Rigorous Evidence of Freedom from Concurrency Faults in Industrial Control Software

Richard Bonichon[1], Géraud Canet[1], Loïc Correnson[1], Eric Goubault[1], Emmanuel Haucourt[1], Michel Hirschowitz[1], Sébastien Labbé[2], and Samuel Mimram[1]

[1] CEA, LIST,
Gif-sur-Yvette, F-91191, France
firstname.lastname@cea.fr
[2] EDF Research & Development
6 quai Watier, Chatou, F-78401, France
firstname.lastname@edf.fr

Abstract. In the power generation industry, digital control systems may play an important role in plant safety. Thus, these systems are the object of rigorous analyzes and safety assessments. In particular, the quality, correctness and dependability of control systems software need to be justified. This paper reports on the development of a tool-based methodology to address the demonstration of freedom from intrinsic software faults related to concurrency and synchronization, and its practical application to an industrial control software case study. We describe the underlying theoretical foundations, the main mechanisms involved in the tools and the main results and lessons learned from this work. An important conclusion of the paper is that the used verification techniques and tools scale efficiently and accurately to industrial control system software, which is a major requirement for real-life safety assessments.

Keywords: Digital control systems, software dependability, formal verification, concurrency, deadlock.

1 Introduction — Intrinsic Software Faults

Dependability assessment of digital control systems require elements from control systems designers in order to establish the excellence of production; e.g. evidence of systematic, fully documented and reviewable engineering process, quality assurance, test and simulation at different stages of development, operational experience and demonstration of conformity to applicable standards.

Complementary measures may be taken in order to demonstrate properties, and provide rigorous evidence of the freedom from postulated categories of faults. Considered faults include *intrinsic software faults*, i.e. faults in a software design that can be identified independently of functional specifications.

F. Flammini, S. Bologna, and V. Vittorini (Eds.): SAFECOMP 2011, LNCS 6894, pp. 85–98, 2011.

Technical studies and surveys performed in recent years have led to consider three main categories of software intrinsic faults as relevant to this domain, i.e. intrinsic faults that might be postulated in software important to availability. The taxonomy we rely on [24] should be comprehensive, though not exhaustive. It has been established by using various sources:

- Experience gained in software formal verification at EDF (e.g. [28] and internal technical reports) and other institutions or companies sharing similar interest in such methods [1, 15, 25];
- Community lists of software faults, like CWE [12];
- Lists of addressed faults published by software analysis tool vendors.

The mentioned categories of faults are the following:

- Faults in concurrency and synchronization (detailed below);
- Faults in dynamic handling of memory, namely: memory leaks, segmentation faults and other memory-related undesirable behaviors;
- Other, more "basic", faults such as divisions by zero, out-of-bounds array access, use of uninitialized variables, other numeric manipulation errors, etc.

Depending on the importance to safety of a system software, some of the above categories of faults can be ruled out by design. For instance in critical software, dynamic management of the memory is usually not allowed by design rules. In this paper, only the concurrency aspects are developed; specifically when considering these aspects, it is of interest to demonstrate the absence of the following intrinsic faults:

- Resource starvation such as deadlocks and livelocks, e.g. points where no further progress is possible for a given program run;
- Non-determinism or race condition, e.g. points where a program may behave differently given the same inputs (because side effects depend upon synchronization);
- Incorrect protection of shared resources, e.g. concurrent and non protected accesses to a shared variable;
- Incorrect handling of priorities;
- Unexpectedly unreachable program states, e.g. no path may lead to a given state while it was supposed to be reachable.

In some cases, intrinsic faults are unlikely to be detected via classical V&V methods[1], for instance faults that might be triggered in specific configurations of variables (e.g. division by zero, arithmetic overflow), or after long runs (e.g. overrun in a very large buffer, out-of-memory error due to a small leak), etc.

Tool-based analytic approaches to systematically identify intrinsic faults preferably rely on formal verification techniques. The gained confidence can then be used in higher level assessments, e.g. to support claims about I&C software contribution to the reliability of a safety function, or to alleviate concerns regarding digital Common Cause Failures.

[1] Functional correctness still needs to be addressed with appropriate approaches, e.g. functional testing, theorem proving or simulation.

2 Tool-Based Methodology — Outline

This paper reports on the development of a tool-based methodology to demonstrate the freedom from intrinsic software faults related to concurrency in industrial control software. The following two related projects are involved (both projects have eponymous tools):

- `MIEL`: Interactive model extraction from software — Analyze software written in `C` language in order to facilitate code understanding and extract representative models to be analyzed by third-party tools;
- `ALCOOL`: Analysis of coordination in concurrent software — Develop a theory, algorithms and a static analysis tool with the ability to verify synchronization properties (particularly, freedom from intrinsic faults) in complex software.

The tool-based methodology developed in these two projects aims at analyzing software that can be found in digital control systems either safety related, or important to availability, in power plants. Compared to the most critical software parts, complex programming mechanisms might be more freely used in such software, e.g. concurrent interactions and synchronization.

On the other hand, the dependability requirements for such systems are high; then the system lifecycle must (at least partly) establish required properties, e.g. predictability. In particular, design measures are generally applied to effectively reduce the complexity and restrict the set of potential vulnerabilities. Memory allocation, task and synchronization resources creation are indeed to be performed *only* during a dedicated initialization phase. Then, the behavior in normal operation is intended mostly cyclic and steady. For instance, an iteration of a loop is expected to "know" as little as possible from previous iterations. Also, communication and synchronization are expected to be restricted to the necessary. The presented tool-based methodology is intended to take advantage of those characteristics, to provide rigorous verification tools with high efficiency.

The rest of the paper is structured as follows. The tools `MIEL` and `ALCOOL`, together with their theoretical foundations are described in Sec. 3. An industrial software case study and practical experiments are described in Sec. 4. Some related works are presented in Sec. 5. Finally, the main results, lessons learned and perspectives to this work are synthesized in Sec. 6.

3 Theoretical Framework and Tools

3.1 Static Analysis for Model Extraction and the `MIEL` Tool

The `MIEL` tool is a model extractor for `C` programs. It runs as a plug-in of the `Frama-C` static analysis platform [11, 17], which is dedicated to the analysis of software source code written in the `C` programming language.

The main requirement for models extracted by `MIEL` is to be *conservatively* representative with respect to the specified point of interest, i.e. behaviors related to a specific aspect of interest in the original program must be included in the behaviors denoted by the corresponding model. Various intrinsic aspects of

```
function pthread_create
  of type thread creation
  has arg 3 of type
          function name
```

```
function pthread_mutex_unlock of type release
  has arg 1 of type semaphore

function pthread_mutex_lock of type lock
  has arg 1 of type semaphore
```

Fig. 1. Excerpts of `MIEL` description file: `POSIX` thread creation (left), locks (right)

software can be of interest, including calls to memory management, concurrency, synchronization or communication primitives.

Accordingly, extracted sets of information may then be provided as models to dedicated external tools, with the purpose of demonstrating or refuting properties. The `MIEL` tool is primarily geared toward extracting models from concurrent programs; its algorithm applies also to sequential programs.

Model extraction is based on a description file, which indicates the points of interests in the program: a list of functions, together with a signature (using types known by `MIEL`).[2] Only arguments of interest should be specified in a signature. For example, a user interested in threads or processes created in a program using `POSIX` primitives, might write the description of Fig. 1 (left). This declares to the analyzer, first, that every occurrence of thread creation must appear in the model, and second, that the starting functions of threads are given in the third argument of pthread_create. Now, considering synchronization faults, e.g. deadlocks, the model must also encompass all events where locks are taken and released. Fig. 1 (right) suggests a suitable description.

The `MIEL` tool implements a syntactic model extraction algorithm, which is sound for a specific class of software where the interesting function identifiers are syntactically reachable. That is to say, the functions of interest must not be aliased, and resource requests of interest must be syntactically distinct. We will see in the following how these soundness requirements are consistent with the characteristics of the targeted systems software.

Considering the typical design features of the targeted classes of software (safety related or important to availability, cf. Sec. 2), it is expected that synchronization or memory management primitives can be easily identified (no aliases on these function names). It is also expected that loop unrolling is sufficient to make sure accesses to different synchronization or memory resources (threads, mutexes, semaphores, memory cells, etc.) can be syntactically distinguished. This can be achieved through code annotation within `Frama-C`. Detailed examples of the annotations needed for the presented case study are given in Sec. 4.2.

3.2 Geometric Semantics for Concurrency Analysis, `ALCOOL` Tool

The `ALCOOL` tool is based on the directed algebraic topology of cubical areas. Roughly speaking, a cubical area of dimension $n \in \mathbb{N}$ is a finite union of hyperrectangles of dimension $n \in \mathbb{N}$ (i.e. finite products of n non empty intervals of the real line $\mathbb{R}$). As mathematical objects, the cubical areas enjoy a structure which

[2] The types used to classify functions include: thread creation, process creation, lock, release, delay, priority, interrupt handler, ...

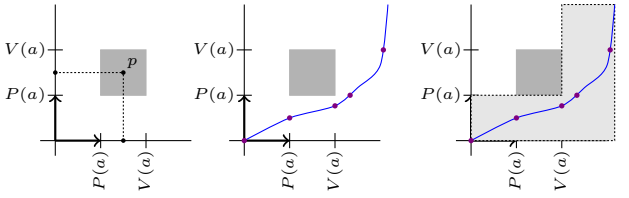

Fig. 2. The forbidden square

is implemented as a library used by the ALCOOL tool. Basically it takes as input a program written in a specific input language (in practical cases, models are preferably automatically generated by MIEL) and produces a geometric model which is a cubical area. From this model, ALCOOL then can identify forbidden states, deadlocks, unreachable states, critical sections. Using an algorithm which performs the "prime" decomposition of a cubical area [3], ALCOOL can also find the subgroups of processes which actually run independently from each others.

The PV language was introduced by E.W. Dijkstra to illustrate the problems which may arise when designing parallel programs [13]. In the original version, P and V respectively stand for the dutch words "pakken" (take) and "vrijlaten" (release). The processes indeed share a pool of resources, and each of them may request the authorization to access a resource a by executing the instruction $P(a)$. If a is available, then it is granted to the process until it releases it by executing the instruction $V(a)$. Otherwise, the process stops and waits until the resource is dropped by the previous holder.

On the practical side, the development of the ALCOOL tool was originally driven by the need for dealing with industrial C programs. The ALCOOL input language (called PV in the following), therefore contains a broader set of concurrency primitives, including features from POSIX and VxWorks.

We now illustrate the abilities of ALCOOL through some examples. Consider the sequential process $\pi := P(a).V(a)$, where a is a resource that can be held by a single process at a time (a mutex). Then run two copies of π simultaneously: $\pi|\pi$ (cf. Fig. 2). Generally, each running process is associated with a dimension, therefore the geometric model is here 2-dimensional. The point p represents a state in which both instruction pointers, which are the projections of p, lie between $P(a)$ and $V(a)$. That is, in this state both copies of the process π hold the resource a, which is by definition forbidden (not feasible). The geometric model is thus $\mathbb{R}^2 \backslash [1, 2[^2$: all the points in the gray square are forbidden.

The fact intervals are open or closed depends on whether an instruction is executed exactly when it is reached by the instruction pointer. In this framework, the program execution traces correspond to increasing continuous paths in the geometric model (cf. Fig. 2). The library dealing with cubical areas allows ALCOOL to identify these, all represented by the same cubical area.

For the next example in Fig. 3 (left), we consider two processes $\pi_0 := P(a).P(b).V(b).V(a)$ and $\pi_1 := P(b).P(a).V(a).V(b)$ running concurrently. We have two resources a and b each of which generates a forbidden rectangle. In this

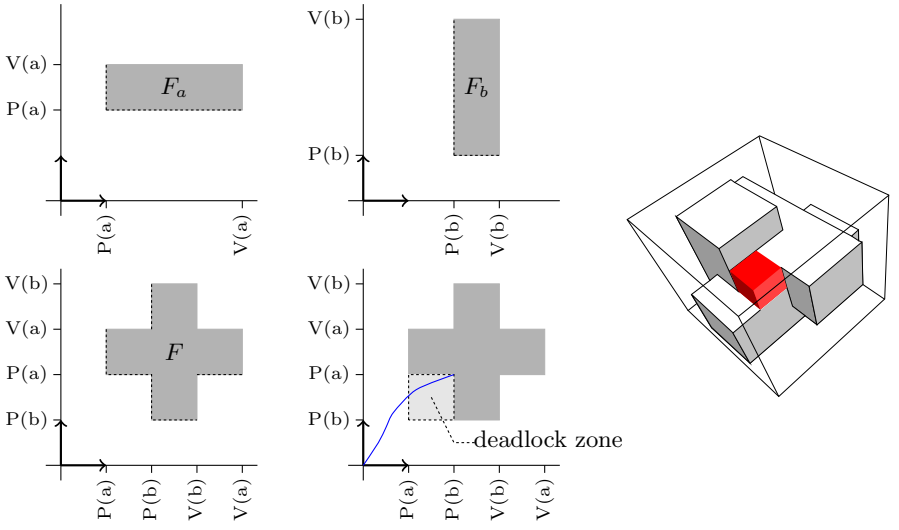

Fig. 3. Deadlock examples: The Swiss flag (left), The 3 dining philosophers (right)

example we have a potential deadlock: if the first process takes the resource a while the second one takes b, then both won't be able to progress further.

The 3-dimensional models arise from the study of programs made of 3 processes, such as the well-known 3 dining philosophers, cf. Fig. 3 (right). In this picture, the central (red) cube represents the deadlock zone.

As we shall see, `ALCOOL` can also perform the factorization of a `PV` program from its geometric model. In the next example we introduce counting semaphores: resources that can be shared by more than 2 processes. We suppose a and b are mutexes and c is a 3-semaphore, meaning c can be held by 2 processes at a time but not 3. Then we consider the following processes:

$$\pi_a := P(a).P(c).V(c).V(a)$$
$$\pi_b := P(b).P(c).V(c).V(b)$$

Then we consider the `PV` program $\pi_a|\pi_b|\pi_a|\pi_b$. A handmade analysis reveals that the semaphore c is in fact useless. The program can indeed be split into two groups of processes $\{1,3\}$ and $\{2,4\}$. Each group cannot hold more than one occurrence of the c resource, so it cannot run out of stock. The `ALCOOL` tool detects this situation by performing an algebraic factorization, proving that the geometric model of the program can be written as a 2-fold Cartesian product:

$$\Big([0,1[\times[0,\infty[\ \cup\ [4,\infty[\times[0,\infty[\ \cup\ [0,\infty[\times[0,1[\ \cup\ [0,\infty[\times[4,\infty[\Big)^2$$

From a theoretical point of view, a cubical area can be written as a Cartesian product in a unique way (compare with natural numbers and prime numbers). This feature is extremely important since it allows, when the decomposition of

the geometric model is not trivial, to split the analysis of a program into the analysis of several simpler subprograms.

The cubical area library lies upon some facts we now state. In the sequel, a cubical area should be understood as a finite union of hyperrectangles.

- The intersection of finitely many cubical areas of dimension n is a cubical area of dimension n;
- The complement (in $\mathbb{R}^n$) of a cubical area of dimension n is a cubical area of dimension n;
- Any cubical area has a "normal form" given by the collection of all its maximal sub-hyperrectangles. A hyperrectangle contained in a cubical area X is said to be maximal (in X) when any strictly bigger hyperrectangle is not contained in X;
- The previous assertions over the n dimensional cubical areas are compatible with the action of the symmetric group $\mathfrak{S}_n$ (in other words the permutation of coordinates in a "coherent" way);
- There is a notion of directed homotopy so that each equivalence class of directed path over a cubical area is characterized by a cubical area X. Indeed, a path γ is in the class E_X if and only if its image $\{\gamma(t) \mid t \in [0,1]\}$ is contained in X.

More details about the way theoretical facts are exploited by `ALCOOL` may be found in [19]. The cubical areas are special cases of pospaces, they are introduced in [26] without any mention to their directed homotopy aspects. Gentle introductions to the latter can be found in [18] and [16]. An abstract treatment of Directed Algebraic Topology can be found in [20].

4 Case Study

This section reports on a practical application of the verification approach presented in Sec. 3. Section 4.1 briefly describes a real-world software unit, which is embedded in a programmable logic controller. Then, we explain in Sec. 4.2 how the verification framework and tools support rigorous evidence of freedom from synchronization faults in this software.

4.1 Industrial Control System Software

Software under analysis in the present case study is a processing unit of an industrial programmable logic controller (PLC) used in digital control systems.The source code is written in `C` language; its size is approximately 85 kloc, where 1 kloc = 1 000 lines of code (107 kloc including specific header files and 135 kloc including all header files).

This software is involved in processing inputs and outputs (interfacing control systems with sensors and actuators, including local processing of I/O data), handling network communications, self-monitoring and maintenance functions. For other low-level internal resource management and processing, the software relies

on a commercial real-time kernel threads API: for scheduling, handling of priority and interruption, creation and management of threads and communication resources such as semaphores and queues.

The following is a succinct description of the software architecture and nominal dynamic behavior. After kernel initialization, the main thread configures interruptions, connects interruption routines, and creates all the needed threads and communications resources (there are about 10 of each item created). Each thread is created with one of the following purposes: handling process inputs, screening and detecting state changes in I/O boards; handling time-controlled inputs and outputs and network communications; PLC configuration; update of redundant processing unit, self-checking, etc. These threads run concurrently until the end of the main thread. Shared data include queues, semaphores (some of which are gathered in arrays), some events and configuration values.

4.2 Verification of Synchronization Properties

Model Extraction. As explained in Sec. 3.1, users of `MIEL` need to check that each call to a function of interest in the source file can be syntactically seen as such in the source files. Concerning our case study, code inspection indicates that some parts need annotations for the abstracted model to be correct. As argued earlier, two conditions might invalidate that correction: aliases for synchronization primitives names, and calls to synchronization primitives nested within loops. In the former case, we have seen in Sec. 2 that relying on design considerations, it is expected that there is no alias on synchronization primitives names (we have also confirmed this assumption by code inspection). In the latter case, we rely on loop unrolling to extract a correct model.

More precisely, where semaphores are locked then unlocked, the encompassing loop has to be syntactically expanded by `Frama-C` before the analysis by `MIEL`, as in Fig. 4. Actually, every part of the original code that is both a remote parent of a call to a function of interest, and located within a loop will need to have its loop unrolled in this way. With the additional use of semantic constant folding (done by a `Frama-C` plug-in) the semaphores of Fig. 4 can be identified in the model without ambiguity (because syntactically different after constant folding).

After the initial phases of specifying the aspects of interest in the program — in this case: primitives related to semaphores, threads and message queues — and inserting the appropriate annotations, `MIEL` performs an automatic analysis. Fig. 4 shows the whole description file needed for the case study. No additional code modifications or annotations are needed.

Given the code snippet of Fig. 4 (right), `MIEL` yields a model sketched in Fig. 5. During the analysis, the entry points of threads are automatically found. Launching `MIEL` analysis of the case study files takes a few seconds on a recent Linux workstation. It yields a quite large model file: $\approx$ 2000 lines in the `PV` language (fifty times smaller than the original `C` code).

Verification. Whether programs are concurrent or not, the analysis of programs with loops is a problem known to be undecidable. Hence, `ALCOOL` analyzes are parametrized by the number of synchronization steps to be unrolled to find

```
function semTake of type lock
  has arg 2 of type delay
      arg 1 of type semaphore

function semGive of type release
  has arg 1 of type semaphore

function taskSpawn
  of type thread creation
  has arg 5 of type function name

function msgQSend of type send
  has arg 5 of type priority
      arg 4 of type delay
      arg 1 of type queue

function msgQReceive of type receive
  has arg 4 of type delay
      arg 1 of type queue
```

```
/*@loop pragma UNROLL 10; */
for( i = 0 ; i < 10 ; i++ ) {
    if( message[i] ) {
    if(semTake(sem_array[i],
         NO_WAIT))
    {
      ...
      message[i] = 0;
      new__frame[i] = ... ;
      semGive(sem_array[i]);
    }
```

Fig. 4. Inputs: `MIEL` description file for the case study (left), `Frama-C` annotation for loop expansion (right)

```
exec_loop =
(((P(sem_array_0).V(sem_array_0))
  + ...) + ...).
(((P(sem_array_1).V(sem_array_1))
  + ...) + ...).
...
(((P(sem_array_9).V(sem_array_9))
  + ...) + ...).
```

```
Geometric Model (forbidden area):
[0, +∞[^2 × [1, +∞[ × [0, +∞[^10
∪ [0, +∞[^3 × [1, +∞[ × [0, +∞[^9
∪ [0, +∞[^7 × [1, +∞[ × [0, +∞[^5
∪ [0, +∞[^9 × [1, +∞[ × [0, +∞[^3
∪ [0, +∞[^10 × [1, +∞[ × [0, +∞[^2
∪ [0, +∞[^11 × [1, +∞[^2

Deadlock Attractor: ∅
Critical sections: No conflict.
Unreachable area: ∅
```

Fig. 5. Outputs: Part of the `PV` model for the code of Fig. 4 (left), `ALCOOL` output on the case study (right)

possible intrinsic faults in concurrency. During the analysis, `ALCOOL` priorly chooses branches of "if then else" on which resource primitives appear. The idea is to focus the verification effort on scenarios that might lead to a synchronization fault. The results are displayed as in Fig. 5. Here, the geometric model has dimension 13 (no graphical representation available). Back to the definition of intrinsic faults related to synchronization and concurrency in Sec. 1, the results displayed by `ALCOOL` give evidence of freedom from deadlocks (item "*Deadlock attractor*"), incorrect protection of shared resources (item "*Critical sections*"), and unexpectedly unreachable program states (item "*Unreachable area*"). Non-determinism and priorities are currently not supported.

When the depth of analysis varies, the qualitative results remain the same as in Fig. 5. As shown in Fig. 6, the computation time grows non-linearly with the depth of analysis. The analysis takes less than one hour at depth 10^6, and less than one minute at depth 10^5. Depth 5×10^6 can be practically reached (around 17 hours; computation times obtained on a Z600 Linux workstation).

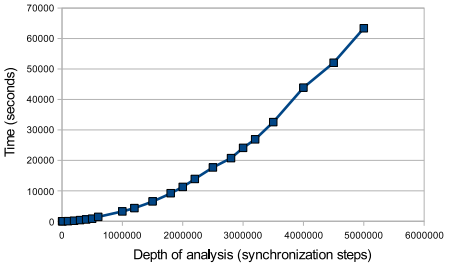

Fig. 6. ALCOOL: Computation time *versus* depth of analysis

We recall that depth of analysis is here expressed in terms of calls to synchronization primitives; the associated concrete traces in the original program are consequently far longer, referring to the 1 : 50 ratio between model and source code (cf. Sec. 4.2, part "model extraction").

As we have seen, analyzes with ALCOOL can be practically performed at a significant depth. The significance is consolidated under the assumption that loop iterations have limited memory from previous iterations (cf. Sec. 2). More generally, these results confirm the considerations in Sec. 2 about how the characteristics of the addressed classes of programs can be helpful when designing or using formal analysis methods. Code inspection in this software case study indeed shows that synchronization primitives are moderately used (few occurrences of resource requirements, limited interactions between tasks...). The resulting concurrent model is quite simple, given the size of the software, and compared to what absolute concurrency may allow.

In cases where ALCOOL finds a synchronization fault, the variables behaviors have to be thoroughly studied in order to check whether the execution traces that lead to the fault are in fact feasible or not.

No intrinsic fault related to concurrency has been found in the original source code; this outcome is likely when considering high integrity software. In the remainder of this section, we will see how voluntarily inserted faults can be detected by the tools (the instance presented is a deadlock).

Voluntary Insertion of a Deadlock. This short presentation is meant as an example; the fault detection ability of the tools is based on the framework in Sec. 3 and is validated elsewhere (against sets of sample codes). We actually insert three threads in the original annotated code of the case study. Fig. 7 (left) shows the additional code accordingly. These additional threads implement three dining philosophers in a configuration known to lead to starvation. Fig. 7 (right) shows a snippet of the PV model generated by MIEL, focused on the additional threads. Fig. 8 shows that ALCOOL indeed finds the deadlock induced by the three additional threads.

```
void Ph1(void) {
  semTake(mutex_a, WAIT_FOREVER) ;
  semTake(mutex_b, WAIT_FOREVER) ;
  semGive(mutex_a) ;
  semGive(mutex_b) ;
}
void Ph2(void) {
  semTake(mutex_b, WAIT_FOREVER) ;
  semTake(mutex_c, WAIT_FOREVER) ;
  semGive(mutex_b) ;
  semGive(mutex_c) ;
}
void Ph3(void) {
  semTake(mutex_c, WAIT_FOREVER) ;
  semTake(mutex_a, WAIT_FOREVER) ;
  semGive(mutex_c) ;
  semGive(mutex_a) ;
}

static void create_tasks(void) {
  mutex_a = semBCreate() ;
  mutex_b = semBCreate() ;
  mutex_c = semBCreate();

  taskSpawn("Russell",90,
       VX_NO_STACK_FILL,1000,Ph1,
       0,0,0,0,0,0,0,0,0,0);
  taskSpawn("Goedel",90,
       VX_NO_STACK_FILL,1000,Ph2,
       0,0,0,0,0,0,0,0,0,0);
  taskSpawn("Hilbert",90,
       VX_NO_STACK_FILL,1000,Ph3,
       0,0,0,0,0,0,0,0,0,0);
  /*Rest of the initial code */
```

```
do_Ph1 =
(P(mutex_a).P(mutex_b)
 .V(mutex_a).V(mutex_b))

do_Ph2 =
(P(mutex_b).P(mutex_c)
 .V(mutex_b).V(mutex_c))

do_Ph3 =
(P(mutex_c).P(mutex_a)
 .V(mutex_c).V(mutex_a))
....

init: do_Task1 | do_Task2
| do_Task3 | do_Task4
| do_Task5 | do_Task6
| do_Task7 | do_Task8
| do_Task9 | do_Task10
| do_Task11 | do_Task12
| do_Task13 | do_Ph3
| do_Ph2 | do_Ph1
```

Fig. 7. Inserting philosophers in the original source code (left), Philosophers in the `PV` model (right)

```
Geometric Model (forbidden area):
  [0,+∞[^2×[1,+∞[×[0,+∞[^13 ∪ [0,+∞[^3×[1,+∞[×[0,+∞[^12
∪ [0,+∞[^9×[1,+∞[×[0,+∞[^6 ∪ [0,+∞[^11×[1,+∞[^2×[0,+∞[^3
∪ [0,+∞[^13×[1,3[×[2,4[×[0,+∞[ ∪ [0,+∞[^13×[2,4[×[0,+∞[×[1,3[
∪ [0,+∞[^14×[1,3[×[2,4[

Local Deadlock Attractor: [0,+∞[^13×[1,2[^3
```

Fig. 8. `ALCOOL` output on the modified case study

5 Related Work

Tools implementing model checking techniques [2, 9] usually work on a representative *model* of the program to be analyzed, e.g. `SPIN` [21], `FAST` [5], `UPPAAL` [6]. While `SPIN` addresses general concurrent systems and their synchronization issues, `UPPAAL` is dedicated to real-time systems and is thus more focused on to timing issues, e.g. *delays*. The `FAST` tool is dedicated to the analysis of infinite systems. It mainly aims at computing the exact (infinite) set of configurations

reachable from a given set of initial configurations. In some cases, the verification models can be generated by auxiliary tools, e.g. the `MODEX` tool for `SPIN` [22]; the `TOPICS` tool for `FAST` [23]. As seen in sections 3 and 4, the `ALCOOL` tool can similarly be used in conjunction with the `MIEL` automatic model extractor.

Automata theory is a widely used framework for the theoretical foundations of model checking tools; for instance `SPIN`, `FAST` and `UPPAAL` respectively work on Büchi automata, counter automata and timed automata.

In contrast, the `ALCOOL` tool is based on the *topological* notion of directed spaces. Generally speaking, the parallel composition operator is modeled by the Cartesian product in a well-suited category: $[\![A|B]\!] = [\![A]\!] \times [\![B]\!]$

An automaton is a directed graph endowed with some extra structure. Directed graphs form a category in which any Cartesian product exists though they do not fit to concurrency theory. Our claim is that using directed spaces is natural since the Cartesian product in the category of topological spaces behave as concurrency theory expects.

Other existing approaches include predicate abstraction and refinement (`ARMC` [27], `BLAST` [7], `SLAM` [4]), symbolic model checking (`NuSMV` [8]), or combining model checking and theorem proving (`SLAB` [14]).

6 Conclusion

This paper reports on the development of a tool-based methodology to demonstrate the freedom from certain types of software faults, and its practical application to an industrial control software case study. Two main phases are involved: automatically extract a correct and representative model from `C` code, and then check for properties in the model.

Lessons Learned. The methodology presented in this paper aims at analyzing software that can be found in systems either safety related, or important to availability, in power plants. Experience in assessments of control systems has lead us to identify generic characteristics for such software (cf. Sec. 2). For instance, memory allocation, task and synchronization resources creation are usually performed only during a dedicated initialization phase. We have also relied on a taxonomy of synchronization faults that might be postulated in control systems software, established in previous works (cf. Sec. 1), and discussed how the tools can give rigorous evidence against a part of it. Finally, Sec. 4 shows that the tools scale up efficiently to analyze a real-world control system software unit. Also, voluntarily inserted faults have been identified.

An important learning is that generic characteristics of targeted classes of software can be taken in account in order to provide rigorous verification tools with high efficiency.

The soundness of this approach depends on the following requirements. The main requirement ($R1$) is that there is no dynamic creation of threads nor synchronization resources. The other requirements hold only for model extraction with `MIEL`: ($R2$) the programming language must be `C`, ($R3$) there must be no

aliases on the names of the functions of interest, and ($R4$) calls to functions of interest must be syntactically distinct. An important claim regarding applicability of the methodology is that these requirements are compatible with the generic characteristics of the targeted class of software. Under these assumptions, the methodology should widely apply within the considered class.

Additionally, let us examine the following cases. If one wants to analyze software where:

- only ($R1$) holds, then `ALCOOL` can be applied to a model extracted by other means than `MIEL` (e.g. expert knowledge or another tool);
- only ($R1$) to ($R3$) hold: the required user manipulation for having a sound model extraction with `MIEL` should remain fairly reasonable and practicable, i.e. unrolling loop so that each access to synchronization resources becomes syntactically noticeable and distinct for `MIEL`, as in Sec. 4.2. If user intervention is thought too demanding, one would consider the case below.
- only ($R1$) and ($R2$) hold: we are currently working on extending the methodology with a enhanced model extractor, to deal with this case; cf. Sec. 6, part "ongoing work".

Ongoing Work. We are currently experimenting the use of semantic analyzes using `Frama-C`'s value analysis [10] to provide a sound value analysis for concurrent programs in order to formally and accurately identify synchronization variables and threads, and use this information to refine model extraction.

The `ALCOOL` tool is based on a mathematical library which allows it to deal with geometric models drawn on a higher dimensional torus. However the representation of a finite directed graph on a hypertorus, which is known to be theoretically possible, has not been implemented yet. Roughly speaking, `ALCOOL` is meant to provide any concurrent program with a mathematical structure which generalizes the notion of control flow graph. Once this structure is determined, standard methods from static analysis can be applied. The tool-based approach should then provide a finer-grained analysis of message passing mechanisms.

References

1. Aiken, A., Foster, J.S., Kodumal, J., Terauchi, T.: Checking and inferring local non-aliasing. In: PLDI, pp. 128–140 (2003)
2. Baier, C., Katoen, J.P.: Principles of Model-Checking. MIT Press, Cambridge (2008)
3. Balabonski, T., Haucourt, E.: A geometric approach to the problem of unique decomposition of processes. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 132–146. Springer, Heidelberg (2010)
4. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In: POPL, pp. 1–3 (2002)
5. Bardin, S., Finkel, A., Leroux, J., Petrucci, L.: FAST: acceleration from theory to practice. STTT 10(5), 401–424 (2008)
6. Behrmann, G., David, A., Larsen, K.G.: A tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)

7. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker BLAST. STTT 9(5-6), 505–525 (2007)
8. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NUSMV: A new symbolic model checker. STTT 2(4), 410–425 (2000)
9. Clarke Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
10. Cuoq, P., Prevosto, V.: FRAMA-C's value analysis plug-in. CEA LIST Technical Report (2010), http://frama-c.com/download/frama-c-value-analysis.pdf
11. Cuoq, P., Signoles, J., Baudin, P., Bonichon, R., Canet, G., Correnson, L., Monate, B., Prevosto, V., Puccetti, A.: Experience report: OCaml for an industrial-strength static analysis framework. In: ICFP, pp. 281–286 (2009)
12. CWE Common Weakness Enumeration —, http://cwe.mitre.org/
13. Dijkstra, E.W.: Cooperating sequential processes. In: Programming Languages: NATO Advanced Study Institute, pp. 43–112. Academic Press, London (1968)
14. Dräger, K., Kupriyanov, A., Finkbeiner, B., Wehrheim, H.: SLAB: A certifying model checker for infinite-state concurrent systems. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 271–274. Springer, Heidelberg (2010)
15. Emanuelsson, P., Nilsson, U.: A Comparative Study of Industrial Static Analysis Tools. Linköpink University Technical Report (2008)
16. Fajstrup, L., Goubault, E., Raußen, M.: Algebraic topology and concurrency. Theoretical Computer Science 357, 241–278 (2006)
17. FRAMA-C Software Analyzers —, http://frama-c.com/
18. Goubault, E.: Geometry and concurrency: a user's guide. Mathematical Structures in Computer Science 10(4), 411–425 (2000)
19. Goubault, E., Haucourt, E.: A practical application of geometric semantics to static analysis of concurrent programs. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 503–517. Springer, Heidelberg (2005)
20. Grandis, M.: Directed Algebraic Topology. New Mathematical Monographs. Cambridge University Press, Cambridge (2009)
21. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional, Reading (2003)
22. Holzmann, G.J., Ruys, T.C.: Effective bug hunting with SPIN and MODEX. In: Godefroid, P. (ed.) SPIN 2005. LNCS, vol. 3639, p. 24. Springer, Heidelberg (2005)
23. Labbé, S., Sangnier, A.: Formal verification of industrial software with dynamic memory management. In: IEEE PRDC. pp. 77–84 (2010)
24. Labbé, S., Thuy, N.: Formal verification of freedom from intrinsic software faults in digital control systems. In: ANS NPIC&HMIT, pp. 2191–2201 (2010)
25. Larochelle, D., Evans, D.: Statically detecting likely buffer overflow vulnerabilities. In: USENIX Security Symposium, pp. 177–190 (2001)
26. Nachbin, L.: Topology and Order. Mathematical Studies, vol. 4. Van Nostrand, Princeton (1965)
27. Podelski, A., Rybalchenko, A.: ARMC: The logical choice for software model checking with abstraction refinement. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 245–259. Springer, Heidelberg (2006)
28. Thuy, N., Ourghanlian, A.: Dependability assessment of safety-critical system software by static analysis methods. In: DSN, pp. 75–79 (2003)