisms to handle recursion or dynamic memory allocation, which means that everything that is used by the analyzer has to be preallocated before the analysis. The particular issue of recursion didn’t pose a problem, since in the development of avionics software recursion is forbidden. But the lack of handling of dynamic memory made particularly complicated the creation of *stubs* that handled inherently dynamic structures, such as files or shared memory.

**Not adapted for multi-processing.** Even though the analyzer gives full support for all the structures associated with multi-threading operations, it has not yet an implemented environment for several processes. This implied that part of the time had to be dedicated to develop simulations of multi-processing, since the structures aren’t actually present on the analyzer. Among these structures was included, for example, a PCB where the file descriptors of a process could be stored.

**Non-relational, flow insensitive** parallel analysis, which rends several alarms that are very hard to clear, since they arise due to the lack of information on the relationships that exist between the different shared variables. These relationships cannot be directly established, and so alternate methods had to be used.

### 2.4.3. Directives of interest

One of the most attractive features of AstréeA is that it provides a series of directives that can be inserted into the code, and that tell the analyzer to perform specific actions at particular points of execution. These directives include functionality from printing information to enriching the deduced semantics.

A complete list of the directives for Astrée (which are all present on AstréeA) can be found on [1]. Some directives that are particular to AstréeA can be found on [19].

From those two sources the following are some of the directives that were of particular importance for the analysis made. Since they are analyzer particular calls, all these directives are prepended with `__ASTREE__`, which will be omitted here for the sake of space.

- **log_vars((V_1,\ldots,V_n))**. At the point of the analysis where this directive is found, the analyzer will perform a print on the log where it will show the state of the abstract domains for the variables included in the list passed as parameter. This is certainly the most useful debugging technique.

- **partition_control**. Is written before a control structure (for, if, while), and tells the analyzer to perform, from the point where the directive appears and on, a separate analysis for each of the values of the variable that determines the control flow. This is, either the condition variable, or the iteration’s iterate.
2.4. The AstréeA analyzer

The analyzer does this partitioning by default, but it joins all the generated traces just after the control structure. By placing the directive, AstréeA will keep the analysis partitioned until notified to join the traces.

AstréeA also joins partitions automatically right after a function’s execution has ended, losing outside of the function any precision obtained by separating the traces inside the call. Nonetheless, if this directive is placed before the call, AstréeA will maintain the partitioning on the caller.

**partition_begin((V)).** Also performs a partitioning over execution traces, but this time it does so over the possible values of the variable indicated as parameter.

**partition_merge_last(()).** Ends with the last initiated partition, by combining all the invariants deduced for the partitioned value. This is used when the partition is no longer necessary, since no more alarms will come out of the generalization. It is necessary to place these directives in the proper places, otherwise the analysis will be heavily deteriorated in execution time.

**boolean_pack((V₀,...,Vₙ;B₀,...,Bₙ)).** Even though the analyzer counts with heuristic methods to define when to make boolean decision trees, generally if it is wished to include a set of variables in this domain, it will be necessary to explicitly tell the analyzer to do so. When this directive is present, the analyzer will create a decision tree using variables V₀,...,Vₙ as leaves and variables B₀,...,Bₙ as inner nodes.

**absolute_address((V,address)).** Usually when dealing with memory, AstréeA will keep track of which pointers point to which variables, but by default there is no actual physical address mapping. Using this directive the analyzer can be told on which particular physical location a variable is stored. Of course, usually it is not necessary to have these “low level” details included in the analysis. But when it is required to validate the return values of functions like `shmget`, which return a pointer on success, and −1 on failure, the analyzer needs to know that the pointer returned is not being placed on address −1, so a physical address has to be given.

**known_fact((condition)).** Allows the user to tell the analyzer that at a particular program point, even if the analyzer cannot prove it, the given condition is true. So, the analyzer will use this information to refine the semantics that have been calculated so far. The only case when this directive will generate an error is when the analyzer can prove that condition is always false.

**assert((condition)).** At a given program point, this directive asks the analyzer to verify that for every possible state condition is true. This is used to enforce invariants that must be true at a specific point, but that aren’t defined by any particular operation.

**unroll((integer)).** By default the analyzer will start performing widening steps on iterations after analyzing a predefined number of loops. This standard value is established by the command line arguments `–main-unroll n` and `–inner-unroll n`. But these options address every loop, and it is sometimes hard to find one single number that will at the same time suit all iterations and result in efficient execution times. So, by using this directive before a loop, one can override the number of iterations analyzed before the widening of that particular structure. This allows to choose a much smaller general unroll value, and then performing the precision adjustments necessary only on a few determined loops.
2. Background theory

smash_variable((var, size)). Allows the representation of all the elements of arrays contained in var that have more elements than size in a single cell. This means that an update on any of these elements is considered as an update on the whole array. This process is called variable folding, and its great advantage is that when an array is accessed using an imprecise index, the update on n elements will be done through only one cell, instead of having to perform an update on n cells. When analyses start becoming imprecise, having the correct folding of variables can represent a major gain in execution times.

2.4.4. Types of alarms

When performing an analysis, AstréeA will write on the analysis’ log an alarm every time it finds a possible run-time error. By default the analyzer emits 3 types of alarms [1], but during the development of the stubs one more type of alarm was added. This new type is not emitted by the analyzer, but by the calls to the C standard library functions.

The different alarms are:

Definite runtime error. Is a functionality introduced by AbsInt, which indicates that all possible invariants that can reach a particular point lead to an indicated run-time error. At this point the analyzer abandons the analysis of all these traces, and goes on with the executions that don’t lead to that particular program point.

WARN (A). Represents serious errors on program execution, such as assertion failures (section 2.4.3) or array access out of bounds. On finding these errors the analyzer “recovers”, by choosing the more general semantics’ states where the error would not occur, and goes on with the analysis using only those invariants.

WARN (C). Represents less serious errors, like implicit casting or integer overflows. As with the previous alarm, with these errors AstréeA also performs a “recovery”.

WARN (!). Alarms introduced by the stubs’ implementation. Are used to denote improper use of system calls or calls that would have an undefined behavior according to the conformance documents of the operating system on which the application would run. They are not directly run-time errors, but are useful to identify the sources of alarms of the other 3 types, and also possible bugs.
Chapter 3.

3

Methodology

In order to perform the static analysis of the ATSU in the most efficient and organized way, a methodology had to be defined.

Due to the not so extensive documentation on using static analysis to verify extremely large software, most of the steps taken had to be designed at the beginning of the investigation process.

The methodology was based on the one used in [11], but it had to be adapted to the now multi-process analysis. The resulting method was then updated as necessary in order to achieve a more refined methodology that was better adapted to the particular needs.

3.1. Analysis sequence

The following is a series of iterative steps. They represent the main actions that had to be taken, in order, to perform a complete analysis of the software.

Analysis of old stubs: Thanks to previous analyses done at Airbus [11][10], there was already a library of stubs available, which was used to perform the validation of other embedded software. So, the initial step taken was to read and understand these, in order to get a general idea of how a call has to be implemented and what the requirements for them are.

Preprocessing: Now, since AstréeA is in it’s development stage, it still doesn’t count with a graphical interface to interact with the analyzer. That means that every time an analysis is made, a command has to be executed on a console.

In order to avoid mistakes on different executions, and of course to save time, a “Makefile” had to be made. It executes the corresponding commands to pre-process and link the code, and then runs the analyzer over this result. This “Makefile” makes it very convenient in further stages to modify the analysis options in order to change the parameters on the abstract domains used.

Also, if the reader remembers, the analysis was made by separating one module from the whole environment to analyze it independently. This evidently implies that there will be certain functions and variables that aren’t directly declared on the module, but that are imported from other packages. So, this stage of the process also includes finding and linking those packages to the analysis project. In this particular case, for example, it was necessary to include, apart from the IHM module, the “BIB_API_STRUCT” module, which contains
3. Methodology

all the communication mechanisms, and the “BIB_COMMUNE” module which contains a
POSIX interface for LynxOS 2.4.

Correcting and expanding the stubs: The received stubs were actually developed for a program
that would run on a PikeOS system. This OS is also POSIX compliant, but it implements
the standard differently on the points that are left “implementation defined”. This means
that each call that was made by the ATSU that had also a representation on the given
library, had to be studied and modified in order to make it compliant to the LynxOS 2.4
implementation.

Also, since the target application is different, then for some calls the precision had to be
adjusted so it would suit the new needs.

There were also certain calls to the standard library that were not used by the previous
application, and so they were not stubbed. This meant that at some point some functions
like write or read had to be developed from scratch.

In general, the stubs are developed on demand. This means that the whole C library is not
stubbred. On the contrary, as the analysis progresses AstréeA will emit an alarm every time
it finds a function that is not implemented. It’s at that moment that such implementation
is given, and not before.

Since this is an iterative process, sometimes in this phase it was necessary to not only
develop or expand the stubs, but also to correct those that were already implemented, either
because they had errors or because they needed precision adjustments.

Setting up the environment: The different modules of the ATSU have intense interaction be-
tween them. This means that, even though the IHM was being analyzed alone, the code
still expected certain interaction with the rest of the SA. These interactions range from the
IPCs to the environment variables that are set at the moment of launching the process.

In a more general analysis, AstréeA should be able to deduce by itself all these environment
interactions. But since in the current state the analyzer is not given the code, then they
must all be manually deduced and hard-coded into the analysis before running the main
function of IHM. For an extensive list of all the suppositions made on the environment,
please refer to section 5.2.

Tunning the analysis: Finally, the most important and time consuming step of the iteration
consists in running the analyzer and, for every alarm given, try to define if it is a real bug,
or if it is a false alarm. In the case of a real bug, then it has to be reported. In the case of a
false alarm, then one must find the means, as explained in section 3.2, to give the analyzer
enough precision so it can deduce that the alarm is not possible.

As expained in [11], the most optimal approach to identify false alarms is to start by
analyzing the definite runtime errors, and trying to clear them. This is because these alarms
cut complete execution traces that could be analyzed if the given alarm is false. Then, after
having an analysis with as few definite run-time errors as possible, the rest of the alarms
can be analyzed one by one in the order in which they are emitted by the analyzer.

Now, the tunning process is not always about deleting false alarms. It is also important that
the analyses actually finish so reports can be made on the number of alarms emitted. So,
generally heuristically, a balance has to be made between tolerating alarms and imprecisions,
in favor of a lower execution time.
The **success criteria** for an analysis is defined as the moment when it finishes, leaving no alarms at all (which is ideal), or when it finishes but all alarms given can be characterized as either real bugs or false alarms. If an analysis finishes but these criteria are not met, or the analysis doesn’t finish, then the previously described sequence has to be repeated, looking for points of improvement.

### 3.2. Analysis techniques

As commented before, probably the most important and laborious part of the analysis is actually tuning all the parameters and assumptions of the analyzer in order to achieve the precision that would allow to clear false alarms. Next, a series of techniques are explained that deal with some cases that can be generalized to a wide variety of situations that arrive when performing an analysis.

Most of these techniques were taken from particular cases found in the analyzed code, and then were generalized to more situations that were alike.

#### 3.2.1. Separation of code

As explained in [11], sometimes it can be considerably hard to find the reason why an alarm is false when such alarm is located either in a very long call stack, or very far ahead in the analysis. In this situation, the log’s messages can either contain too much information to be understood, or could take very long to arrive, making debugging by emitting output messages a very slow process.

What is proposed in this situation is to actually separate the “snippet” of code into an individual file, along with enough context to simulate the environment in which the particular instructions are executed. Then, an analysis can be made only on this piece of code, which will clearly take much less time than analyzing the full program.

If the alarm can indeed be reproduced and solved, then the corrected code can be reinserted in it’s original environment and a full analysis can be made again to make sure that the modifications still represent a solution. If they don’t, then that means probably the environment was not properly extracted, and so it is necessary to rethink the extracted “snippet” to better adapt it to it’s real execution. There are, though, cases when the environment is properly extracted but the alarm cannot be reproduced. In such cases, then the extraction of code is enough to prove that the alarm is false. The reasons why this happens is out of the scope of this investigation.

#### 3.2.2. Separation of threads

On a multi-threaded software, and particularly when there are a lot of calculations done on each thread, it can become extremely slow to wait for an analysis to reach the points where the debugging information is. This is because of how the interference semantics are executed. If the reader remembers from section 2.4.1 AstréeA performs iteratively sequential analyses of each thread, where it calculates the interference set. The sequential analyses are done in the order in which the threads were created. So when the threads carry a strong payload, the analyzer can take quite some time to reach checkpoints on the threads that were created last.
3. Methodology

So, an intelligent idea lies in performing separate sequential analyses, where the program only spawns one thread. That is, all the calls to `pthread_create` but one are deleted, and an analysis is run for that thread. This can then be done for each thread spawned.

Basically, this consists of reproducing manually the first “parallel iteration” of AstréeA, where the analyzer goes through the sequential execution of each thread in turn. Of course, it is not an exact replication, because in this first iteration the analyzer does take into account the interferences that are calculated for each thread on the next analyzed one, just that they don’t play such an important role.

So, performing these separate analyses, the “sequential alarms” (not related to the interferences) that occur in each thread can be identified and corrected. Having done that, a parallel analysis can be done, where much less alarms will be reported. This is a great gain, considering that solving parallelism related bugs is already complicated without having to distinguish the parallel alarms from the sequential ones.

3.2.3. Partition of randomly failing functions

In order to make this method more clear, it will be presented with a particular case, then it will be generalized. Also, there is a proposed improvement to this method, which is discussed in section 6.2. This improvement involves a rather new functionality of the directive `__ASTREE_partition_control` which wasn’t entirely tested when the research took place.

It is a very common, yet not evident situation, that a call to a function shall fail due to a random circumstance. A very particular example of this is a call to an interruptible input/output function, such as `read`, `write` or `open`. These can, at any given point, fail if the program receives a signal during their execution. Even more so, if the signal doesn’t kill the process, it is guaranteed that the function will not do anything, but return $-1$ to the caller.

In the stubs this can be implemented by adding the code presented on figure 3.1 to the beginning of the I/O function.

```c
1 // Declare this variable global
2 int zero_one;
3 __ASTREE_volatile_input((zero_one, [0, 1]));
4
5 // This code goes at the beginning of the I/O
6 if (zero_one)
7 {
8     errno = SIGINTR;
9     return -1;
10 }
```

**Figure 3.1.:** Simulation of the reception of a signal in a stub

The `__ASTREE_volatile_input` directive will make sure that the abstract value of the variable always remains in the indicated range. So in this case `zero_one` can be used to take random decisions.

Now, looking at the code in figure 3.2 and assuming that in the executed environment a successful read (not interrupted) will always read `nbyte` elements (doesn’t reach the end of the file), then the deduced interval for `ret` will be $[-1, nbyte]$, since the intervals for both possible returns are
joined. Now, that means that after doing the filtering on line 3, \( \text{ret} \) will have the abstract value \([0, \text{nbyte}]\). But it is clear that under the specified environment that is a very coarse abstraction that will generate a false alarm for division by 0 on line 5.

```c
1 int ret = read(fildes, buf, nbyte);
2 if (ret = -1) return 0;
3 int inv_ret = 1 / ret;
```

**Figure 3.2.** Example code for partitioning I/O

The problem here is that it can become hard to remove this false alarm, seeing that the analyzer has already merged both traces. Also, a partitioning of variable `zero_one` is not possible, because due to it’s volatile condition, even when partitioned AstréeA will keep the same interval on both traces, therefore obtaining the exact same result interval twice. Also, taking away the volatile condition on the variable is not possible, because that would let the analyzer make deductions about it, which would make it lose its randomness.

So, the proposed solution lies on creating a small “communication bridge” between the stub and the caller through a boolean and a random variable. Let us call the boolean `__is_partitioned` and the variable `__part_zo`. By doing this a partition can be made by performing the correct prologue and epilogue to the call to `read` and some modifications to the handling of the reception of the signal, as explained in figure 3.3.

Having these functions defined, then after the prologue a partitioning directive can be introduced to split the traces over variable `__part_zo`, which is not volatile. In this situation then, at the return of the `read` the analyzer will have the two separate traces, one with the return value \( \text{ret} = -1 \) and the other with \( \text{ret} = \text{nbyte} \). So this time, after the filtering of the error case, the analyzer will only have one trace with the abstraction \( \text{ret} = \text{nbyte} \), therefore eliminating the false alarm.

It is important now to note that even though it was discussed here how to perform this partitioning for an interruptible I/O function, this can actually be applied to any function that can fail randomly. All that is necessary is to create the “bridge” with the boolean and the partitioned random variable.

### 3.2.4. Division of intervals

The need for this arises because of the inability to filter values from the intervals domain. For example, on the code presented in figure 3.3, the analyzer would raise an alarm because of an assertion failure on line 6. Note that this is actually impossible, since the case where \( t = 1 \) is filtered by the conditional on line 5.

The problem here rises because AstréeA is incapable of removing values that are inside an interval abstraction. So even though there is an explicit return on the case where \( t = 1 \), the abstraction contained in the analyzer is \( t = [0, 2] \), so the `return` makes no difference.

The general solution proposed for these situations counts on the fact that even though AstréeA cannot remove values from inside an interval, it can remove them from the sides. So, it is useful
3. Methodology

```c
void prologue ()
{
    __part_zo = zero_one;
    __is_partitioned = 1;
    return;
}

void epilogue ()
{
    __is_partitioned = 0;
}

// This is the new code that simulates the signal
if (! __is_partitioned)
{
    if (zero_one)
    {
        errno = EINTR;
        return -1;
    }
}
else
{
    if (__part_zo)
    {
        errno = EINTR;
        return -1;
    }
}
```

Figure 3.3.: The prologue and epilogue to a partitioned I/O

to take advantage of this fact to remove false alarms by using the partitioning directives as shown on figure 3.3.

The idea here is to force a partition on a condition that is inserted by the user, and that will divide the interval in two, leaving the undesired value in one of the sides. Now, in this case, the analyzer will have after the `return` two traces, one where \( t = \{0\} \) and one where \( t = \{2\} \), therefore eliminating the false alarm.

This solution can also be combined with the directive `__ASTREE_known_fact` to remove alarms that have the same structure but that don’t have the conditional, but where one can prove manually that in fact the undesired value is impossible. It is also important to note that it is possible to nest the partitions, meaning that intervals can be made finer through successive partitioning. But the user must have into consideration that this may have serious effects on the performance of the analysis and that maybe it is better to tolerate the alarms in such case.

3.2.5. Folding of different array’s elements differently

This addresses a “limitation” imposed by the analyzer when performing a smash of an array using `__ASTREE_smash_variable`.

According to [1] and [19], even though this directive can be applied to subscripts of a structure, it cannot be applied to sub-elements of an array. This means that a folding directive applied to
3.2. Analysis techniques

```c
1  int t;
2  .
3  .
4  // Here the value for t is [0, 2];
5  if (t = 1) return;
6  ASTREE_assert((t != 1));
7  .
8  .
```

Figure 3.4.: Example of the utility of interval division (with alarm)

```c
1  int t;
2  .
3  .
4  // Here the value for t is [0, 2];
5  __ASTREE_partition_control if (t <= 1);
6  if (t = 1) return;
7  __ASTREE_assert((t != 1));
8  __ASTREE_partition_merge_last();
9  .
10  .
```

Figure 3.5.: Example of the utility of interval division (no alarm)

An array variable will be applied to every element of such array. Nonetheless, that may not be always desirable.

An example of this is the structure used to represent files in the analyzer that was used in the stubs. An initial approach to do this was to create an array of file structures, which were called `file_spec`, that would contain every file that exists on the disk. So, something like what can be seen on figure 3.6.

```c
1  struct file_spec FILES[MAX_SYSTEM_FILES];
2  __ASTREE_smash_variable((FILES, 0));
```

Figure 3.6.: Initial approach to file stubs

Now, there exist several types of files. The chosen implementation includes, for this issue, an attribute inside the structure `file_spec` that marks this difference. But then, throughout the analysis it became clear that the access to named pipes was too imprecise, so it was desirable to fold only the files that represented such structures. Initially this wouldn’t have been possible, but an alternate definition of this code can be done as shown on figure 3.7.

So in this solution the general `FILES` array will contain in each position a pointer to either a regular file from `FILES_REGULAR` or to a FIFO file from `FILES_FIFO`. For the first, the contents, which had important parameters, were completely unfolded. While for the second ones, the contents became too imprecise due to excessive interactions between the threads, so they could be completely folded to gain analysis speed.
#define MAX_SYSTEM_FILES MAX_FILES_REG + MAX_FILES_FIFO;

struct file_spec *FILES[MAX_SYSTEM_FILES];
__ASTREE_smash_variable((FILES, 0));

struct file_spec FILES_REGULAR[MAX_FILES_REG];
__ASTREE_smash_variable((FILES_REGULAR, 0));

struct file_spec FILES_FIFO[MAX_FILES_FIFO];
__ASTREE_smash_variable((FILES_FIFO, 0));
__ASTREE_smash_variable((FILES_FIFO[].content, 1));

Figure 3.7.: A solution that allows different unrolling on the same array
Chapter 4.

Development of stubs

As commented on previous chapters of this report, one of the main elements that AstréeA needs to perform an analysis is an abstraction of the calls to the C standard library that are made by the analyzed program.

In this chapter an in depth explanation of the developed library of stubs will be given. It will include a more precise description of what a stub exactly is and how it is created. Then a discussion of the most important and interesting calls developed for the analyses of the ATSU will be given.

4.1. Definition

The AstréeA analyzer, as seen on section 2.4.1 performs its static analysis by evaluating, in turn, each instruction that can be executed in the program, and by modifying the abstract domains that have been calculated so far. This means, in particular, that every time a function is called the analyzer will have to “jump” to the code of that function and perform the proper analysis to calculate the possible return values and side effects.

In what function calls concerns, there are two different types: the user defined functions and the library calls. For the first there is, in principle, no problem for the analyzer, since the programmer must provide the code for it’s defined functions in order to perform a regular compilation. The real inconvenient appears on the second case, because the writer of the code counts on the fact that certain functions come from libraries that are provided by the run-time system, and so has no need to provide the code for himself. These are what we call the C standard library functions, that must be made available by every C compiler that abides the standard [21].

So, the inconvenient arises when the analyzer arrives at a point where it has to “jump” to a call provided by these standard libraries. It can certainly not perform an analysis directly on the implementation given by the system, since these use a number of low level directives that could make modifications on attributes directly related to the system. For example, a call to write would perform a modification on a file, and could possibly create a new inode, on a Linux environment. In general AstréeA won’t have access to these kinds of resources.

The solution to this inconvenient then is to have the user of the analyzer “re-develop” the C standard library and provide it at the moment of the analysis. Clearly, for the same reasons mentioned above, and also because of some of the limitations discussed in section 2.4.2, the functions will not have an exact copy of the real library call. In fact, the idea behind developing
A stub is to capture not the functionality but the semantics of the call, and then express that by the means of AstréeA directives.

An example of this is the implementation of a stub for function `strlen` on figure 4.1. In it one can see that the call uses variables that are most surely not present on a regular operating system, such as `in_parallel`, but that serve the analyzer to deduce the semantics. In this particular case, if AstréeA is not on it’s parallel phase, then it will perform a fully unrolled loop that calculates the length of the string, just as it would be done regularly. But, if it is indeed on the parallel phase, then a much coarser abstraction of the value is returned.

```c
1 unsigned int u_zero_inf;
2 __ASTREE_volatile_input((u_zero_inf, [0,4294967295]));
3
4 size_t strlen(const char *s)
5 {
6   size_t i;
7   if(!in_parallel)
8     {__ASTREE_unroll((MAX_FILE_NAME_SIZE))
9      for(i = 0; s[i] != 0; ++i) ;
10     return i;}
11 else
12 {
13   // Make sure the string is well formed
14    for(;*s;++s);
15    return u_zero_inf;
16 }
17 }
```

**Figure 4.1.: Example of a stub for strlen**

The second case exemplifies precisely the concept of giving only the semantics of the function, and not it’s real implementation. Intuitively one can deduce that a call to `strlen` will return, for any well formed string, a positive number containing it’s size. So if no precision is necessary, then the stub can just return exactly that, a positive number representable by the computer.

In order to get the semantics of a particular library call, the documentation for the target operating system has to be consulted. In particular for LynxOS 2.4, which complies with IEEE POSIX 1003.1 and 1003.1b [14], the semantics of the C standard library can be obtained by reading the respective POSIX standards and then filling the gaps they leave using the LynxOS 2.4 compliance documentation.

### 4.2. Relevant developed stubs

On section 4.1 it was established that it is necessary, before doing a static analysis to develop the stubs for the standard C library. Clearly it is not necessary to make abstractions for all of the functions that are defined there, but only for the ones that are actually used by the program. In that sense, the next few sections will present the most interesting functions that were developed, along with the structures that support them, for the analysis of the ATSU, in particular of the IHM module.
4.2. Relevant developed stubs

### 4.2.1. Multi-process support

As was established on section 2.4.2, AstréeA has currently no support for multi-processing analysis, only multi-threading. But it was necessary, in order to allow the solution to scale for when such a support is given, to give structures that could allow the presence of several processes in the same analysis. This scaling was imperative seeing that the ATSU is a multi-threaded and multi-processed application. So, even though the current analysis only covers one of such processes, in a future it will be necessary to count with an analysis that also takes into account the interactions between several execution units.

Just as in the job from [10] were an array called THREAD was provided with a particular structure that had the information about the threads, here a structure called PROCESS, as shown in figure 4.2, was given.

```c
#define MAX_NUM_PROC 1
struct {
    int uid;
    int gid;
    int prio;
    struct fildes open_files[MAX_FILDES];
    int num_open;
}PROCESS[MAX_NUM_PROC];
__ASTREE_smash_variable((PROCESS, 0));

extern int NB_PROC;

pid_t process_my_id( void ) ;
```

![Figure 4.2: The support for several processes](image)

In this structure all the attributes that are process dependent can be stored, so in a sense it can be seen as the PCB that is usually provided by the operating systems. Currently the only necessity on multiprocessing that was required was the handling of file descriptors (file management is further discussed in section 4.2.2), but it is evident that this structure can be easily adapted to receive further parameters. It would suffice simply to add the desired structures as fields to the existing one.

The support for files lies in the open_files array, which contains the file descriptors of the files that have been opened by the process. This structure is the equivalent to the file descriptor table provided by the OS. It is important to notice that this representation meets the criteria that establishes that the file descriptor table is shared between threads, but independent for each process.

Along with the PROCESS array a method is given, called process_my_id. This is in fact just a wrapper meant to guarantee scalability. Currently, since only one process is being analyzed, it always returns 0. But as was the case with threads, when multi-process support is given, then along with it there must be a method to obtain, for a particular thread, the id of the process to which it belongs. This function can be then placed inside process_my_id to guarantee the correctness of all the stubs already implemented. Usually the result of this function call is the value used to access the PROCESS array.
4.2.2. File manipulation (regular and FIFO)

Since AstréeA is not yet fully adapted to handle dynamic information, then building a support for access to file system calls such as read, write or open posed a real challenge.

In particular on the IHM module the case was that there was I/O both on regular files and on UNIX special FIFO files. This implied that the implementation had to be perfectly adaptable to these two specifications.

The first step in developing the stubs was to come up with a structure that could hold the information of the files in memory, for AstréeA to use. The structures used are the ones presented on figure 4.3.

```c
struct file_spec
{
    char name[MAX_FILE_NAME_SIZE];
    char content[MAX_FILE_SIZE];
    unsigned int size;
    int owner_id;
    int group_id;
    int mod;
    int fifo_head;
    FILE_TYPE type;
    int num_oread;
};

struct fildes
{
    struct file_spec *file;
    unsigned int seek;
    int oflags;
};
```

The first structure, file_spec, whose name is short for “file specification”, is the one that represents an actual file in disk. As can be seen, it has most of the attributes necessary to perform all kinds of validations on calls to open. For example, with this structure it can be guaranteed that the program doesn’t try to open a file with the wrong permissions. Also, since the size of the file has to be allocated statically, then the content array has a predefined size of MAX_FILE_SIZE, but the real size of the file is controlled by the variable size. Currently this structure supports regular files and FIFO files, but it is evident that it can be easily extended to support other types such as STREAMS.

The second structure, fildes, represents a file descriptor inside the system. This is the structure that will be contained in the array presented in figure 4.2. The idea is that every call to open has to create a new file descriptor containing the information of the opened file, and save this structure on the open_files array of the corresponding process. This file descriptor will then be in charge of controlling the correct access to the file, using the oflags variable set by open, and also control the seek pointer on the file, by the means of the seek variable.

Apart from this structure there was also one last issue to address. The IHM code accessed a great amount of files that were created before the initialization of the module, and that had information about structures that had to be initialized. So, it was necessary to give the analyzer the means
4.2. Relevant developed stubs

to have access to these files. This was achieved using the **xxd** UNIX command [22]. With it, any file that is already present at the moment of execution of IHM can be transformed from its binary form, using the “-i” option so such form is given as a C include file. Then, having the C representation of all used files, they can be put into an array of elements of the same structure as file_spec. This array is given to the analyzer, who will load everything into the FILES array, so it’ll be available for the analysis.

Finally, after having defined all the necessary structures, and how to get the external information, developing a stub for a particular call is rather simple. All that is required is to have at hand the POSIX specification and the compliance document for the OS, and then by the means of AstréeA directives provide the semantics that are expressed there. For a complete reference implementation of functions **open** and **read** refer to appendix A.

4.2.3. Shared memory

Another important library developed were the functions contained inside smem.h and shm.h. These are also used by the software to perform communication between the different processes. In particular, the shared memories are used by module GEST to send certain parameters to IHM before starting it. So, in order to guarantee a correct analysis of this module, it was necessary to give some support for shared memory.

In principle the implementation of the structures that support these calls was very much like the one used for files. That is, the predefined shared memory segments are given in a separate file, and that information is used to populate a structure like the one seen in figure 4.4.

```c
struct {
  int init;
  unsigned int key;
  char *name;
  int size;
  char *content;
} SHMEM[];

__ASTREE_smash_variable((SHMEM, 0));
```

**Figure 4.4.** Structure used to represent shared memory in the stubs

But there is a slight difference that has to be considered before having an implementation that can be correctly interpreted by the analyzer. The small error lies in the fact that function **shmat** returns, on success, the address of the attached memory segment, but on failure it returns −1 [15]. This implies that on several parts of the code, the programmers try to compare the value of the return of this function, which should have a pointer type, with −1. Unfortunately, in general AstréeA doesn’t correctly support the comparisons between a pointer and an integer, because the physical location of the pointers is abstracted away by the interval domain [1].

To deal with this inconsistency then, it is necessary to redo the mapping of each shared memory pointer initialized to an actual physical address. This was achieved by using the directive __ASTREE_absolute_address, which allows the user to specify a physical location where the pointer should be located. The declaration of a subset of the imported shared memories can be seen on figure 4.5.

---

[22]: #
[15]: #
[1]: #
[4.4]: #
[4.5]: #
One last important consideration in this regard was that the shared memory segments required a very high level of precision for the analyses on the initialization phase of IHM, but then on the parallel phase this precision was unnecessary. That is why the smashing directives receive, in this case, the special value \(-1\), which prevents the static folding of the variable, while still allowing smashing when the access to it becomes imprecise.

4.3. Other stubs

In sections 4.2.1, 4.2.2 and 4.2.3 some structures for developing stubs were presented. But these were not the only ones necessary to give support for an analysis of the IHM module. For a reference on all the developed stubs please refer to appendix B. There a complete list of the developed C standard functions, plus the auxiliary functions used to give an implementation, are presented.

In total, 78 functions were stubbed. These include some that were imported and modified from the library used by [10], and others that had to be developed from scratch.

The functionality includes:

- Thread-safe implementation of the \texttt{errno} variable.
- From the \texttt{pthread} library:
  - Handling of multiple threads and their creation attributes.
  - Handling of condition variables and their creation attributes.
  - Handling of mutexes and their creation attributes.
- Support for opening, reading, writing and seeking pre-existing regular or FIFO files. Files created dynamically are not supported.
- Modification of the signal mask.
- Acquisition of pre-existant shared memory segments.
- Manipulation of environment variables.
4.3. Other stubs

- Basic string manipulation.
- Message passing using message queues.
- Time manipulation.
- Consultation of system parameters such as user, group and process id.

Clearly each of the libraries implemented required along with it a series of structures that had to be designed so the analyzer could handle the side effects that functions have on the system’s side.
Chapter 5.

Presentation and discussion of the results

After having performed the iterative process mentioned in section 3.1 doing big efforts to obtain a balance between false alarms and execution times, several complete analyses were achieved. In this chapter the latest analysis will be discussed, since it represents the most actual state of the research.

These results were obtained using the static analyzer Astrée. It was executed on a computer running Red Hat Enterprise Linux 6, kernel version 2.6 as operating system, with 32 Intel(R) Xeon(R) E5-2690, 2.90GHz processors and 256 gigabytes of RAM. The command line arguments passed to the analyzer were those shown on figure 5.1. For a thorough explanation of what each of these parameters mean, refer to [1].

5.1. Summary of the alarms

In the current state, the analysis is rendering 7 different definite run-time errors occurring 7,489 times, and 916 different alarms occurring 672,369 times. The result yields an approximate of one unique alarm every 1,000 lines of code analyzed.

The high number of occurrences of alarms is due to the fact that every time the analyzer arrives at a location of code that would generate an alarm, it will report it. So for example in loops, if there is an instruction that raises a possible error, then every iteration of the loop that is made will write an alarm on the log. This means that certain warnings are repeated numerous times, which is why there is also a count of different alarms.

This analysis runs in 34 hours, 52 minutes and 16 seconds, which can be decomposed as follows:

- **Initialization phase** 0 hours, 0 minutes and 14 seconds.
- **First parallel iteration** 7 hours, 28 minutes and 37 seconds.
- **Second parallel iteration** 7 hours, 1 minute and 9 seconds.
- **Third parallel iteration** 6 hours, 5 minutes and 8 seconds.
- **Fourth parallel iteration** 6 hours, 47 minutes and 4 seconds.
- **Check parallel iteration** 7 hours, 29 minutes and 47 seconds.
As expected, the analysis of the initialization phase of the process, which covers all the code executed before the threads are started, is the shortest part of the analysis. Even though a very high precision was requested for this part, in order to reduce the number of alarms in the parallel phases, apparently the number of instructions executed is still not that much as to take too much time. This event can also be attributed to the fact that, as seen on section 5.2, the functions that do error handling were stubbed. Then, since the error processing is not deeply analyzed, what is left is to make sure that the “correct” traces don’t generate any run-time error. This labor is not so time consuming.

It was also expected that the parallel iterations should take that long. But, what was a surprise was that the second parallel iteration actually ran faster than the first one. Usually the opposite is the case, since in the first iteration the analyzer still has a lot of precision, which was gained on the initialization. When the interferences appear, usually the imprecision introduced forces AstréeA to analyze much more traces, therefore rendering a longer analysis. This atypical phenomenon could be explained, perhaps, by the fact that most of the communication in the ATSU happens between processes, not threads of the same process, and via IPCs, which were completely abstracted. This means that the interference set doesn’t vary that much, and so AstréeA can use cached iterates of invariants of previous executions to make faster deductions. This same fact also explains the low number of parallel iterations, meaning that the interference set stabilizes quickly.
5.1. Summary of the alarms

Now, with respect to the alarm count, the following presents the alarms that occurred in strictly
different locations of the code (line and column). The total may differ from the one presented
by AstréeA, since the sum introduced by AbsInt uses a different criteria to determine if two
alarms are different. This type of classification, though, allows for a better understanding of
which locations in the code are actually generating the errors. Usually the same location tends
to generate several different alarms due to different errors that can be the result of the same
imprecision. So knowing the alarms from different lines of code gives a much better approach on
how many places a tuning has to be made. The result is the following:

**Mild alarms (C):** 860 different alarms in 56.152 occurrences.

**Serious alarms (A):** 56 different alarms in 90.787 occurrences.

From the serious alarms, there are some that are known to be false alarms, since they were foreseen
to appear due to imprecisions introduced in the analysis to gain on execution time.

In particular there are 2 locations related to invalid pointers to functions. These are caused
by two particular locations in the code that have a structure like the one presented on figure 5.2.

```c
1  if( function_pointers[ State ][ Event ]. Action )
2    Result = function_pointers[ State ][ Event ]. Function
```

**Figure 5.2.: Reasons for the invalid pointers to functions**

In the analyzed code, the equivalent to table `function_pointers` is calculated with complete
precision, which means that for each `State` and each `Event` the analyzer has exact knowledge of the
function that is pointed. It is also proven by the analyzer, that every time that the corresponding
action is different from 0, as validated by the conditional, there is indeed an associated function.
In every turn AstréeA has also good precision over the value of `Event`. So, these false alarms
could be easily solved by partitioning the analysis over the values of variable `State`. But, the case
is that sometimes several elements contain the same function, so a partitioning actually causes
analyses to be repeated up to 5 times. But when the `State` is not partitioned, the analyzer will
calculate a set generalization of the possible values of the function called in line 2 of figure 5.2
where each possible value is unique. Then each possible function called will be analyzed only
once. This represents a huge save in execution time, since some of the called functions perform
very heavy computations. So, the alarms are tolerated in regard of speed.

Another important source of serious alarms, that comprise around 15 serious warnings, are
the ones related to accessing buffers using offsets taken from message queues, or performing
`memcpy`s with these offsets. The inconvenient in clearing these alarms was that the scope of the
analysis didn’t involve modules other than IHM. So, it was not possible to validate that the
values received from the message queues wouldn’t cause overflows on the buffers because there
was no way of knowing which these parameters would be. Nonetheless, when a full scale analysis
is accomplished, it should be fairly straightforward to clear this.

So, generalizing a little on the reasoning over these two sources of alarms, it is safe two say that
at least 15 warnings of type A are either known to be false or easily verifiable through a full
analysis. This leaves around 40 serious alarms that have yet to be checked.

But an important point that has to be noted here is that between these serious warnings there
are 3 that have to be solved with utter priority before continuing the analyses. These alarms
indicated the call of functions \texttt{strncat}, \texttt{strncpy} and \texttt{strstr}, which have not yet been stubbed. It is essential then to implement them, because AstréeA will introduce completely general, and sometimes incorrect, hypotheses after these calls, and that could be the cause of several definite run-time errors and alarms, or could even rend the analysis unsound.

Regarding the mild alarms, a similar reasoning to the one made for the serious ones can be made, in order to reduce the real work to be done in the future.

At least 48 of these mild warnings were related to implicit casts between signed and unsigned integral types of several different sizes. Now, even though this could eventually lead to errors due to modular arithmetics on the integer overflows that may be overlooked, there was no evidence on the alarm logs to prove that these implicit casts could cause bugs.

So, merging the two previous reasonings, it can be concluded that for this work to be completed there are still around 800 alarms that have to be dealt with.

Regarding the definite run-time errors, there are also several of those alarms that can be cleared since they are associated to the assertions introduced to avoid the analysis of the error logging functions, which will be discussed in more detail in section 5.2. These are 2 of the 7 reported by the analyzer, so that means that there are still 5 definite errors that must be studied. As commented in 3.1, future works should start by clearing out these errors, in order to allow the analyses of some traces that may be left out because of them.

As of the rest of the alarms, their reason and veracity has to be determined by further investigation on this software through the iterative process already described.

5.2. Suppositions for correctness

As has been thoroughly mentioned, the IHM module was analyzed “detached” from the full environment where it is supposed to be executed. This means that there were several elements that either gave alarms or made the analysis too slow, and that were related to the interactions that the module has with the rest of the SA. In order to achieve a better alarm count and faster times, a series of suppositions were made about this environment. These suppositions have to be verified in order to maintain the correctness of the results presented before. If any of them doesn’t hold, then the appropriate adjustments have to be made to the analysis that’s being made in order to correct the false assumption and get a realistic result.

Most of these assumptions were included in a startup routine, called \texttt{astree\_main}, that is analyzed before the main function of IHM. When a full analysis is made this initialization function has to be taken out of the analysis, so AstréeA can work with its own deductions.

The suppositions were the following:

1. The command line arguments are given to IHM at the moment of it’s “forking” by GEST. The values that are passed, including their possible ranges, had to be deduced by manually checking the code of GEST until the call to \texttt{fork}. The values deduced were introduced in the initialization routine as follows:

   - \textbf{First command line argument:} A number between 0 and 3 or between 5 and 8.
   - \textbf{Second command line argument:} 0 or 1.
   - \textbf{Third command line argument:} A number between 1 and 3.
5.2. Suppositions for correctness

These assumptions were also validated by [3], but they remain to be proven correct by the analyzer.

2. As with the command line arguments, before the beginning of IHM the GEST module sets up an environment for the execution. The values and names of these variables were also deduced from the code, and are as indicated in figure 5.3.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEMOIRE</td>
<td>/usr/memoire</td>
</tr>
<tr>
<td>CONFIG</td>
<td>../../usr/config</td>
</tr>
<tr>
<td>DEVICE</td>
<td>/dev</td>
</tr>
<tr>
<td>IPC_SA</td>
<td>../../bin/ipc</td>
</tr>
<tr>
<td>IPC</td>
<td>../../usr/ipc</td>
</tr>
<tr>
<td>VERSIONS</td>
<td>../../usr/config/versions.cfg</td>
</tr>
</tbody>
</table>

Figure 5.3.: Assumptions of names and values for environment variables

3. As has already been commented, IHM doesn’t perform the creation of any file, on the contrary it expects a series of pre-existing files that have certain information. The files that were deduced to be already existing are the following:

- /tmp/usr/config/creation_sa.cfg,
- /tmp/usr/config/creation_sa_aux.cfg,
- /usr/config/creation_app.cfg,
- /usr/config/info_iiev.cfg,
- /bin/ipc/pipe_sa,
- /bin/ipc/pipe_ihm,
- /bin/ipc/pipe_mcdu,
- /bin/ipc/pipe_dcdu,
- /bin/ipc/pipe_[10-26],
- /dev/fifo_qat_02,
- /dev/fifo_qat_03

Even more so, the files ending in “.cfg” must meet a certain structure and contents. But these are way too long and complex to include here. On the folder containing all the analysis’ data there are the binary representations of the files that were used during the project.

4. The files on the previous point that represent named PIPEs on the file system must have been already opened for reading before the IHM process starts. This is because IHM only opens them for writing using the O_NDELAY flag. In this case if there are no readers the call to open will immediately fail and abort the execution of the module.

5. Shared memory objects with the following names must exist before the start of IHM:

- /usr/memoire/info_iiev,
- /usr/memoire/emission,
5. Results

- /usr/memoire/label,
- /usr/memoire/sorties_sao,
- /usr/memoire/ram_3,
- /usr/memoire/config_iev,
- /usr/memoire/indexbds

6. Regarding the messages received via message queues, it is indeed true that no information can be inferred directly by the analyzer, due to the fact that it doesn’t count with the code of the processes that send the messages. But in [3] there is a thorough explanation of how these messages must be formatted before being sent.

To avoid a propagation of too many alarms, it was assumed that all processes respect the formats mentioned when sending a message. In particular the messages received on the message queues must have a maximum size of 304 bytes. Also a correct formatting guarantees that the area where the length of the message should be specified will contain a number between 1 and 300, since the last 4 bytes of the message are used to place a “magic cookie” that validates the correctness of what is sent.

7. There are a couple of functions that are are used for notifying when errors occur during the execution of a module. Neither has any side effects on the variables used by the program, their only effect is that they send a message to the GEST module using a particular named pipe.

But the inconvenient with these two functions is that they are called everywhere a failure could occur. And since AstréeA has in its abstractions the possible error cases, then on each of these checkpoints it performs an analysis of one of these functions. This results in an endless number of repeated analyses that have an execution time cost so big as to reach a point where an analysis was running for about 23 days and hadn’t finished yet.

To solve this issue these functions were stubbed by placing at the beginning of each a directive \texttt{\_\_ASTREE\_\_assert((0))}, which clearly will always cause a failure. This implies an immense number of alarms of type A indicating the failure of the assertions, but at least it avoids the repeated analysis of these functions. This solution has the drawback though, that it assumes that every trace that leads to the execution of any of these functions will cause the death of the process. It also assumes that the execution of the functions themselves before the death is completely bug free.

5.3. Recommendations to ÉNS

Even though no bug was found on the execution of the analyzer and it’s calculations of abstract domains, there were some semantics that were abstracted way too coarsely, an so resulted in analyses that were way too imprecise. So, it was suggested to perform corrections over these calculations to achieve closer approximations. The recommendations were the following:

- When a pointer was casted to its integer representation, operated with some other integer, and then casted back to it’s original representation, the analyzer lost complete precision on the abstraction it had of the pointer. This means that after the operation was done there was no trace of the real offset the pointer had with respect to it’s base.
5.4. **Recommendations to Airbus**

This abstraction is correct because it will contain the scenario were the pointer is only moved the desired offset. But it also includes every other possible case, which means that from the point of the operation and on there is no useful information for the analyzer on this variable.

There was a request to modify this behavior, so the variable after being casted to integer and back would retain its properties inside the pointer domain.

- One area which was not very polished was the handling of pointers when their address was fixed using the \texttt{\_\_ASTREE\_\_absolute\_address} directive, as was seen on figure \ref{fig:absolute-address}. In particular the analyzer was not capable of determining that a pointer fixed on an address different than 0 was in effect not a \texttt{NULL} pointer. That would mean that when the variable was compared with \texttt{NULL}, it would return just a random result, indicating that the outcome of the operation could be anything.

Again, the mentioned abstraction is correct because it covers all possible cases on the result of the comparison. But for the target application, this imprecision could mean the analysis of several not-so-short functions that could have serious repercussions on execution times. So in order to get precision on these points, a request was made to modify this behavior.

5.5. **Resources produced**

The whole set of resources that was generated during the investigation is being left under the directory assigned for the development of the solution. This directory contains the following information:

- **ANALYSES/**: Extracts of the ATSU’s code that had to be independently analyzed in order to clear certain false alarms. Each directory inside is named after the function that it extracts and analyzes.

- **ATSU\_FILES/**: The predefined files that had to be incorporated as hypotheses to the analyzer. Their contents are supposed to be the same as when IHM starts.

- **CODE/**: The analyzed source of the ATSU, including the IHM modules and all the supplementary libraries needed to compile it. The code present here is annotated with AstréeA directives that solve certain false alarms.

- **CODE/ORIGINAL/**: The unannotated code.
5. Results

LOGS/: All the outputs of the analyses made. With its contents, the progress of the research can be seen from its very beginning.

LOGS/ALARMS/: Alarms given by each of the analyses that finished. They are grouped by level (definite, serious, mild and stub).

LOGS/ANALYSES/: The full log files resulting from all the analyses, both the ones that finished and the ones that didn’t. They are grouped according to the thread that was particularly analyzed.

Makefile: Used to execute the analysis. It is parameterized to help easily execute analyses in parallel by assigning them different ids. It also allows the generation of solely the preprocessed code used for analysis, or to perform an analysis based on an already preprocessed code. To get the different implemented functionalities, use the command “make” on the directory containing this file.

MAKE_SCRIPTS/: Small utilities that serve the purpose of automatically generating the structures that have to be included in the analyzer, such as the input files or the shared memory segments.

MERGED_SOURCES/: The preprocessed code used for each of the analyses made. It is organized in the same fashion as “CODE/ANALYSES”, and their elements have a one-to-one mapping that associates a log to each of the preprocessed files.

stub_libc/: All the files belonging to the stub library developed.

stub_libc/impl/: The actual implementation using AstréeA directives of the standard C library functions. Here there’s the reference implementation used by [10] for PikeOS, and the implementation used for analyzing the ATSU on LynxOS 2.4.

stub_libc/include/: All the headers of the standard C library.

STUBS/inc: The information that was assumed to be true and so was introduced as input to the analyzer. This includes the structures containing the binary representations of files, the shared memory structures and the multi-processing extensions.

TESTS/: Several tests that exploit the functionality of the analyzer and serve to have a better understanding of how the abstract domains implemented work and which are their limitations. It also performs several tests on independent analyzer directives for partitioning, packing and logging, among others.

x86.abi: Configuration file that tells AstréeA the low level details associated with the operating system, such as the byte sizes of variables, the results of pointer arithmetics, alignment of values and “endianess”.
Chapter 6.

Conclusions and recommendations

6.1. Conclusions

Having performed this study, it becomes more than evident that using informal methods, such as unit testing, is certainly not enough guarantee on a safety critical environment. As was seen, there are way too many border cases and rough edges that cannot be covered by a human designed test.

In particular because of that last point, there surges a need of extending the use of formally proven methods, such as abstract interpretation, to validate software that may have serious repercussions when failing. It is imperative for every piece of program on which human lives count on to be certified to not fail on any unforeseen situation. It is here when AstréeA becomes an useful tool. It has been made specially for the analysis of software running on a real-time scheduling OS with fixed priorities and a mono-processor architecture, such as the ones developed at Airbus, and it either guarantees that a program will not fail, or will give the points with real errors.

In adopting a static analyzer as part of the certification process, it becomes also necessary to adapt the code that is written to the use of such tool. Programming in a way that allows the analyzer to infer invariants would cut back a lot of time during its run, since it can have a more clear idea of what the data actually looks like. A clear example of that are the two ways of implementing a condition for a loop shown on figure 6.1.

```
1 //Two ways of specifying an en condition
2 int it = 0;
3 int total = 100
4
5 while(it != 100)
6 {
7     // Do something that results in the increment of it
8 }
9
10 while(it < 100)
11 {
12     // Do something that results in the increment of it
13 }
```

Figure 6.1.: Two ways of implementing a condition on a while loop
6. Conclusions and recommendations

Even though the general semantics of those two end conditions are different, there are many cases when both of their loop invariants are the same. In those cases it is preferable to have the second condition, since it allows the analyzer to always have a bound on it, while on the first condition it is easier for AstréeA to just generalize the value of the iterator to a much bigger interval.

Now, even though AstréeA is a very powerful tool to validate software, it also has several limitations that must be considered before incorporating it to a project.

First of all, it has to be noted that the use of a static analyzer is not conceived for high speed, changing environments, were code has to be handed in in short periods of time and is constantly changing. This is because performing a static analysis is a slow process that requires a lot of resources to get a final result. As can be seen, this investigation lasted 6 months and still more than 800 alarms remained to be solved. An AstréeA user must take his time to analyze each and every alarm given by the analyzer, in order to define if it is true or false. If it is false, then the imprecision must be fixed so the correct invariant can be inferred, and this is not always an evident process.

Also, it is important to take into account that as the size of the software grows, the logs produced by AstréeA become bigger and bigger, especially if the analyses are not well optimized. This implies that the computers where the analyzer works need a lot of storage capacity, as well as computation capacity, in order to get the most optimal conditions.

Lastly, one big drawback on the execution of these types of analyses is that by the time a zero false alarm analysis has been reached, the resources produced are so specialized that it becomes hard to reuse them for a different purpose. Probably the most useful element is the stub library, but this is also adapted to the particular operating system where the application is supposed to run. This means that starting a new analysis is always a slow process, since very little can be reused from the previous ones. But, as more studies of this kind are made, more resources will be available for reutilization, and maybe at a given point several resources can be merged in order to get initial analyses without much effort.

6.2. Recommendations for future work

As was made evident in section 5.1, the analysis of the ATSU is far from being over. There is still the need of finish solving the almost 800 alarms that are still pending, and then expand the analysis to the rest of the modules, and possibly to incorporate a full inter-process communication analysis.

In order to achieve this goal, the future researcher must first, of course, acquire the necessary knowledge in abstract interpretation. The suggested order of documentation for a complete beginner would be the one explained in section 1.2. This approach goes from the most basic concepts to the more complex, avionic specific approaches of abstract interpretation, covering all the necessary information to perform an acceptable analysis. Then, of course, it is imperative to read and comprehend at least one library of stubs and see how it was used to perform an analysis on a real piece of software. This should give the most general ideas that can be applied to almost every static analysis.

On any investigation using AstréeA, it is imperative to maintain constant contact with the ÉNS. Since the analyzer is still in its development phase, there are many functionalities that may not work as desired and with a simple email this can be corrected. But most importantly, there are
also parts of the analyzer that are not included in the documentation, but that exist and solve most of the problems that one can find.

Now, with respect to the current stub library. It clearly needs to be extended in order to allow for more functions of the C library to be analyzed. Also there are many functions that will have to be tuned to adapt better to the analyses of other modules. But, there is in particular one modification to the stubs that is suggested, for efficiency and readability issues. This regards the partitioning of interruptible I/O, mentioned in section 3.2.3. Since AstréeA is a work in progress, there was one particular functionality associated with the directive __ASTREE_partition_control that was not entirely evident at the moment of the research, but that may now be mature enough to be used in an analysis. It consists on an extension of the directive, which can be applied now not only to control structures but also to function calls.

So, in the particular case of a function call, when the directive is present just before the instruction that performs it, the regular behavior that merges all traces just before the return of a procedure is overridden. This means that using this new method, the different traces generated by a function call, along with the values associated to the variables, can be kept on the caller to prevent the loss of information on return values and side effects associated to them. This can indeed be used to do a reimplementation of the I/O partitioning method previously discussed, which would no longer need explicit communication between the caller and the callee. It is encouraged to implement this on the places where the method was used, in order to improve efficiency and readability of the code.

Also regarding the stub library, it is imperative to implement functions strncat, strncpy and strstr, since without them a full analysis cannot be reached.

And finally, one last comment on extra knowledge that is recommended. The analyses can run for very long times, therefore generating log files containing several gigabytes of information. As the size of the code gets bigger, these files can become hard to deal with, generating a lot of problems when being opened on editors and using them to search for particular information that is needed. That is why it is advisable to get strong background on regular expressions, that allow the construction of patterns to quickly extract the necessary information from the logs using programs like egrep or Perl. A personal advice is to thoroughly study [13]. Some examples on regular expressions used to search the logs can be found on appendix C.
Bibliography


Appendix A.

Stubs for the open and read functions

```c
int open(const char *path, int oflag, ...)
{
  pid_t pid = process_my_id();
  int i, ret;

  // Enter critical zone for reading PROCESS[i].num_open
  __ASTREE_parallel_lock_mutex((-3));

  // Detect maximum number of file descriptors reached
  if (PROCESS[pid].num_open >= MAX_FILDES) {
    __ASTREE_log_vars((PROCESS[pid].num_open));
    __ASTREE_parallel_unlock_mutex((-3));
    __ASTREE_print(("WARN(!): open: Maximum number of file descriptors reached");
    pthread_t tid_err;
    __ASTREE_parallel_id_process((&tid_err));
    if (!in_parallel) tid_err = 0;
    THREAD[tid_err].t_errno = EMFILE;
    return -1;
  }

  // See if the file exists
  __ASTREE_unroll((MAX_FILES))
  for (i = 0; i < NB_FILES; ++ i)
    // This comparison only takes into account absolute paths
    if ((FILES[i] != NULL) && !strcmp(path, FILES[i]->name))
      break;

  // Make sure to log that the program made an attempt to open
  ++ PROCESS[pid].num_open;

  __ASTREE_parallel_unlock_mutex((-3));

  // Randomly see if a signal interrupted execution
  #ifndef NO_SIGNALS
  /* Only if there is a signal that can kill the process */
  if (!__all_signals_kill)
    {
      if (zero_one)
        {__ASTREE_log_vars();}
      __ASTREE_print(("WARN(!): open: Call interrupted.");
      pthread_t tid_err;
```
A. Examples of some implemented stubs

```c
44 __ASTREE_parallel_id_process((&tid_err));
45 if(!in_parallel) tid_err = 0;
46 THREAD[tid_err].t_errno = EINTR;
47 return -1;
48 }
49 }
50 #endif
51
52 // Make a distinction if the file has to be created or if it already exists
53 if(i == NB_FILES)
54 {
55 __ASTREE_print(("open:/uni2423File/notfound/inFS/creating/new"));
56 // File not found, must create a new one
57 if(NB_FILES >= MAX_FILES)
58 {
59 __ASTREE_print(("WARN(!) : open::Number_of_files_created_exceeds_system_limit.");
60 pthread_t tid_err;
61 __ASTREE_parallel_id_process((&tid_err));
62 if(!in_parallel) tid_err = 0;
63 THREAD[tid_err].t_errno = ENOSPC;
64 return -1;
65 }
66
67
68 __ASTREE_print(("WARN(!) : Valid call to open_to_create a new file Not supported"));
69 __ASTREE_assert((0));
70 }
71 else
72 {
73 __ASTREE_print(("open::Opening an existing file ");
74 __ASTREE_log_vars((i , FILES[i] >name));
75
76 // Check the permissions on the file
77 /* Build the mod flag based on the received flags */
78 int mod_r = 0; /* request mode */
79 if(oflag & O_RDONLY) mod_r |= MOD_R;
80 if(oflag & O_WRONLY) mod_r |= MOD_W;
81 if(FILES[i] >owner_id == getuid()) mod_r <<= SHIFT_U;
82 else if(FILES[i] >group_id == getgid()) mod_r <<= SHIFT_G;
83 else mod_r <<= SHIFT_O;
84 if((mod_r & FILES[i] >mod) != mod_r)
85 {
86 int gid = getgid();
87 __ASTREE_log_vars((mod_r , FILES[i] >mod , FILES[i] >group_id , gid));
88 __ASTREE_print(("WARN(!) : Trying to open file in unauthorized mode");
```
pthread_t tid_err;
__ASTREE_parallel_id_process((&tid_err));
if(!in_parallel) tid_err = 0;
THREAD[tid_err].t_errno = EACCES;
return -1;
}

if((i < (NB_FILES - 1)) && (oflag & O_CREAT) && (oflag & O_EXCL))
{
  __ASTREE_log_vars();
  __ASTREE_print(('WARN(!):open:File/exists/and/O_CREAT/and/O_EXCL/are/set.'));
  pthread_t tid_err;
  __ASTREE_parallel_id_process((&tid_err));
  if(!in_parallel) tid_err = 0;
  THREAD[tid_err].t_errno = EEXIST;
  return -1;
}

//Here the file was either found or has been created, in any case
//it will be found on index 'i' of the FILES array
if(oflag & O_TRUNC) FILES[i]->size = 0;

//Enter critical section for FILES[i].num_ored
__ASTREE_parallel_lock_mutex((-4));
if((FILES[i]->type == FIFO) && (oflag & O_RDONLY)) ++ FILES[i]->num_ored;
__ASTREE_parallel_unlock_mutex((-4));

// Get a unique file descriptor.
__ASTREE_parallel_create_id((&ret, FILDES));
__ASTREE_print(('open:The/unique/file/descriptor/created/for/this/open:'))
__ASTREE_log_vars((ret, PROCESS[pid].num_open));
PROCESS[pid].open_files[ret].file = FILES[i];
PROCESS[pid].open_files[ret].seek = 0;
PROCESS[pid].open_files[ret].oflags = oflag;

//File operations could block thread
parallel_yield();
return ret;
}

static size_t read_regular(int fildes, void *buf, size_t nbyte, int pid)
{
  // The number of bytes read will be the minimum between the requested bytes
  // and the bytes that are left in the file
  int r = (nbyte <= (PROCESS[pid].open_files[fildes].file)->size)
    // Number of bytes left in the buffer
    ? PROCESS[pid].open_files[fildes].seek
    : (PROCESS[pid].open_files[fildes].file)->size
    - PROCESS[pid].open_files[fildes].seek;

  return r;
}
A. Examples of some implemented stubs

```c
char *cbuf = (char *)buf;

// Perform the read
__ASTREE_memcpy((cbuf,
    ((PROCESS[ pid ].open_files[fildes].file)->content) +
    PROCESS[ pid ].open_files[fildes].seek,
    r
));
PROCESS[ pid ].open_files[fildes].seek += r;

// File operations could block the thread
parallel_yield();
return r;
}

static ssize_t read_fifo (int fildes, void *buf, size_t nbyte, int pid)
{
    int r;
    __ASTREE_modify((r));
    __ASTREE_known_fact((r < nbyte));

    // Tell the analyzer to put the most general value possible on each byte of
    // the buffer
    __ASTREE_trash((buf, nbyte));
    return r;
}

ssize_t read(int fildes, void *buf, size_t nbyte)
{
    ssize_t r = 0;
    pid_t pid = process_my_id();
    char *cbuf = (char *)buf;

    // Enter critical zone for reading PROCESS[i].num_open
    __ASTREE_parallel_lock_mutex((-3));

    if(fildes < 0 ||
        !(PROCESS[ pid ].open_files[fildes].oflags & O_RDONLY) || // Wrong access rights
        PROCESS[ pid ].open_files[fildes].file == NULL) // Closed FD
    {
        __ASTREE_log_vars((fildes, PROCESS[ pid ].open_files[fildes].oflags,
            PROCESS[ pid ].num_open));
        __ASTREE_parallel_unlock_mutex((-3));
    }

    __ASTREE_print(("WARN/uni2423(!) :/uni2423read :/uni2423Bad/uni2423 file /uni2423descriptor "));
    pthread_t tid_err;
    __ASTREE_parallel_id_process((&tid_err));
    if(!in_parallel) tid_err = 0;
    THREAD[tid_err].t_errno = EBADF;
    return -1;
}

if(!buf)
{
}
```

54
218 __ASTREE_log_vars((buf));
219 __ASTREE_print(("WARN(!) : read Buffer is a null pointer"));
220 pthread_t tid_err;
221 __ASTREE_parallel_id_process((&tid_err));
222 if(!in_parallel) tid_err = 0;
223 THREAD[tid_err].t_errno = EFAULT;
224 return -1;
225 }
226 }
227 #ifndef NO_SIGNALS
228 /* This must only be take into account if there are signals that don't
229 * kill the current process */
230 if(!_all_signals_kill)
231 {
232 if(!_astree_is_partitioned)
233 {
234 /* If a partition was not asked, proceed with volatile random variable */
235 if(zero_one)
236 {
237 __ASTREE_log_vars();
238 __ASTREE_print(("WARN(!) : read interrupted by a signal, no partition demanded");
239 pthread_t tid_err;
240 __ASTREE_parallel_id_process((&tid_err));
241 if(!in_parallel) tid_err = 0;
242 THREAD[tid_err].t_errno = EINTR;
243 return -1;
244 }
245 }
246 else
247 {
248 /* If a partition was demanded, do the decision with the partitioned */
249 * variable */
250 if(_partition_zero_one)
251 {
252 __ASTREE_log_vars();
253 __ASTREE_print(("WARN(!) : read interrupted by a signal, partition was demanded");
254 pthread_t tid_err;
255 __ASTREE_parallel_id_process((&tid_err));
256 if(!in_parallel) tid_err = 0;
257 THREAD[tid_err].t_errno = EINTR;
258 return -1;
259 }
260 }
261 }
262 }
263 #endif
264 switch((PROCESS[pid].open_files[fildes].file)->type)
265 {
266 case REGULAR:
267 return read_regular(fildes, buf, nbyte, pid);
268 break;
269 case FIFO:
270 return read_fifo(fildes, buf, nbyte, pid);
271 break;
272 default:
273 __ASTREE_print(("WARN(!) : trying to manipulate an unsupported file");
274 switch((PROCESS[pid].open_files[fildes].file)->type)
A. Examples of some implemented stubs

275    __ASTREE_assert((0));
276    break;
277 }
278
279    __ASTREE_print(("WARN(!) : read an unreachable point");
280    return -1;
281 }
Appendix B.

Functions implemented in the library of stubs for LynxOS 2.4

1. pid_t process_my_id(void)
2. static void parallel_lock_mutex(pthread_mutex_t mutex)
3. static int parallel_unlock_mutex(pthread_mutex_t mutex)
4. static void parallel_yield(void)
5. static void parallel_cond_mutex_bind(pthread_cond_t *cond, pthread_mutex_t *mutex)
6. static int parallel_cond_wait(pthread_cond_t cond, int timed)
7. static void parallel_cond_signal(pthread_cond_t cond, int broadcast)
8. static void parallel_cond_mutex_check(pthread_cond_t cond)
9. static void parallel_entry_point(void)
10. int *errno(void)
11. int open(const char *path, int oflag, ...)
12. __pthread_attr_init(pthread_attr_t *attr)
13. __pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize)
14. __pthread_attr_setinheritsched(pthread_attr_t *attr, int inheritsched)
15. __pthread_attr_setscope(pthread_attr_t *attr, int contentionscope)
16. __pthread_attr_getschedparam(const pthread_attr_t *attr, struct sched_param *param)
17. __pthread_attr_setschedparam(pthread_attr_t *attr, const struct sched_param *param)
18. __pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy)
19. __pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr)
20. __pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)
21. __pthread_mutex_lock(pthread_mutex_t *mutex)
22. __pthread_mutex_unlock(pthread_mutex_t *mutex)
23. pthread_t __pthread_self(void)
24. __pthread_sighandle(int how, const sigset_t *set, sigset_t *oset)
25. sigaction(int sig, const struct sigaction *act, struct sigaction *oact)
26. sigemptyset(sigset_t *set)
27. sigaddset(sigset_t *set, int sig)
28. _str_equal(char *s1, char *s2)
29. char *smem_get(char *name, long size, int perm)
30. void abort(void)
39  int atoi (const char *str)
40  char *getenv(const char *name)
41  void *memcpy(void *s1, const void *s2, size_t n)
42  void *memset(void *s, int c, size_t n)
43  static int strcmp_pr(const char *s1, const char *s2)
44  static int strcmp_impr(const char *s1, const char *s2)
45  static char *strcpy_pr(char *s1, const char *s2)
46  static char *strcpy_impr(char *s1, const char *s2)
47  char *strcpy(char *s1, const char *s2)
48  static int strcmp(const char *s1, const char *s2)
49  static char *strcpy_pr(char *s1, const char *s2)
50  static char *strcpy_impr(char *s1, const char *s2)
51  size_t strlen_pr(const char *s)
52  size_t strlen_impr(const char *s)
53  size_t strlen(const char *s)
54  int msgget(key_t key, int msgflg)
55  ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long int msgtyp, int msgflg)
56  int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg)
57  time_t time(time_t *clock)
58  int clock_gettime(clockid_t clock_id, struct timespec *tp)
59  int close(int fildes)
60  gid_t getgid(void)
61  pid_t getpid(void)
62  uid_t getuid(void)
63  off_t lseek(int fildes, off_t offset, int whence)
64  static ssize_t read_regular_pr(int fildes, void *buf, size_t nbyte, int pid)
65  static ssize_t read_regular_impr(int fildes, void *buf, size_t nbyte, int pid)
66  static ssize_t read_fifo(int fildes, void *buf, size_t nbyte, int pid)
67  static ssize_t read_regular(int fildes, void *buf, size_t nbyte, int pid)
68  static ssize_t read_fifo(int fildes, void *buf, size_t nbyte, int pid)
69  ssize_t read(int fildes, void *buf, size_t nbyte)
70  long sysconf(int name)
71  static ssize_t write_regular(int fildes, const void *buf, size_t nbyte, int pid)
72  static ssize_t write_fifo(int fildes, const void *buf, size_t nbyte, int pid)
73  static ssize_t write_regular(int fildes, const void *buf, size_t nbyte, int pid)
74  ssize_t write(int fildes, const void *buf, size_t nbyte)
75  __partition_interrupt_prologue()
76  __partition_interrupt_epilogue()
77  int getpriority(int which, id_t pid)
78  static void start_init()
79  static void start_parallel()
80  void astree_main(void)
Appendix C.

Regular expressions used to search AstréeA’s logs

1. VIm regex:
   - Get all alarms except some that can’t be cleared and implicit casts:

   ```plaintext
   \v\^WARN \(([^!\}\]: ]+ (Accessing|Call interrupted|Assertion false and error) | ^ (signed|unsigned)} (int|long)-(signed|unsigned) (int|long))@!
   ```

   - Get all alarms except stub warnings, implicit casts and a particular invalid function pointer alarm:

   ```plaintext
   \v( (HDA_Si_ExecTransition|HMA_Si_ExecTransition)\d+: [ ]+ )@<!
   \v\^WARN \(([^!\}\]: ]+ )@<!
   ```

2. Perl regex:
   - Get all alarms except stub warnings and implicit casts. Note that the false alarm on function pointers is tolerated, since lookbehinds in Perl have to be fixed length:

   ```plaintext
   WARN \((?! !\}\) | (signed|unsigned) (int|long)-\)(signed|unsigned) (int|long))@!
   ```
Find function definitions. It’s not complete, but it’s enough:

```
^\w+\s+)+(?:\*\s*)?\w+\s*\(
(?:\void |
 (?:\s*
 (?:\w+\s*)+
 \*?\s*\w+
 (?:\s*,\s*)+
 (?:\w+\s*)+
 \*?\s*\w+)*
 (?:\s*,\s*\.\.\.)?
 )?)+\)$
```