Formal verification of industrial software with dynamic memory management

Sébastien LABBÉ
EDF Research & Development
Chatou, France
Email: sebastien.labbe@edf.fr

Arnaud SANGNIER
LIAFA - Université Paris 7 - CNRS
Paris, France
Email: sangnier@liafa.jussieu.fr

***Abstract*—Tool-based analytic techniques such as formal verification may be used to justify the quality, correctness and dependability of software involved in digital control systems. This paper reports on the development and application of a tool-based methodology, the purpose of which is the formal verification of freedom from intrinsic software faults related to dynamic memory management. The paper introduces the operational and research context in the power generation industry, in which this work takes place. The theoretical framework and the tool at the cornerstone of the methodology are then presented. The paper also presents the practical aspects of the research: software under analysis, experimental results and lessons learned. The results are seen promising, as the methodology scales accurately in identified conditions of analysis, and has a number of perspectives which are currently under study in ongoing work.**

***Keywords*-Digital control systems, software dependability, formal verification, memory allocation, data structures.**

I. INTRODUCTION

Digital equipment is increasingly used in the design of industrial control systems, particularly in the power generation industry. This technological evolution entails beneficial effects: digital equipment is usually more reliable than counterpart analog equipment, while providing the capability to implement more complex functionalities. Monitoring, fault detection and diagnosis are also facilitated, providing basis for improving the overall safety in industrial plants. On the other hand, detrimental effects may also be encountered, such as complexity increased, and technology-specific issues raised. To address those, additional or specific effort is needed in qualification of digital control systems and licensing of industrial installations, particularly for the purpose of safety and reliability assessments. In current industrial practices, dependability assessment of software using formal verification techniques is mainly carried out on critical software, cf. examples in [1]. Besides, relying on recent technical advances, current work at EDF R&D aims at extending these approaches to more complex and less critical software, e.g. software important to availability. Such software may be implemented in control systems which: operate in a larger variety of environments, address more demanding functional needs, make wider use of pre-existing software, or have more configuration features. This situation may cause more "complex" programming mechanisms to be used in the implementation of software, which in return will be more difficult to formally analyze: e.g. concurrency, interrupts, dynamic memory management.

This paper reports on research in software formal verification done during a project named AVERILES[1]. Its purpose was to develop advanced techniques and tools for analyzing and verifying complex embedded software. Among the important properties to be verified is the freedom from intrinsic faults (i.e., faults that can be identified independently of functional specifications) in memory management:

- Memory leak: a memory leak occurs when a program becomes unable to release memory it has allocated;
- Illegal dereference of pointers, e.g. when a program tries to dereference a pointer which may be null, or to access to an unauthorized memory area;
- Multiple consecutive releases of a given memory cell. Semantics are indeed undefined in ANSI `C`.

Some of the theoretical developments achieved during the project have been implemented in formal verification tools, such as `TOPICS` [2], [3], or `L2CA` [4].

This paper reports on the development and application of a tool-based methodology using `TOPICS` and other tools for formal verification of industrial software with dynamic memory management. The paper presents both theoretical and practical aspects of the research: the theoretical framework on which is built the `TOPICS` tool, and the tool itself, are explained respectively in sections II and III. A case study is then described in section IV, from which the results and lessons learned are synthesized in section VI.

II. FORMAL VERIFICATION FRAMEWORK

In this section, we present the verification framework we have used to analyze the case study. The main idea lies in a translation which takes in input programs with different

[1]AVERILES : *Analysis and verification of software with dynamic data structures* (2006 – 2009). Partners involved: LSV (École Normale Supérieure de Cachan), EDF R&D (Électricité de France), VERIMAG (Grenoble University), LIAFA (Paris 7 University) and Alyotech France. Project partially supported by the French National Research Agency (ANR).

variables and produces a model instance which belongs to the classes of counter machines.

A. Counter Machines

Counter machines are finite automata extended with integer variables which are manipulated by the transition relation. In the field of verification, this model enjoys a central position for both theoretical results and maturity of tools like FAST [5], TReX [6] or ASPIC [7]. Before to give their formal definitions, we introduce some notations.

We recall that *Presburger arithmetic* is the first order theory of the structure $\langle \mathbb{N}, +, = \rangle$. The syntax of the formulae of the Presburger arithmetic can be described by the following grammar where t described a term and ϕ a formula and x, y belongs to a set of variables:

$$\begin{array}{lll} t & ::= & 0 \mid 1 \mid x \mid t+t \\ \phi & ::= & t = t \mid \neg\phi \mid \phi \wedge \phi \mid \exists y.\phi \end{array}$$

We denote then by $Presb(k)$ the set of Presburger formulae whose free variables are in $\{x_1, \ldots, x_k\}$. Given a vector $\mathbf{v} \in \mathbb{N}^k$ and a formula $\phi \in Presb(k)$, we write $\mathbf{v} \models \phi$ if the formula obtained by replacing each x_i by $\mathbf{v}(i)$ is true and let $[\![\phi]\!]$ be the set $\{\mathbf{v} \in \mathbb{N}^k \mid \mathbf{v} \models \phi\}$. Finally $Linear(k)$ represents the set of linear functions from $\mathbb{N}^k$ to $\mathbb{N}^k$.

Definition 1: A k-dim counter machine (shortly counter machine) M is a tuple $\langle Q, E \rangle$ where:

- $k > 0$ characterizes the number of counters manipulated by the machine;
- Q is a finite set of control states;
- $E \subseteq Q \times Presb(k) \times Linear(k) \times Q$ is a finite set of edges, each of which is labelled with a guard on the counters expressed by a Presburger formula and with an update of the counters values given by a linear function.

The semantics of a counter machine M is given by its associated transition system $TS(M) = \langle Q \times \mathbb{N}^k, \rightarrow \rangle$ where $Q \times \mathbb{N}^k$ is the set of configurations and $\rightarrow$ is the transition relation defined as follows: for all $(q, \mathbf{v}), (q', \mathbf{v}') \in Q \times \mathbb{N}^k$, $(q, \mathbf{v}) \rightarrow (q', \mathbf{v}')$ iff there exists $(q, \phi, f, q') \in E$ such that $\mathbf{v} \models \phi$ and $\mathbf{v}' = f(\mathbf{v})$. We denote by $\rightarrow^*$ the reflexive and transitive closure of $\rightarrow$ and given a set $I \subseteq Q \times \mathbb{N}^k$ of initial configurations, $\texttt{Reach}(M, I)$ is the set $\{(q', \mathbf{v}') \in Q \times \mathbb{N}^k \mid \exists (q, \mathbf{v}) \in I \text{ s.t. } (q, \mathbf{v}) \rightarrow^* (q', \mathbf{v}')\}$ of configurations which are reachable from configurations in I. Given a Presburger formula $\phi_0 \in Presb(k)$, we write $(q, [\![\phi_0]\!])$ to represent the set of initial configurations built up of pairs $(q, \mathbf{v})$ where $\mathbf{v} \models \phi_0$. We will say that a set $I \subseteq Q \times \mathbb{N}^k$ is Presburger definable if there exists $|Q|$ Presburger formulas $\phi_1, \ldots, \phi_{|Q|}$ such that $I = \bigcup_{i \in \{1,\ldots,|Q|\}} (q, [\![\phi_q]\!])$.

A famous problem in verification of infinite-state systems is the control state reachability problem (shortly reachability problem) which can be written in our framework as follows:

Input: A counter machine M, a set of initial configurations I and a control state q;

Output: Is there a vector $\mathbf{v}$ such that $(q, \mathbf{v}) \in \texttt{Reach}(M, I)$?

From [8], we know that the reachability problem is undecidable even when restricted to counter machines of dimension 2. However, some methods have been proposed in order to try to solve this problem, and succeed in some cases. We shortly present here two methods which have been implemented in two tools FAST [5] and ASPIC [7]. Given a counter machine M and a Presburger definable set of initial configurations I these two methods compute approximations of the set $\texttt{Reach}(M, I)$.

The first method originally presented in [9] computes a sequence of Presburger definable sets of configurations $C_1, C_2, \ldots$ which underapproximate $\texttt{Reach}(M, I)$, i.e. such that for all $i \in \mathbb{N}$, $C_i \subseteq \texttt{Reach}(M, I)$. This algorithm stops whenever it has computed a set of configurations $C_k = \texttt{Reach}(M, I)$. The advantage of this algorithm is that it aims at computing exactly the set $\texttt{Reach}(M, I)$ but it may also not terminate — for instance if the set $\texttt{Reach}(M, I)$ is not a Presburger definable set. Nevertheless as shown in [10], in many practical cases the Algorithm succeeds in computing the reachability set. This algorithm has been implemented in the tool FAST [5].

The second method introduced in [11] computes, using abstract interpretation [12], a set C which overapproximates $\texttt{Reach}(M, I)$ (i.e. $\texttt{Reach}(M, I) \subseteq C$). In opposite to the previous method, this algorithm always terminate but is also less accurate. In fact, to decide the reachability of a control state q from an initial configuration in I: if there is no configuration containing q in the computed set, then we are sure that q is not reachable from I in M. Otherwise if q appears in C then it is not possible to know whether q is reachable or whether it appears in the computed set because of the overapproximation. The tool ASPIC [7] implements this second method.

B. From programs with lists to counter machines

In [13], the authors have proposed a model to represent the behavior of sequential programs manipulating single-linked lists. The main idea is to abstract the contents of the cells of the linked lists and to represent the memory heap as a finite graph in which each node has at most one successor. There is furthermore a special node in this graph which characterizes the NULL pointer. These graphs are defined as follows.

Definition 2: Given a finite set V of pointer variables, a memory graph over V is a labelled graph $G = (N, next, l)$ such that:

- N is a finite set of nodes such that $N \cap \{\texttt{NULL}\} = \emptyset$;
- $next : N \rightarrow N \cup \{\texttt{NULL}\}$ is a partial function assigning to each node its successor;
- $l : V \rightarrow N \cup \{\texttt{NULL}\}$ is a partial function assigning a node to each pointer variable;
- for all nodes $n \in N$, there is $v \in V$ and $i \in \mathbb{N}$ such that $n = next^i(l(v))$

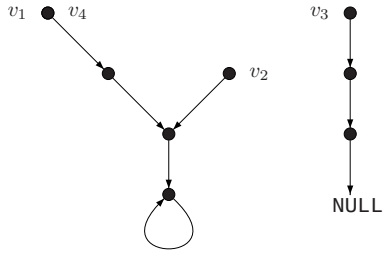

Figure 1. A memory graph

Note that the last condition in Definition 2 ensures that all the nodes in the memory graph can be reached from a node labelled with a pointer variable, in other words we do not allow memory leak in the considered graphs. Furthermore the function l is partial because some of the pointer variables might refer to an undefined location in the memory. We denote by $\mathcal{G}_V$ the set of memory graphs over V. In order to characterize the behaviors of programs working over single-linked lists, we will consider models labelled with guards and actions to be performed over such memory graphs. Given a set of pointer variables V, we first define the set of pointer guards g over V by the following grammar:

$$g \quad ::= \quad v == \texttt{NULL} \mid v == v' \mid \neg g \mid g \wedge g$$

where $v, v' \in V$. We denote by $Guard(V)$ this set of pointer guards. Given a memory graph $G \in \mathcal{G}_V$ and a pointer guard g, we write $G \models g$ if the conditions expressed by the guard are satisfied by the graph. We now define a set of pointer actions a over V as follows:

$$\begin{array}{lll} a & ::= & v := e \mid v.next := e \mid v := malloc \mid free(v) \\ e & ::= & NULL \mid v \mid v.next \end{array}$$

where $v \in V$. We denote by $Action(V)$ this set of pointer actions. Given a pointer action a, we define the function $[\![a]\!] : \mathcal{G}_V \mapsto \mathcal{G}_V \cup \{SegF, MemLeak\}$ which describes the effect of a on a given memory graph. Note that the effect of an action can be an error of type $SegF$, i.e. a segmentation fault, or $MemLeak$, i.e. a memory leak. In the sequel, we will denote $\mathcal{G}_V^*$ the set $\mathcal{G}_V \cup \{SegF, MemLeak\}$. A formal definition of $[\![a]\!]$ can be found in [3], we only provide here some informal explanations.

Figure 1 gives an example of a memory graph G over the set of variables $\{v_1, v_2, v_3, v_4\}$. This memory graph satisfies $\neg(v_1 == v_2)$, since the variables v_1 and v_2 are associated to two different nodes, and also for instance $\neg(v_3 == \texttt{NULL})$ since the variable v_3 is not associated to the special node NULL. The following gives some examples of actions effects on this graph. The action $v_3 := v_2$ would lead to $MemLeak$; indeed, after having moved the variable v_3 to the node pointed to by v_2, the list reachable by v_3 is subsequently no longer reachable by any variable. This situation is called a memory leak. Also, the action $v_4 := v_3.next$ moves the variable v_4 to the successor in the graph of the node labelled by v_3. Concerning the action $v_4 := malloc$ its effect would be to add to the graph a disconnected node labelled with the variable v_4 while the action $free(v_4)$ would simply delete from the graph the label v_4. We can now define our model to represent programs which manipulate single-linked lists.

Definition 3: A pointer machine PM over a (finite) set of pointer variables V is a tuple $\langle Q, E \rangle$ where:

- Q is a finite set of control states;
- $E \subseteq Q \times Guard(V) \times Action(V) \times Q$ is a finite set of edges, each of which being labelled with a guard and an action over V.

The semantics of a pointer machine PM is given by its associated transition system $TS(PM) = \langle Q \times \mathcal{G}_V^*, \rightarrow \rangle$ where $Q \times \mathcal{G}_V^*$ is the set of configurations and $\rightarrow$ is the transition relation defined as follows: for all $(q, G), (q', G') \in Q \times \mathcal{G}_V^*$, $(q, G) \rightarrow (q', G')$ iff there exists $(q, g, q, q') \in E$ such that $G \models g$ and $[\![a]\!](G) = G'$. As for counter machines above, we will also use the notations $\rightarrow^*$ and $\texttt{Reach}(PM, I)$ with $I \subseteq Q \times \mathcal{G}_V^*$ for pointer machines. The safety problem we are interested in can then be defined as follows:

Input: A pointer machine PM and a set of initial configurations I;
Output: Do we have $\{(q, SegF), (q, MemLeak) \mid q \in Q\} \cap \texttt{Reach}(PM, I) = \emptyset$?

As mentioned in [13] this problem is undecidable, however in [14] and [4] a method has been proposed which allows to use techniques developed for the analysis of counter machines in order to verify pointer machines. This method relies on a reduction from the safety problem for pointer machines to the reachability problem for counter machines. In fact, these works have proved that given a pointer machine $PM = (Q, E)$ and an initial configuration $c_0 = (q_0, G_0)$ of M it is possible to build a counter machine $M' = (Q', E')$ equipped with two special states q_{SegF} and $q_{MemLeak}$ in Q and an initial configuration $c'_0 = (q'_0, \mathbf{v}_0)$ of M' such that an execution of machine PM leaving from c_0 will lead to a segmentation fault [resp. a memory leak] **if and only if** the control state q_{SegF} [resp. $q_{MemLeak}$] is reachable in M' from the initial configuration c'_0. Note that as discussed in [14] and [15] this construction can be extended to some special infinite sets of configurations for the pointer machine, obtaining then Presburger definable sets of configurations for the counter machines. The verification method we have used then follows two main steps:

1) Translation of the pointer machine into the corresponding counter machine;
2) Reachability analysis of the counter machine.

We stress the fact that already the first step provides some pieces of information concerning the safety problem for pointer machines. In fact it can be the case that after having built the corresponding counter machine, one of the special control state q_{SegF} [resp. $q_{MemLeak}$] is not connected to the

control state q'_0 of the initial configuration, then we are sure that the pointer machine do not perform any segmentation fault [resp. any memory leak].

III. TOOL

We give here a presentation of the tool TOPICS (Translation of Programs Into Counter Systems) [2] which is founded on the framework introduced in section II, and implements the translation from pointer machines to counter machines in order to verify the behavior of programs manipulating dynamically the memory heap.

A. Features of TOPICS

TOPICS is a tool written in JAVA which takes in input a program written into a syntactic subset of the C programming language. The programs which can be analyzed by TOPICS manipulate integers variables, bounded size arrays of integers, single-linked lists and bounded size arrays of linked lists. Function calls (excluding recursive calls) can also be used. The use of recursion is forbidden in order to ensure that, given a program and the name of a function appearing in the program, a unique control flow graph can be built. This graph can then be seen as a finite automaton, each edge of which is labelled with a guard and an action on the different variables (integers, arrays, lists) manipulated by the program. Hence, TOPICS translates this control flow graph into a counter machine, partly in using the translation from pointer machines to counter machines. In order to deal with finite arrays, each cell of an array is associated to a variable[2], and the translation can then be performed without modifying the actions on integer variables. TOPICS seeks for the four following categories of faults:

1) Memory Leak;
2) Segmentation Fault;
3) Out of bound errors which happen when the program manipulates indexes of an array greater than the size of the array;
4) Testing of undefined variable errors which happen when there is an instruction of the program which tests variables that have not been previously initialized.

For each of these categories of faults, identification can be reduced to a reachability problem over a translation of the program into counter machine. Descriptions of translated counter machines are written by TOPICS into files using the input syntax of the tools FAST and ASPIC which can then be run to solve the reachability problems.

B. Syntax of TOPICS

As said previously, the program syntax accepted by TOPICS is a syntactic subset of the C programming language. A complete description of the syntax can be found in the User's manual of the tool which is available on the web page [2]. In reality, a feature which does not exist in C has been added to the syntax of TOPICS, in order for this tool to handle non deterministic branching. The programs given in input of TOPICS can actually make use of an additional keyword any in conditional statements.

[2]For arrays of integers [resp. of lists], integer [resp. pointer] variables are used.

```
typedef struct List {
  struct List *next;
}* Liste;
void main(){
  Liste y,z;
  y=NULL;
  while(any){
    z=malloc(sizeof(struct List));
    z->next=y;
    y=z;
  }
}
```

Figure 2. A program written in the syntax of TOPICS

For instance, let us consider the program in Figure 2. In this program, the function main builds an acyclic single-linked list of a non-deterministically chosen size. In the conditional statement of the while loop, the keyword any is used to stipulate that this loop will eventually terminate after a finite arbitrary number of iterations.

This is actually a nice feature of the tool because it allows users to give directly in input the C programs they want to verify, once having abstracted away parts which don't strictly follow the syntax of TOPICS — non conformance to this syntax is automatically identified by the underlying parser, and notified to the user. Note that if we use the keyword true (or 1) instead of any in the example of Figure 2, then the semantics associated to this program is different because in this case the loop never terminates.

Apart from the keyword any, the syntax accepted by TOPICS is exactly the one of C programs (i.e. it is compatible with any standard C compiler).

C. Using TOPICS

In order to verify a program with TOPICS, the user gives the source code as input to the tool and specifies which function of the program he wishes to analyze. With these data TOPICS performs the translation into counter machines and produces files for ASPIC and FAST. As mentioned in section II-B, at the translation stage the tool can already prove the absence of faults in some cases. Alternatively, such outputs also inform the user that some faults may exist.

Specifically, if TOPICS returns a message saying that the analyzed function is free of Segmentation Fault then there is no need to analyze the produced counter machine to detect such an error, on the other hand if it says that there may be a Segmentation Fault, then the user has to perform a reachability analysis of the counter machine using the files produced by TOPICS. Note that TOPICS produces one input file for the tool FAST but as many files as there might be errors for the tool ASPIC. This is due to the syntax of ASPIC, which does not allow users to describe complex sets

(as union of control states) to be tested for reachability. Some experimental results on classical programs manipulating single-linked lists can be found on the web page of the tool [2]; the produced counter machines for these programs are also provided.

IV. Practical aspects

To support advanced features of digital technologies for monitoring, fault detection and diagnosis of equipment and systems (cf. section I), I&C architectures including digital networks for inter- and intra-system data communication are currently being proposed. This section presents practical aspects of formal verification applied to software involved in digital communication.

A. Software under analysis

The software under analysis is a preliminary (consequently not supposed as error-free) version of a "network spy" application which is embedded in a network monitoring system for local industrial networks. The main functionality of the network spy is to capture OSI frames exchanged between non classified applications on local industrial networks. For the purpose of the experimentation reported in this paper, focus has been placed on a multi-threaded program making use of complex programming mechanisms such as dynamic allocation of memory, circular buffers, synchronization by events, socket communication, and shared memory. The size of this software application is ≈ 7000 lines of C code.

Specifically, the software architecture is composed of six threads which are synchronized by events. The main processings of the analyzer, as sketched above and at Figure 3, are performed incrementally in four threads. Each of these four threads contribute at one stage of putting application messages together again, starting from network frames and progressively filtering out inappropriate, incorrect or redundant frames. Collected information at each stage is stored in dynamically allocated variables or copied into circular buffers. Typically, each circular buffer is shared between two threads, where one is writing and the other one is reading.

Figure 3 also illustrates the symmetry in software architecture, where each stage of the incremental processing is realized by one thread using data from the previous stage and eventually writing data for use at the subsequent stage. Concerning management of memory, the symmetry still holds where each thread allocates cells to store extracted data locally and temporarily; once the processing is achieved, filtered data is copied in global variables and buffers, and the temporary cells are released. Dynamic memory management is thus performed intra-thread (i.e. for a given dynamic variable, allocation and release are both performed by the same thread), whereas shared variables and circular buffers are indeed statically allocated at initialization phase.

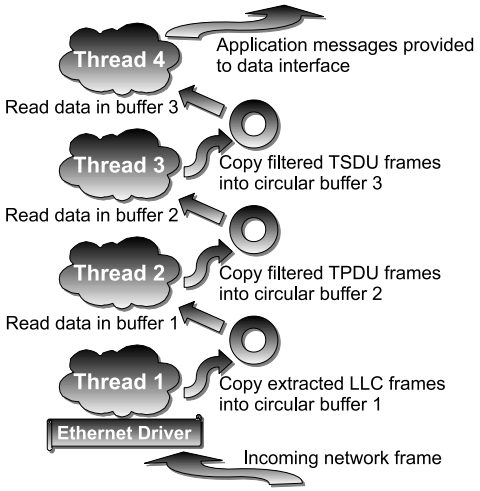

Figure 3. Overview of software architecture

Typical data structures involved in dynamic allocation are of size several hundreds, and are formed of nested arrays and C-structures — fields, each of which can have basic or array types, where "basic" types can be `int`, `char`, `char*`, etc. For instance in Thread 3, dynamic allocation and release can occur both on basic fields, and on nested array fields, at each index of an array.

B. Models

From original source code, models have been extracted according to the syntax accepted by `TOPICS` (cf. section III-B). This activity mainly involved removing system calls that are not included in `TOPICS` syntax and other processings not relevant to memory management, and replacing selected conditional statements by the keyword `any` — typically for statements which depend on the software environment, e.g. actual contents of incoming network frames as in Figure 4. The main requirement implemented when extracting models is to keep them *conservatively* representative of the original program, with respect to memory management. That is to say, behaviors and features related to memory management in the original program are preserved in the corresponding model, e.g. reachability of memory management primitives, control flow.

C. Experimental results

This section explains how our verification approach has been firstly applied to a model of one of the threads, and then to a second model as a double-check (resp. Threads 3 and 2 in Figure 3).

1) Thread 3: The implementation of Thread 3 has been modeled according to the principles mentioned in section IV-B, resulting in a compilable `C` file (≈ 500 lines). Figure 4 shows typical models for the parts of code where memory is allocated or released. As mentioned in section IV-A, dynamic memory management in this program

```
// Typical model excerpt: allocation
  if(any){ /* condition depending on actual data
       received in frames */
    if (info[index] == NULL){
      tmp = malloc(sizeof(struct cel)); /* type used
          for modeling memory allocation */
      info[index] = tmp;
      if (info[index] == NULL)
      return SPYKO; // modeling of memory overflow
    }
    return SPYOK; /* successful extraction of
        information */
  }

// Typical model excerpt: release
  if (info[index]!= NULL) {
    tmp=info[index];
    free(tmp);
  }
  info[index] = NULL;
```

Figure 4. Typical model excerpts: allocation and release

```
Region NoSegF:={!state=SegF};
Region NoOOB:={!state=OOBound};
Region NoUndef:={!state=Undef};
Region Reach:=post*(init,t,3);
if(subSet(Reach,NoMemLeak))
  then print("There is no Memory Leak.");
  else print("There is a Memory Leak.");
endif
if(subSet(Reach,NoSegF))
  then print("There is no Segmentation Fault.");
[...]
```

Figure 5. Input to FAST for Thread 3 (excerpt)

involves sizable composite arrays and data structures. With the original sizes, computations by TOPICS are overly time-consuming. The chosen workaround is to define a smaller size for these structures. Again, the justifying argument is symmetry: considering that allocation and release of memory follow the same program paths for any row in an array, the presence or absence of a fault related to memory management is independent of the absolute size of arrays. This statement has been verified by checking that results of analyses remain unchanged when array sizes vary.

Given the resulting model of Thread 3, TOPICS generates a counter machine and outputs it in files, whenever there is a category of faults for which the tool in non conclusive. Output files with `.fst` extension can be given as input to FAST and ASPIC tools. Figure 6 shows the resulting information displayed by TOPICS on the terminal window. TOPICS indicates it has proved the absence of memory leak in Thread 3. For other categories of faults handled by TOPICS, the verdict is not conclusive and `.fst` files are accordingly generated ($\approx$ 900 lines), allowing the user to verify the property of freedom from such intrinsic faults, with FAST or ASPIC.

We recall that given a `.fst` file, FAST can be used to verify several reachability properties, provided that the file defines a counter machine and several "regions" of interest — each region associated to a fault category. Here, the file `Thread3.fst` output by TOPICS contains the counter machine translated from Thread 3 model and the

```
$ java -jar TOPICS.jar modThread3.c srcThread3
       + NO MEMORY LEAK
       + MAYBE A SEGMENTATION FAULT
       + MAYBE AN OUT OF BOUND ERROR
       + MAYBE AN UNDEF ERROR

$ time fast srcThread3.fst
nb accelerations 17
"There is a Segmentation Fault."
"There is no Out of Bound error."
"There is no Undef error."
user    2739m51.210s

$ aspic srcThread3_SegF_aspic.fst
&& aspic srcThread3_OOBound_aspic.fst
&& aspic srcThread3_Undef_aspic.fst
   Result : Don't know
   Result : All the bad locations are unreachable
__bad_0 - > > empty(26)
   Result : All the bad locations are unreachable
__bad_0 - > > empty(26)
```

Figure 6. TOPICS, FAST and ASPIC results with Thread 3

lines shown in figure 5. The basic idea is to verify with FAST if error states are included in reachable states; results are shown in figure 6. On the other hand, the syntax of input files to ASPIC requires TOPICS to generate one file per fault category. For instance, the following line is inserted below the extracted counter machine in a `.fst` file, for the purpose of memory leak verification: `Region bad:=state=MemL;`. Figure 6 also shows the text output of ASPIC verification with Thread 3 model; computation time is here negligible.

The results obtained show the following. TOPICS proves the absence of memory leak; the generated files consequently don't handle this category of fault. Concerning "Out of bounds" and "Undef" errors, TOPICS is not conclusive, but fault freedom is proved by FAST, and confirmed by ASPIC. Concerning segmentation faults, FAST indicates it has identified one instance, whereas ASPIC is not conclusive. Typically, FAST is more accurate than ASPIC on this case study, while the computation time of FAST is considerably longer than that of ASPIC. These observations are consistent with the underlying formal techniques: computation of the *exact* set of configurations reachable from a given set of initial configurations [5], resp. accelerated linear relation analysis [7]. It is also noticeable that no contradiction appear between the results of the three tools.

Additionally, there is currently up to our knowledge no commercial tool available, for which formal verification of dynamic memory issues is claimed. Still, numerous tools are able of identifying dynamic memory faults (e.g., tools cited in section V). This case study was further analyzed with one of these tools, for which "Leak", "Null pointer dereference" and "Free null pointer" belong to the list of software faults susceptible to be identified. This analysis (computation time ≈ 1 minute) gives several verdicts to locate possible faults, including 1 "Leak", 1 "Free Null Pointer" and 4 "Null Pointer Dereference".

After code inspection at the associated line number,

the “Leak” verdict appears to be a false positive, consequently not compromising the correctness of TOPICS results (cf. Figure 6): this verdict is indeed associated to a normal allocation of a variable immediately after its declaration, as shown within the short extract of code below.

```
int *IndexAdd;
IndexAdd=malloc(k*sizeof(int));
```

Faults related to as “Null pointer dereference” and “Free null pointer” by the commercial tool used here are both denoted as “Segmentation Faults” by TOPICS. Code inspections at the respective lines for the four “Null pointer dereference” verdicts led us to identifyreal faults, due to the fact malloc might return a null pointer (in case no memory is available or some error prevented memory from being allocated), while there is no defensive test against the null value at these points of the program. Similarly, the “Free null pointer” verdict is visibly linked to the fact one of the functions of Thread 3 which allocates a buffer has an error case where the returned buffer is null, and a subsequent call to free is not protected by a defensive test. *NB.* It has been checked additionally that every identified fault indeed exists in the original program.

These observations lead us to write a corrected version of Thread 3 model, where the identified faults are removed. With this model, TOPICS proves the absence of segmentation fault: the output is similar to that of Figure 6, with +NO SEGMENTATION FAULT instead of +MAYBE A SEGMENTATION FAULT (achieving this result actually took a couple of iterations between commercial and academic tools). Finally, this second version of Thread 3 model was also analyzed with the same commercial tool. The resulting verdicts still indicate some possible faults under the category of ’segmentation fault’. These remaining verdicts appeared to be false positives after code inspection, for similar reason as earlier. Then we had got a confirmation of TOPICS results that freedom from segmentation fault had been reached.

2) Thread 2: The same methodology as explained above for Thread 3 has been applied to Thread 2. Experimental results with this model are shown in figures 7and 7. Analyzing Thread 2 model, TOPICS proves the absence of out-of-bounds access and, as expected, produces a counter machine for third-party tools to check the three other types of considered faults; NB. regions defined are similar to that of Figure 5, with NoMemLeak:=!state=MemL; instead of NoOOB:=!state=OOBound;. In each case, one instance is found by FAST, while ASPIC is not conclusive.

Observations made for Thread 3 about accuracy, computation time and consistency between the results of different tools still hold in the case of Thread 2. As earlier, this case study was further analyzed with the same commercial tool, which notably produces 4 “Leak”, 1 “Uninitialized Variable”, 2 “Free Null Pointer” and 3 “Null Pointer Dereference” verdicts. So far, the tool-based methodology applied

```
$ java -jar TOPICS.jar modThread2.c srcThread2
        + MAYBE A MEMORY LEAK
        + MAYBE A SEGMENTATION FAULT
        + NO OUT OF BOUND ERROR
        + MAYBE AN UNDEF ERROR

$ time fast srcThread2.fst
nb accelerations 111
"There is a Memory Leak."
"There is a Segmentation Fault."
"There is an Undef error."
user    2527m18.230s

$ aspic srcThread2_MemL_aspic.fst
&& aspic srcThread2_SegF_aspic.fst
&& aspic srcThread2_Undef_aspic.fst
   Result : Don't know
   Result : Don't know
   Result : Don't know
```

Figure 7. TOPICS, FAST and ASPIC results with Thread 2

to Thread 3 in the first part of section IV-C is also applicable in the case of Thread 2: firstly perform formal analysis, and then if needed, try and complement with external tools (e.g. commercial) and code inspection to locate possibly remaining faults, and finally iterate until demonstration of freedom from intrinsic faults among the considered categories. Further issues related to this methodology are discussed in section IV-D.

D. Lessons learned

The tool-based methodology and experiments reported in this paper show that memory safety can be formally proved in local processings of industrial software (e.g. analysis thread by thread). A version of TOPICS has also been developed in order to tackle concurrency more directly (with synchronous products). However, the techniques implemented for that matter are presently not efficient enough to scale to this case study (the combinatorial calculations can be handled on smaller case studies; experiments not reported in this paper). Designing a tool to this purpose is currently a theoretical and practical challenge. With this in mind, methods for modular thread analysis and interleaving reduction belong to the approaches worth of investigation.

Experiments performed have also shown that separate translation and analysis phases may lead to a situation where generated models are large, with a consequently time-consuming analysis with existing tools. Performance may then be significantly improved by on-the-fly analyzes in order to reduce the size of generated counter machines.

Finally, potential for improving the maturity of this approach particularly resides in automation of the model extraction phase briefly described in section IV-B, which practically requires substantial effort on real-size software.

The learnings above are providing guidance and aims in ongoing work, cf. section VI.

V. Related works and tools

Many tools and techniques to check safety properties on programs manipulating pointers have been developed

recently. In [16], the framework of predicate abstraction is used to manipulate boolean formulae representing the heap. In [17], the authors present a shape analysis method based on separation logics formulae to analyze programs manipulating singly-linked lists. Their method always terminates but might yield false alarms due to the over-approximation brought by the abstraction. Other methods have been proposed which use already existing model-checking techniques. For instance, in [18], the authors verify safety properties on programs using abstract regular model-checking and in [19] the authors propose to combine shape analysis and arithmetic analysis.

As seen in section IV, software analysis tools such as [20] can be useful to complement our tool-based formal verification methodology. Most of the main available software analysis tools — although practical and efficient in their main targeted application cases (e.g. supporting developers in improving code quality, robustness or security during the development process) — may however infer false negatives, i.e. actual faults that remain undetected within the categories of faults identifiable by the tool. Besides, some of the available commercial software analysis tools are designed to exclude false negatives, but to our knowledge, the categories of faults considered in this study are currently not in the scope of such tools.

The `Frama-C` toolset [21] is considered as a promising solution to go further on this methodology (see also section VI). The CMBC tool [22] could also be investigated on a case study, as a complementary approach.

VI. Conclusion

This paper reports on the development of a tool-based methodology and its application to industrial software. As an important conclusion, the considered approach has been shown to be applicable accurately to local processings of industrial software involved in control systems. Identified limitations (cf. section IV-D) are to be addressed in ongoing work. To achieve this, the VERIDYC project [23] will notably investigate modularization of formal analyzes, and build on `Frama-C` [21] capabilities, such as analyzing full `C` language, program slicing, and allowing the use of independently developed plug-in analyzers.

References

[1] N. Thuy and A. Ourghanlian, "Dependability assessment of safety-critical system software by static analysis methods," in *DSN*, 2003, pp. 75–79.

[2] `TOPICS`: Translation Of Programs Into Counter Systems. http://www.lsv.ens-cachan.fr/~sangnier/TOPICS/index.php.

[3] A. Sangnier, "Vérification de systèmes avec compteurs et pointeurs," Thèse de doctorat (PhD thesis), Laboratoire Spécification et Vérification, ENS Cachan, France, 2008.

[4] A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar, "Programs with lists are counter automata," in *CAV*, 2006, pp. 517–531.

[5] `FAST`: Fast Acceleration of Symbolic Transition systems. http://www.lsv.ens-cachan.fr/Software/fast.

[6] `TReX`: A Tool for Reachability Analysis of CompleX Systems. http://www.liafa.jussieu.fr/~sighirea/trex.

[7] `ASPIC`: Accelerated Symbolic Polyhedral Invariant Computation. http://laure.gonnord.org/pro/aspic/aspic.html.

[8] M. L. Minsky, *Computation: finite and infinite machines*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1967.

[9] A. Finkel and J. Leroux, "How to compose presburger-accelerations: Applications to broadcast protocols," in *FSTTCS*, 2002, pp. 145–156.

[10] S. Bardin, A. Finkel, and J. Leroux, "Faster acceleration of counter automata in practice," in *TACAS*, 2004, pp. 576–590.

[11] L. Gonnord and N. Halbwachs, "Combining widening and acceleration in linear relation analysis," in *SAS*, 2006, pp. 144–160.

[12] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL*, 1977, pp. 238–252.

[13] S. Bardin, A. Finkel, and D. Nowak, "Toward symbolic verification of programs handling pointers," in *AVIS*, 2004.

[14] S. Bardin, A. Finkel, É. Lozes, and A. Sangnier, "From pointer systems to counter systems using shape analysis," in *AVIS*, 2006.

[15] A. Finkel, É. Lozes, and A. Sangnier, "Towards model-checking programs with lists," in *ILC*, 2007, pp. 56–86.

[16] A. Podelski and T. Wies, "Boolean heaps," in *SAS'05*, ser. LNCS, vol. 3672. Springer, 2005, pp. 268–283.

[17] D. Distefano, P. W. O'Hearn, and H. Yang, "A local shape analysis based on separation logic." in *TACAS'06*, ser. LNCS, vol. 3920. Springer, 2006, pp. 287–302.

[18] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar, "Abstract regular tree model checking of complex dynamic data structures." in *SAS'06*, ser. LNCS, vol. 4134. Springer, 2006, pp. 52–70.

[19] S. Magill, J. Berdine, E. Clarke, and B. Cook, "Arithmetic strengthening of shape analyses based on separation logic." in *SAS'07*, ser. LNCS, vol. 4634. Springer, 2007, pp. 419–436.

[20] Software Analysis Tools, e.g. Grammatech CodeSonar, Coverity Static Analysis, Cppcheck, Fortify 360 Source Code Analyzer, Klocwork Insight, etc.

[21] Frama-C Software Analyzers — http://frama-c.com.

[22] CBMC Model Checker
http://www.cs.cmu.edu/~modelcheck/cbmc.

[23] VERIDYC project
http://www-verimag.imag.fr/VERIDYC.html.