Model Checking a C++ Software Framework: A Case Study

John Lång
University of Helsinki
Finland
john.lang@mykolab.com

I.S.W.B. Prasetya
Utrecht University
the Netherlands
s.w.b.prasetya@uu.nl

ABSTRACT

This paper presents a case study on applying two model checkers, Spin and Divine, to verify key properties of a C++ software framework, known as ADAPRO, originally developed at CERN. Spin was used for verifying properties on the design level. Divine was used for verifying simple test applications that interacted with the implementation. Both model checkers were found to have their own respective sets of pros and cons, but the overall experience was positive. Because both model checkers were used in a complementary manner, they provided valuable new insights into the framework, which would arguably have been hard to gain by traditional testing and analysis tools only. Translating the C++ source code into the modeling language of the Spin model checker helped to find flaws in the original design. With Divine, defects were found in parts of the code base that had already been subject to hundreds of hours of unit tests, integration tests, and acceptance tests. Most importantly, model checking was found to be easy to integrate into the workflow of the software project and bring added value, not only as verification, but also validation methodology. Therefore, using model checking for developing library-level code seems realistic and worth the effort.

CCS CONCEPTS

• **Software and its engineering** → **Formal software verification**.

KEYWORDS

model checking concurrent C++, verification concurrent C++, model checking C++ case study

1 INTRODUCTION

ADAPRO stands for *ALICE Data Point Processing Framework*. It is an open source C++ 14 software framework[1], consisting of about 6000 lines of source code[2]. It is meant for building configurable, remote-controllable, multi-threaded daemon applications. ADAPRO was originally conceived as a collection of common routines used for implementing the *ALICE Data Point Service* (ADAPOS)[22] software architecture (as part of the ALICE RUN3 upgrade[9]). ALICE stands for *A Large Ion Collider Experiment.* ALICE is one of the four major experiments at the *Large Hadron Collider* (LHC) of the *European Organisation for Nuclear Research* (CERN).

The highly concurrent nature of ADAPRO makes its verification challenging by conventional means. Although the framework and its applications[22] have been subject to hundreds of hours of unit tests, integration tests, and acceptance tests, ADAPRO is too complex for its all behaviours to be anticipated by tests. Therefore, it is also hard to say how adequate these tests actually were in covering the software's concurrent behavior. Even though ADAPRO has its roots in a specific use case, it has evolved into a reusable tool. Some ADAPRO applications may be expected to be able to run autonomously for months without human intervention.

All of these facts reinforce the importance of a formal verification project, because even rarely occurring defects may cause costly damage, as noted by Gerard J. Holzmann, the author of the Spin model checker [15]. A study by John Fitzgerald et al. confirms the impact of formal verification on software quality [10].

To address this challenge we set up a project to explore the feasibility of applying *model checking*[2, 19, 25] to thoroughly verify ADAPRO's critical properties. We chose model checking, because it naturally fits with the *Finite State Machine* (FSM) paradigm used by the framework, which will be discussed later. Furthermore, we decided to use model checking on two levels of abstraction. This decision proved to be a good idea, since using two different model checkers in a complementary manner helped to find different kinds of issues faster.

On the higher level of abstraction, we wanted to verify that the very design of ADAPRO itself is correct, so we constructed a new model for it. After experimenting with the *NuSMV*[7], *TLA+*[21], and Spin[14] model checkers, we decided to choose Spin for this purpose. For verifying ADAPRO's actual implementation, we performed software model checking. The model checker Divine[3] was chosen for this purpose.

Contribution. Formal verification turned out to reveal important issues not previously found by testing, even though it wasn't possible to obtain exhaustive results. Our findings necessitated changes

---

[1]available online at https://gitlab.com/jllang/adapro
[2]as measured in 25 March 2019, using David A. Wheeler's Sloccount utility, available online at https://dwheeler.com/sloccount/

in the design and implementation of ADAPRO. The changes will be part of the foundation of the next major version of ADAPRO (v5.0.0[3]). In this paper we will share our experience, discoveries, and lessons gained from our verification project.

Paper structure. We first give a brief overview on related work in Section 2. Section 3 offers an overview of the ADAPRO framework. Section 4 gives the necessary definitions, assumptions and properties for design-level verification. In Section 5, we discuss our findings on this part of the project. Section 6 describes the effort for verifying the C++ implementation and its results. In Section 7, we discuss our experiences and lessons learned from using the two model checkers. We end the article with conclusion, future prospects, and acknowledgements.

2 RELATED WORK

There are many case studies on using model checking to verify the correctness of production software; too many to list here comprehensively. Some examples include [1, 5, 6, 8, 12, 17, 18, 26]. Fitzgerald et al. have written a survey on many industrial use cases [10].

ADAPRO is similar to the *SMI++*[11] system in its reliance on an FSM model. However, ADAPRO is more restricted, since it doesn't have a *Domain Specific Language* (DSL) and its FSM model is rigid in the sense that the user can't define new states or commands. Instead of synthesizing distributed control systems, ADAPRO is focused on assembling threads into a remote controllable concurrent application. ADAPRO applications can interact with SMI++ based systems using the *Distributed Information Management* (DIM)[13] protocol as its communications layer, which has been demonstrated by the ADAPOS Manager application.

All LHC experiments use the SMI++ system for their control systems. Formal verification has been performed on the *Compact Muon Solenoid* (CMS) control system [18], which demonstrates the feasibility of building and analysing control systems with tens of thousands of nodes, based on hierarchies of FSMs. ADAPRO also follows a similar approach, though it features a simple tree with just root and a number of leafs.

Compared to most of the papers mentioned in this section, our approach was more lightweight in that we didn't use automated model extraction or translation tools. Similarly to [26] and [6], we wanted to explore the correct level of abstraction for finding the most relevant aspects of the algorithms used in ADAPRO, by building the design-level model by hand. Due to the complexity of ADAPRO, full formal software model checking was not computationally feasible. Instead, we used DIVINE as a high coverage bug hunting tool. Our lightweight approach seemed to suit the needs of this project well, and we believe it to be realistically reproducible in other similar projects.

3 ADAPRO

The basic actor in ADAPRO is the abstract *Thread* class, which follows an FSM approach. The domain logic of an application is meant to be implemented as virtual methods and/or callbacks, called *user-defined code*, provided to the framework through specialized
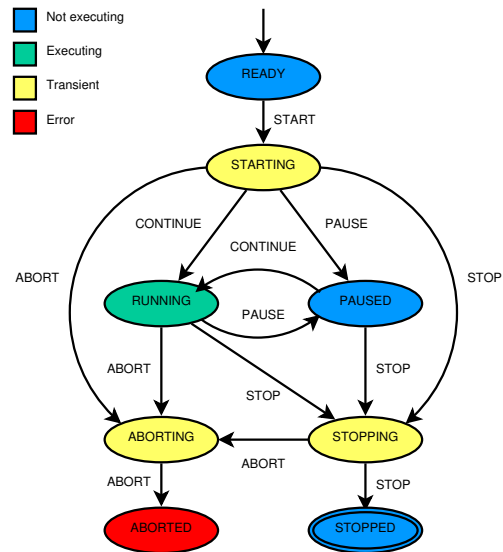

Figure 1: Thread state transition diagram

Thread instances. The role of the framework is to manage Threads. Figure 1 shows the state transition diagram of the Thread FSM.

A Thread moves from one state to another upon *commands* as depicted in Figure 1. Should the Thread ever encounter an undefined state/command combination, it is specified to take no action (other than printing a warning message) in such situation. To prevent this from happening, the FSM is not directly exposed to the user. Instead, an object-oriented approach with safe accessor methods is used. The methods for changing the command of a Thread are called *trigger methods*, and they have *synchronous* (i.e. blocking) and *asynchronous* (non-blocking) variants. More information on the *Application Programmable Interface* (API) is available on GitLab[4].

During a state transition, the Thread[5] invokes a *transition callback*, a user-defined function that takes the target state as its argument. The transition callback is not allowed to throw exceptions or interfere with the framework in any other way. If the Thread enters the state STARTING or STOPPING, it invokes a corresponding virtual method, implemented by the user. The Thread keeps invoking another virtual method in a loop for as long as it stays in the state RUNNING. The three virtual methods associated with these states (prepare, finish, and execute respectively) are allowed to fail by throwing exceptions. If the Thread base class catches an exception thrown by the inheriting class, it moves to the state ABORTING, prints an error message, moves to the state ABORTED, and halts.

The states whose names end with "ING" in Figure 1 are the ones, during which the backend thread is performing computation. A Thread that is in one of the states READY, STOPPED, or ABORTED must neither posess any dynamically allocated objects or resources, nor hold any locks, so that it can be safely deleted. A Thread in

---

[4]see https://gitlab.com/jllang/adapro/-/jobs/artifacts/5.0.0/download?job=manual for the manual and https://jllang.gitlab.io/adapro/ for the API documentation
[5]throughout this article, we use capital initial letter 'T' to distinguish the ADAPRO class Thread from the general concept of a thread
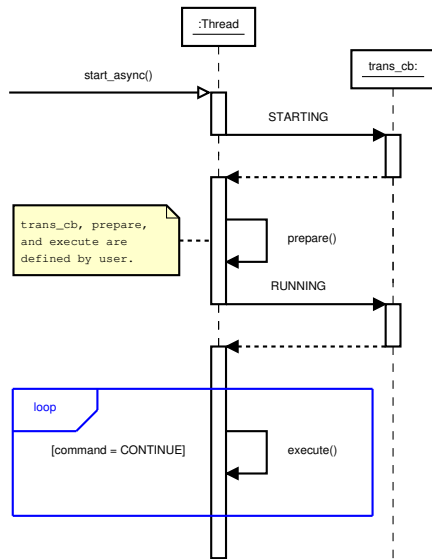
Figure 2: Asynchronous Thread startup

Figure 3: Session initialization

state PAUSED is considered to be temporarily suspended and capable of moving back to the state RUNNING or proceeding to the STOPPING state on short notice.

Thread uses the C++ standard library thread class as its backend. Some advanced functionality also involves the PThreads and Linux APIs through native handles. POSIX or Linux specific parts of the framework are not critical and they're guarded with macros to ensure portability. These parts were left outside of the scope of this verification project.

The interpretation of the states and transitions is up to the implementing class, since Thread doesn't define (i.e. it only *declares*) the virtual methods or the transition callback. The intended use case of the Thread class is running a repetitive task in background, in the way how services or daemons work in process level. For a batch job, a standard library thread is probably more appropriate choice. Figure 2 is a UML sequence diagram showing the asynchronous startup sequence of a Thread as an example on the interactions between the owner of a Thread, the backend of the Thread, and the transition callback.

3.1 Session and Supervisor

An ADAPRO application consists of a special Thread known as the *Supervisor* and one or more other Threads, known as workers. The user mustn't manage Supervisor directly, but instead utilize the static methods of a class called *Session*, which follows the *singleton* design pattern and runs on the main thread. Session is responsible for framework startup and shutdown sequences. For technical reasons, the lifetime of a Session consists of separate initialization and runtime phases.

During the initialization phase, Session registers signal handlers first. Then it constructs the configuration using default values, file, and/or command-line arguments, initializes a logger. After that, Session possibly performs certain interactions with the operating system. At the end of the initialization phase, Session constructs a
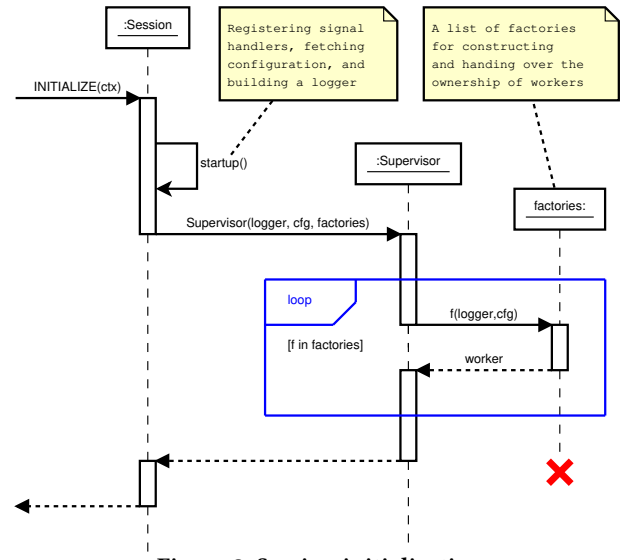
Supervisor, handing over references to the logger, configuration, and user-defined worker factories. Supervisor then takes care of constructing the workers by applying the factories to the logger and configuration. Figure 3 gives a simplified overview on Session initialization.

The state of Supervisor represents the overall state of the application in the sense that Supervisor is the first Thread to start and the last Thread to stop. When a signal or command arrives from an external process, Supervisor propagates the appropriate FSM command to the workers. Figure 4 presents propagation of the START command as an example. The sequences for propagating PAUSE, RESUME, and STOP commands are similar.

The runtime phase of Session starts with Session sending the START command to Supervisor. After that, Session remains passive until the Supervisor halts or a signal handler or the global exception handler is activated. When the runtime phase ends, Session returns a one-byte status code, which is a bitmask of eight different flags representing certain common error categories.

Supervisor starts workers asynchronously, after which it blocks until all workers have ended their startup sequences. When the startup sequence ends, Supervisor and all workers have moved into one of the six states below STARTING shown in Figure 1. In addition to propagating an external STOP command, Supervisor also propagates it if one or more workers have aborted or if all of the workers have stopped.

4 DESIGN-LEVEL VERIFICATION

We first wanted to verify that the design of ADAPRO is correct. To this end, a set of key correctness properties called *the Theory of ADAPRO* was identified and formalized in *Linear Temporal Logic* (LTL)[23], presented in Section 4.2 below. Section 4.4 discusses the construction a model capturing ADAPRO's logic. The model was written in PROMELA[14]. Using the model checker Spin [4, 14], we verified that this model satisfies the Theory of ADAPRO. Section 5 discusses the findings.
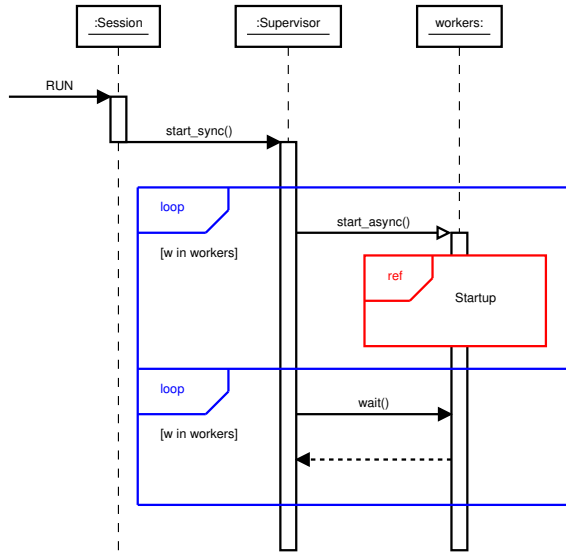
Figure 4: Command propagation

4.1 Preconditions

The ADAPRO framework relies on a programming contract between the framework developers and the application developers. There are necessary *preconditions*/assumptions on the environment and user-defined code, that ADAPRO has to take as granted. The full list of preconditions is given below for completeness:

(1) There will be no sabotage or force majeur incidents of any kind whatsoever.

(2) The C++ compiler, standard library and runtime system, the operating system and the hardware platform function correctly with respect to their specifications.

(3) The operating system scheduler satisfies weak fairness, i.e. the property that every native thread that is eventually always executable, will be run on a processor unboundedly many times.

(4) Allocating memory for new objects always succeeds, i.e. the machine will not run out of (virtual) memory.

(5) User-defined code always terminates.

(6) Unless explicitly permitted to do so, by ADAPRO manual or the API documentation, user-defined code will not modify or delete any object owned by the framework.

(7) User-defined code never
  - calls std::abort;
  - raises a signal that can't be handled;
  - causes deadlocks or livelocks;
  - uses Thread trigger methods inappropriately; or
  - triggers a known issue in the framework.

(8) User-defined code conforms to the well-known *Resource Aqcuisition is Initialization* (RAII) principle, so that all memory and resources acquired by user-defined code will be automatically released when ownership expires (e.g. a destructor of a user-defined object, holding a resource, is invoked).

(9) In DIM server mode or daemon mode, the user-defined code does not directly interact with DIM or Systemd libraries respectively.

The appropriateness of the use of trigger methods is best explained by enumerating the acceptable scenarios. We do this in Sections 4.3 and 6 for the design-level and the implementation-level models respectively.

4.2 The Theory of ADAPRO

Let us first define the following notation:

(i) Denote Supervisor as s;

(ii) The set of all Threads, including Supervisor, as T;

(iii) The set of Workers, i.e. $T \setminus \{s\}$ as V;

(iv) null(x) as the predicate "Thread x doesn't exist";

(v) For each state q the predicate $q(x)$ expressing that "Thread x is in state q" (e.g. ready(x) denotes "Thread x is in state READY");

(vi) halting(x) as stopping(x) $\vee$ aborting(x);

(vii) halted(x) as stopped(x) $\vee$ aborted(x);

(viii) executable(x) as "Thread x is in the state RUNNING with its command set to CONTINUE"; and

(ix) executing(x) as "Thread x is carrying out its action associated with the state RUNNING".

The following are the key correctness properties of ADAPRO, that represent the *postconditions* promised as a part of the programming contract:

(1) $\forall_{t \in T}[\text{null}(t)\mathbf{U}(\text{ready}(t)\mathbf{W}\,\text{starting}(t))]$;

(2) $\forall_{t \in T}\square[\text{starting}(t) \rightarrow (\text{starting}(t)\mathbf{U}(\text{running}(t) \vee \text{paused}(t) \vee \text{halting}(t)))]$;

(3) $\forall_{t \in T}\square[\text{running}(t) \rightarrow (\text{running}(t)\mathbf{W}(\text{paused}(t) \vee \text{halting}(t)))]$;

(4) $\forall_{t \in T}\square[\text{paused}(t) \rightarrow (\text{paused}(t)\mathbf{W}(\text{running}(t) \vee \text{stopping}(t)))]$;

(5) $\forall_{t \in T}\square[\text{stopping}(t) \rightarrow (\text{stopping}(t)\mathbf{U}(\text{stopped}(t) \vee \text{aborting}(t)))]$;

(6) $\forall_{t \in T}\square[\text{stopped}(t) \rightarrow \square\,\text{stopped}(t)]$;

(7) $\forall_{t \in T}\square[\text{aborting}(t) \rightarrow (\text{aborting}(t)\mathbf{U}\,\text{aborted}(t))]$;

(8) $\square(\forall_{t \in T}[\text{aborted}(t) \rightarrow \square\,\text{aborted}(t)])$;

(9) $\square(\text{ready}(s) \rightarrow \forall_{v \in V}[\text{null}(v)])$;

(10) $\square(\text{halted}(s) \rightarrow \forall_{v \in V}[\text{null}(v)])$;

(11) $\square(\exists_{v \in V}[\text{ready}(v)] \rightarrow \text{starting}(s))$;

(12) $\square(\forall_{v \in V}[\text{stopped}(v)] \rightarrow (\text{paused}(s) \vee \text{stopped}(s)))$;

(13) $\square(\exists_{v \in V}[\text{aborted}(v)] \rightarrow (\text{paused}(s) \vee \text{stopped}(s)))$;

(14) $\square(\text{halting}(s) \rightarrow \lozenge\forall_{v \in V}[\text{halted}(v)])$;

(15) $\neg\lozenge\,\text{aborted}(s)$;

(16) $\forall_{t \in T}[\lozenge\square\,\text{executable}(t) \rightarrow \square\lozenge\,\text{executing}(t)]$; and

(17) $\forall_{t \in T}[\square\lozenge(\text{executable}(t) \rightarrow \text{executing}(t))]\mathbf{W}\,\text{halting}(s)$.

Formulae 1 – 8 capture the next-state relation induced by the FSM (see Figure 1). Notice that some of them (1, 2, 5, 7), use strong until (**U**), to express that the corresponding transitions are inevitable. Others use weak until to express that the transitions are not required to be taken.

Formulae 9 – 15 express additional safety properties that every ADAPRO session is expected to satisfy, e.g. 9 and 10 state that no worker should exist while the Supervisor is still in the READY state, or after it halts, whereas 15 says that the Supervisor should never

abort. Formula 16 expresses weak fairness. Finally, formula 17 is a liveness property that asserts that under the right conditions, all Threads get to execute their main task.

4.3 Modeling Strategy

Since a model checker cannot verify a generic system with unbounded number of Thread instances, we model a system representing an ADAPRO application with one Supervisor and two workers. The rationale for using two workers is that it allows the distinction between universally and existentially quantified statements about workers. Having two workers also allows the model checker to expose potentially inconsistent worker state combinations. We suspect that having three or more workers would only make the model larger without introducing essentially new species of errors. As noted in the beginning of Section 5, state space size was a practical reason forcing us to limit our scope to two workers.

The workers are treated as black boxes that only have their FSM interface and do not communicate with each other or share resources. A worker may initiate the following state transitions, simulating unhandled exceptions and the use of the worker's own trigger methods[6]:

- During `prepare`, the worker chooses non-deterministically between pausing, stopping, aborting, and doing nothing.
- During `execute`, the worker chooses non-deterministically between stopping, aborting, and doing nothing.
- During `finish`, the worker chooses non-deterministically between aborting and doing nothing.

Supervisor was modeled quite faithfully with respect to its C++ implementation. In the model, Supervisor doesn't spontaneously send commands to workers. It delegates commands received from the init process, which represents the environment, and reacts if both workers stop or at least one of them aborts.

4.4 Building the PROMELA Model

Since ADAPRO initially did not have a formal design, we reconstructed manually its design/logic from its C++ implementation in the modeling language PROMELA[14]. We focused on the logic that implements the FSM in Figure 1.

To model the behavior of a C++ program inevitably involves figuring out how to map C++ programming patterns to PROMELA. For example, the state of an ADAPRO Thread is stored in a private field[7] of type `std::atomic<State>`, where the template argument is an enumerated type consisting of the eight different states indicated in Figure 1. This field is only accessed through the accessors `get_state` and `set_state`.

In a concurrent C++ application, naïve reading and writing of shared memory does not work as one might expect. Threads executing on CPU cores may use the cores' local cache memory. It's not possible to control caching directly using C++ language constructs. Modifications made by one thread might get stuck in the local cache of a CPU core and never become visible to other threads.

Instruction reordering might also wreak havoc by causing modifications to show in wrong order. Instead of providing its own solutions, ADAPRO relies on C++ standard library synchronization primitives. In particular, reading and writing a shared state involves a memory barrier to ensure that the changes will be visible to all parties as intended:

```cpp
enum State { READY,    STARTING, RUNNING,  PAUSED,
             STOPPING, STOPPED,  ABORTING, ABORTED };
enum Command { START, STOP, PAUSE, CONTINUE, ABORT };
class Thread {
    std::atomic<State> state;
    std::atomic<Command> command;
    std::mutex m;
    void set_state(const State s) noexcept {
        state.store(s, std::memory_order_release); }
public:
    State get_state() const noexcept {
        return state.load(std::memory_order_consume); } };
```

In contrast, PROMELA offers high level *atomicity*: a single assignment is always atomic, and it is possible to declare a block of statements to be atomic. PROMELA also guarantees *sequential consistency*, defined by Leslie Lamport[20] as follows:

> "[T]he result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."

This implies that all (atomic) changes to variables are guaranteed to be visible to all processes. With these properties, inter-thread communications in ADAPRO become easy to model in PROMELA, hence allowing us to focus on modeling its algorithms. The ADAPRO functionality mentioned can be modeled succinctly as as shown below. We assume there are N threads; their states are stored in a global array called `states`:

```
mtype = {READY,STARTING,RUNNING,PAUSED,STOPPING,STOPPED,
         ABORTING,ABORTED};
mtype states[N]; /*the states of the N threads*/
#define get_state(i) states[i]
#define set_state(i,s) states[i] = s
```

In C++, `std::mutex` and `std::condition_variable` are often used to implement inter-thread communication. In PROMELA we can abstract this away. For example, if a process P wants to wait until a certain predicate C becomes true, we can simply write C as a guarding expression, hence resulting in a cleaner model. The code example below presents a simplified Thread model that utilizes the comparisons of the command and state values as guards (e.g. `get_command(k) == START`):

```
proctype Thread(byte k) { /* a Thread with id k */
  /* Startup: */
  get_command(k) == START;
  set_state(k,STARTING);
  set_command(k,CONTINUE);
  prepare(k); /* the action of the state STARTING */
  /* Execution of user code: */
  do
  ::get_command(k) == CONTINUE ->
      if
      :: get_state(k)==STARTING-> set_state(k,RUNNING)
      :: get_state(k)==PAUSED  -> set_state(k,RUNNING)
      :: else -> skip
      fi
      execute(k)
  ::get_command(k) == PAUSE  ->
```

---

[6]in C++, a Thread must always call its own trigger methods asynchronously; otherwise it will end up in a deadlock waiting for itself to finnish executing its own command. This issue was known and documented well before beginning the verification project.
[7]or *data member* in C++ terminology

```
       if
       ::get_state(k)==PAUSED -> get_command(k)!=PAUSED
       ::else ->
           if
           ::get_state(k)==STARTING||get_state(k)==RUNNING
                   -> set_state(k,PAUSED)
           ::else -> skip
           fi
       fi
   ::get_command(k)==STOP||get_command(k)==ABORT -> break
   od
   /* Shutdown: */
   if
   ::get_command(k)==STOP -> /* ... */
   ::get_command(k)==ABORT-> /* ... */
   fi }
```

Notice that in the case where the command is either STOP or ABORT, get_command(i) appears twice. Because it's a macro expanding to commands[i], it doesn't require a new variable. Both comparisons are part of the same atomic PROMELA transaction. Using a temporary variable in a situation like this might increase the state space size of the model and even break temporal properties as the level of atomicity would change. During its execution, a Thread will keep looping indefinitely and performing the appropriate action as long as its command is not STOP or ABORT. If the command changes to one of these, the Thread exits the loop and begins its shutdown sequence.

The full model is nearly 800 lines long (including comments) and can be found at[8]. The model maintains a quite fine grained atomicity in order to maximise interleaving possibilities that will be checked. As the tradeoff, model checking a fine grained model can be expected to consume more resources.

5 VERIFICATION RESULTS WITH SPIN

Verifying the theory of ADAPRO successfully with a Supervisor and two workers SPIN takes less than three gigabytes of memory (without need for compressing states), and a couple of minutes with a high-end laptop. For three workers, the verification takes SPIN ran out of the 17 GB of memory available to it, after nearly two hours. The verifier did manage to verify properties (1) – (15) for three workers though, thanks to state compression option -DCOLLAPSE used for compilation.

Writing the model itself exposed a number of design flaws and bugs in the C++ code. Additional defects were found during model checking because the PROMELA model was accurate enough to replicate them. This section discusses two of the issues that had managed to remain undetected during testing.

5.1 Revoked ABORT Defect

As mentioned in Section 3, a Thread has to move to the state ABORTING if it encounters an unhandled exception. It is important that the ABORT command cannot be revoked. This requirement is captured by specification (7) in the ADAPRO Theory (Section 4.2). It turned out that this property was violated.

After startup, a Thread runs one loop as shown in the previous listing. In C++, the algorithm looks like the following:

```
void Thread::run() noexcept {
    try {
        /* Startup */
```

[8]https://gitlab.com/jllang/adapro/tree/5.0.0-RC3/models/promela

```
       bool shutdown{false};
       while (!shutdown) {
           switch (get_command()) {
               case CONTINUE: /* ... */ break;
               case PAUSE:    /* ... */ break;
               default: shutdown = true;
           }
       }
       /* Shutdown */
   }
   catch (const std::ios_base::failure& e) {HANDLE(e);}
   catch (const std::system_error& e) {HANDLE(e);}
   /* ... */
   catch (const std::exception& e) {HANDLE(e);} }
```

The macro HANDLE takes care of printing an error message and initiating the transition to state ABORTED via ABORT. Multiple catch blocks with the same code are needed to avoid object slicing, which would cause the method std::exception::what to return just "std::exception" (on Linux systems at least), which is not a very helpful error message.

It is possible that user-defined code throws an exception, which is caught by one of the catch blocks. The method handle_exception then sets the command to ABORT. At this point, it may happen that a PAUSE command arrives from an external source, and that Supervisor propagates the command PAUSE to all workers. Now the worker that was about to abort is told to go to the state PAUSED instead. This violates the FSM constraints (see Fig. 1), as STOP and ABORT should never get overridden by commands of lower priority.

To make sure that set_state can never violate the FSM mechanism, and that the ABORT command is irrevocable, it was necessary to redefine set_command (which was previously defined in a fashion similar to set_state) in PROMELA as follows:

```
#define set_command(i, c)                              \
    atomic {                                           \
        if                                             \
        :: c == START && commands[i] == CONTINUE ->    \
                commands[i] = START                    \
        /* ... */                                      \
        :: c == STOP && (commands[i] == CONTINUE ||    \
                        commands[i] == PAUSE)  ->      \
                commands[i] = STOP                     \
        :: c == ABORT ->  commands[i] = ABORT          \
        fi }
```

The corresponding update in the C++ code required changing the locking scheme. It took a few attempts to arrive at a correct solution. The DIVINE model checker was found useful during this development process, as it discovered a flaw in one of the attempted solutions. The final correct version is given below:

```
void Thread::set_command(const Command c) noexcept {
    bool success{false};
    std::lock_guard<std::mutex> lock{m};
    switch (c) {
        case START:
            success = get_state() == READY
                    && get_command() == CONTINUE;
            break;
        case PAUSE:
            success = state_in(STARTING | RUNNING)
                    && get_command() == CONTINUE;
            break;
        /* ... */
        case ABORT:
            success=state_in(STARTING|RUNNING|STOPPING);
    }
    if (success) {
        command.store(c, std::memory_order_release);
```

```
    } }
```

The method run had to be changed as well, to hold a lock on the mutex when performing state transitions. The lock has to be released before, and reacquired after, executing user-defined virtual methods, though. The reason is that the user-defined virtual methods are allowed to use certain trigger methods of the executing Thread (see Section 4.3).

5.2 A Synchronisation Defect

Spin was also to able to re-discover a known issue. There was an overlooked design flaw in the C++ code for a synchronisation method named wait_for_state, part of the API. It had escaped CPPUnit tests of ADAPRO that existed at the time, which had around 90% overall line coverage.

The method wait_for_state is supposed to block until the Thread has moved to a state greater than or equal to a given target state. In terms of executions, some states come after others, so they are greater than their predecessors. As ADAPRO leaves scheduling to the operating system, it cannot guarantee that get_state will be executed exactly at the right moment, and not after the Thread has already moved to a later state, hence the need for also accepting states greater than the target state.

Two synchronization primitives have to be added to the Thread class in order to implement blocking behaviour. The C++ code snippet below shows the erroneous wait_for_state implementation. It relies on the integer representation underlying the enumerated type State for the comparison get_state() < s:

```cpp
class Thread {
    std::atomic<State> state;
    std::mutex m;
    std::condition_variable cv;
    void set_state(const State s) noexcept {
        state.store(s, std::memory_order_release);
        cv.notify_all(); // This new line was needed
    }
public:
    State get_state() const noexcept; // Same as before
    void wait_for_state(const State s) noexcept {
        if (get_state() < s) {
            std::unique_lock<std::mutex> lock{m};
            cv.wait(lock,
                    [this,s] () {return get_state() < s;}
            );
        } }
    };
```

In the code fragment above, the lambda expression given to the standard library method wait uses the method get_state to compare the state of the Thread instance to s. It's a guard against spurious wake-ups. wait will always block until the condition variable is notified and the lambda expression returns true. The behavior can be succinctly modeled in Promela, as shown below[9]:

```
#define LT(x, y) x > y /* Sic */
inline wait_for_state(i, s) { !(LT(get_state(i), s)) }
```

Informally, the state ordering meant in wait_for_state is the preorder of states implied by Figure 1 when reading the arrows as "less than". Formally, the ordering is imposed by the enum definition

---

[9] The peculiar definition for LT, the less-than relation, can be explained with the fact that Spin treats the symbolic names in mtype declarations in big-endian order, i.e. increasing from right to left. Note that there are no spurious wake-ups in Promela.

for State. In most situations these definitions are similar enough to not cause appreciable difference in behaviour. There is a special case, however: when a Thread in state PAUSED is expected to proceed to the state RUNNING. Since PAUSED is formally greater than RUNNING, an invocation wait_for_state(RUNNING) on a paused Thread will immediately unblock, which is *incorrect*! This error was quickly found during the model checking with Spin.

C++ allows the programmer to specify integral constants for enumerators. This enables the use of bitmasks for conveniently expressing the exact set of target states where the waiting method needs to unblock. This detail was not modeled in Promela. Doing so would have required changing mtype for byte and losing symbolic names for states and commands in debug messages. Below is the reviewed C++ definition that enables the use of bit masks:

```cpp
enum State {
    READY    = 1,
    STARTING = 2,
    /* ... */
    ABORTED  = 128 };
#define state_in(mask) (get_state() & mask) > 0
```

The waiting function wait_for_state_mask utilizes bitmasks. It is the correct way to wait for a Thread to enter one of the states in the given bitmask. Below is the implementation:

```cpp
void Thread::wait_for_state_mask(const uint8_t mask)
noexcept {
    if (!state_in(mask)) {
        std::unique_lock<std::mutex> lock{m};
        cv.wait(lock,
                [this,s] () {return state_in(mask);}
        ); }
}
```

This method can wait for a Thread in state PAUSED to continue:

```
wait_for_state_mask( RUNNING  | STOPPING | STOPPED |
                     ABORTING | ABORTED );
```

As mentioned, the Promela definition for states was not changed, so instead of taking a bitmask, a hard-coded inline waiting block was added for each of the the five bitmasks that were needed. For instance, the C++ method invocation above was hard-coded into Promela as follows:

```
inline wait_for_RESUME_mask(i) {
    get_state(i) == RUNNING || get_state(i) == STOPPING ||
    get_state(i) == STOPPED || get_state(i) == ABORTING ||
    get_state(i) == ABORTED; }
```

6 DIVINE MODELS AND RESULTS

The interplay of state and command setters, state transitions, and waiting methods is non-trivial. Not all of the subtleties involved were exposed by the Promela model, partly because of the semantical difference between the Promela and C++ languages, but also because some of the details were left out from the model. This provided motivation for implementation-level model checking.

We created the class DummyWorker for modeling workers in Divine. It inherits the appropriate ADAPRO class and behaves like the Promela worker model. The method prepare lets Divine choose an integer from [0..3] non-deterministically, expressed as __vm_choose(4) for selecting between the asynchronous trigger methods pause_async and stop_async, throwing an exception, and doing nothing. Similarly, the methods execute and finish

use the non-deterministic choice feature for implementing the behaviour described in Section 4.3. The following C++ code example illustrates the implementation:

```cpp
class DummyWorker final : public Worker {
protected :
    virtual void prepare() {
        switch (__vm_choose(4)) {
            case 0: pause_async(); break;
            case 1: stop_async(); break;
            case 2: throw std::runtime_error{"Error"};
            default: break;
        }
    }
    virtual void execute() { /* Stop, throw, or skip */ }
    virtual void finish() { /* Throw or skip */ }
}
```

We created three models for DIVINE using this class; the small model with one DummyWorker, the medium model with Supervisor and one DummyWorker, and the large model that runs a full ADAPRO Session, with its built-in Supervisor and two DummyWorker instances, using a hard-coded configuration. All models also have a main thread, that creates the appropriate ADAPRO objects, waits until they have carried out their tasks, and exits the program.

6.1 Premature Destructor Call Defect

As mentioned above, the smallest DIVINE model featured just the main thread and one worker Thread. In the model, the main thread starts the worker and calls wait_for_state_mask(STOPPED | ABORTED). DIVINE found an execution where the main thread was woken up when the worker was performing an earlier state transition. Before the main thread could proceed to check whether or not the worker had reached the desired state, a context switch was performed, letting the worker to proceed. Next, the worker would set its state to STOPPED, but before it was fully stopped, the main thread was scheduled again for execution. At that point, the main thread observed that the worker had entered into the STOPPED state and proceeded to exit the program.

At this point, a known issue about destroying a Thread with its backend implementation still running was triggered. The worker's destructor was called as a part of the automatic cleanup following the RAII principle. When the backend thread of the worker was scheduled again for execution, it tried to refer to the ADAPRO Thread object that didn't exist anymore which then caused the program to exit abnormally (and seemed to crash DiOS as well, through its Pthreads implementation). The premature destructor call issue has been the cause of the most catastrophical and hard to debug problems in the entire ADAPRO framework.

With the current design of ADAPRO Threads, this issue is not easy to fix. However, there exists a workaround which was found to be correct in all situations by DIVINE. This workaround is to simply make the virtual destructor of the final inheriting class to call Thread::join (which then calls std::thread::join to join the background thread) before releasing any resources.

6.2 Non-Terminated String Defect

A classic low-level programming mistake was found with DIVINE when verifying the unit tests for Thread. The abort method of a Thread was designed to permit the user to trigger the state transition through ABORTING to ABORTED with an error message given as an std::string instance. The method dynamically allocated a C-style string, i.e. an array of ASCII characters, and called std::strcpy using the pointer and length provided by the std::string instance. The length reported was one less than needed for the character array, because a C-style string must end with an additional non-printable null character. The resulting C-style string would contain all the same characters as the std::string object, but wouldn't be null-terminated.

Depending on the contents of the heap memory, reading a non-terminated string might succeed without problems, produce some extra garbage data past the intended ending of the string, or even cause a segmentation violation. There was also a lesser problem, namely that in some situations the C-style string was never deleted, causing a memory leak. Even Valgrind was not able to detect this leak, because it never actualised during debugging.

Both the off-by-one error with the string length and the memory leak were quickly detected by DIVINE, thanks to its sophisticated mechanisms that keep track on the objects allocated from heap and the pointers referring to them. Unit tests couldn't reliably detect this problem, as the next byte in heap usually happened to be zero. Even though this problem would have been easy to fix, the authors decided to abandon the abort method altogether as an unnecessary complication to the overall design.

This defect had been present in the framework for a very long time and wasn't even detected during the hundreds of hours of simulations performed with ADAPOS applications[22], built on top of ADAPRO, under maximum load, because the applications never encountered unhandled exceptions during the simulations.

6.3 Premature Command Defect

A model checker can only find errors reachable from the model under verification. A known issue with starting a Thread asynchronously, and then sending it a command before it has carried out its startup sequence, resulting in the command being ignored, was not detected by DIVINE. The reason is that the models used the framework the way it was intended to be used, never exposing this particular scenario.

7 EXPERIENCES

In this section, we discuss the experiences we had with the tools, techniques, and languages used in this project.

7.1 Mapping C++ to PROMELA

The strength of PROMELA is its language level atomicity and synchronization concepts. This allows synchronization patterns to be expressed more concisely than in C++. The difference in the amount of code can be observed in the code examples above.

There are also programming patterns that cannot be nicely translated from C++ to PROMELA. For example, in C++ Thread is a class. The state of a Thread is represented as a (private) field (i.e. member in C++ parlance) within the class, which can be safely and conveniently accessed in other contexts through accessors. This sort of encapsulation is not possible in PROMELA, and Threads need to be modeled using other means.

We modeled Threads with processes and global arrays. In general, the field f is represented by an array a_f, with $a_f[i]$ storing the

value of f of the ith Thread. Such encoding clutters the model and moreover makes the model itself more error prone.

Not having methods, classes, or inheritance at disposal with Promela added extra challenges, because these programming language features had to be substituted with CPP macros and Promela inline blocks. This complicated debugging the model in this project. While the resulting model in this project is still of manageable size, for a larger project this mismatch may eventually lead to a maintainability problem with hand-written models.

The eight states and five commands of a Thread were modeled with mtype declarations in Promela. As mtype declarations share the same domain (of one byte in size), having two of them opened the possibility of erroneously assigning a state value to a command variable and vice versa. This risk had to be taken, since representing the two enumerated types with two distinct byte values might have grown the state space of the model. As mentioned, bytes cannot be pretty-printed which makes debugging less convenient than with mtypes. On the other hand, bytes can be bitmasked, which is convenient as we have seen. Bitmasks of states and commands are utilized in the C++ implementation of the framework.

7.2 Verifying C++ With Divine

This project initially used an older version of Divine 4, but had to change to the statically linked binary version 4.1.20+2018.12.17 because of a bug in the DiOS operating system of an older Divine version. The properties verified with Divine included the absence of deadlocks, crashes, memory acces violations, and memory leaks.

Divine 4 is composed of a specialized version of the LLVM Clang compiler, a Phtreads implementation, a C++ standard library implementation, a minimalistic operating system, DiOS, the DiVM virtual machine and a model checker [3, 24]. The Divine model checker only observes the state transitions of DiVM. This design allows the Divine model checker to also find errors in the operating system and libraries, in addition to verifying the application proper. A handful of defects in these software layers were found and reported to the Divine developers during this project.

As it turns out, model checking a non-trivial application with Divine might not be as simple as just compiling and verifying the model, even if Divine accepts C++ as an input language. It was found that C++ source code files must be compiled in a single invocation of the divine cc command. Libraries, other than the standard library or Pthreads can't be used, unless included in the same build with the user program.

The C++ codebase of ADAPRO had to be modified with macros, undocumented internal Divine compiler attributes (for optimisation) and DiVM hypercalls for getting it to work with Divine. In general, the extent of modifications required probably depends heavily on the code under verification. We needed to put effort into the conversion, largely because some parts of the Linux and POSIX APIs were missing from the Divine library layer. Thus, these parts of the code needed to be guarded by macros. The missing functionality included access to system clock, setting the nice value of the running process, and setting scheduling options such as thread affinities. We also had to disable many non-critical parts of the code, e.g. logging, a watchdog mechanism, and change most of constans into functions, to keep the state space size feasible.

7.3 State Space Explosion

The hardest challenge in this project was the notorious state space explosion problem. As discussed in the beginning of Section 5, we could only fully verify the Promela model with two workers.

For making verification with Divine possible, the implementation of ADAPRO needed several iterations of refactoring and optimisation to keep the state space size of even the small model manageable. One particular header file (headers/data/Parameters.hpp) had many global constant variables. In C++ global variables have internal linkage, which means that every translation unit gets their own unique copies of these variables, stored in different memory locations in the computer executing the compiled machine code. Among other optimizations, replacing these global variables with functions returning literals on demand, provided around 60% reduction in the size of DiVM states during verification runs.

The number of states was another aspect of the problem. For the small model, running divine check took around 5,000 DiVM states. When the medium-size model was finally checked successfully, the verification took almost 500,000 states. By the time of writing this paper, the large model and an example application involving disk I/O through a virtual filesystem image still remain to be checked.

Disabling the logger facility of ADAPRO and all other standard output and error stream operations also proved to make a huge difference in the size of the state space. Before disabling output operations and watchdog mechanisms, a verification run for the medium-size model on a virtual machine, in a cloud provided by SURFsara, had been running for more than 17 hours of wall clock time, consuming over 37 gigabytes, with no end in sight. After the optimisations, the medium-size model can now be checked in twenty minutes on a laptop, with the peak memory usage being in the order of seven gigabytes. Even after all optimisations, a virtual machine provided by SURFsara couldn't finish the verification run for the large model, given 124 GB of RAM dedicated to Divine.

7.4 Scalability

When formulating the Theory of ADAPRO, we found out that Spin was not able to digest the properties (1) – (8) in a single formula. We first tried to do this and Spin encountered a buffer overflow when parsing the property. On second thought, having one huge property would probably have been prohibitively expensive to verify anyway.

We found out, that even though Divine supports multi-threaded verification, the performance doesn't scale for an arbitrary number of threads. Especially on the virtual machines provided by SURFsara, the scaling proved to be unsatisfactory. As mentioned before, verifying the Promela model takes only a couple of minutes, so multi-threaded verification with Spin was not needed.

It seems that, for the ADAPRO models, two verification threads yield the best overall throughput in terms of DiVM states and instructions explored per second. The higher the number of threads, the higher the ratio of system time to user time. Thus, beyond the saturation point, increasing the number of threads seems to only increase the CPU time consumption and heat production. In fact, verification speed seems to slowly decline as a function of thread count. Divine developers were not aware of such performance issues, so our models might have been anomalous, or perhaps the virtual machine or the guest system was configured suboptimally.

7.5 Thoughts on the Learning Curves

All in all, the experiences with the Spin and Divine model checkers have been positive and encouraging in this project. Even though the Promela language was quite different from C++, it felt easy to learn. Likewise, after some initial learning effort, using Divine to check almost any standard C++ code proved to be easy. Even with their own challenges, using both of the model checkers was easy compared to tactic-based interactive theorem proving with HOL on Poly/ML or the use of propositions-as-types interpretation of intuitionistic type theory in Agda.

Of course, programmers need to get at least some degree of familiarity with the basics of enumerative model checking, in order to understand how Spin and Divine are best exploited. We estimate that it'd take a few weeks of training to acquaint developers with a few years of experience in programming and some mathematical maturity, into the use of these model checkers. Perhaps the effort could be compared with learning a new programming language.

We found the output of Spin model checker harder to understand than the reports generated by static analysis, testing, and code coverage tools, or the Divine model checker. Furthermore, the semantical gap between Promela and C++, especially in terms of their memory models and atomicity, is considerable. This might introduce inaccuracies to hand-made translations between C++ and Promela. Luckily, this was not a problem since we also performed model checking on the very C++ source code with Divine.

7.6 Impact on Workflow

Installing the Spin and Divine model checkers was easy and straightforward. The Debian GNU/Linux operating system has Spin included in its standard package repository, so the installation is just a matter of running `apt-get install spin`. Using the precompiled binary version of Divine is just as easy. We considered it sufficient for this project to run Divine by invoking a custom BASH runner script.

Thanks to their simple command-line interfaces, integrating Spin and Divine into the workflow of the software project was fast and easy (after the initial work spent in learning the correct flags and arguments). These tools don't require maintaining complicated configuration files, but their use can be fully parameterized and automated with regular shell or makefile scripts, for example. Setting up the integration takes maybe even less time than with an average software tool.

After integrating the model checkers into the project workflow, they can be used without hassle on daily basis, just like any other analysis or testing tools. Model checking unit tests with Divine was found especially useful, because that way, existing test cases can be explored exhaustively. As noted above, sheer code coverage is not enough to ensure exhaustiveness of tests. Developing models and implementation in parallel seems to be an efficient analogue of test-driven development.

8 CONCLUSION

The LTL properties listed in Section 4.2, should, from now on, be considered an integral part of the specification of ADAPRO. Model checking was found to be a valuable addition to software developer's toolbox, with a good return of investment value (which already seems to be the consensus in the literature). It seems that different model checkers can find different kinds of issues, on different levels of abstraction (e.g. design/implementation). Many of the issues found during this project require rather specific circumstances to occur, making them nearly impossible to test, but when they do occur, they may trigger chain reactions of dire consequence.

We believe that easy integration to the everyday workflow is important for any software tool to become successful in the industry. In terms of automation possibilities, it seems that we're already getting there with model checkers. However, APIs, output formatting, and documentation still may to need extra polishing, not to mention IDE support. We believe that these points need to be addressed, before formal methods can attract the attention of mainstream programmers.

The benefits of model checking far outweigh the slightly rough edges of the model checkers. The shortcomings could be mitigated, with more resources invested in the development of formal verification tools. The sheer number of rather simple defects that were found in supposedly stable code, only after deploying model checkers, raises a question: *Can software, that has not been formally verified, be trusted?* We doubt it.

8.1 Future Prospects

The effect on fairness imposed by user Threads getting exclusive access to CPU cores through the use of the Pthreads API, has not been explored. As mentioned before, configuration access and Interaction with external systems using signals, the DIM protocol, and the Linux systemd API was not modeled. Investigating these aspects might prove useful, albeit challenging.

Even though the Promela model was successfully verified with two workers, there's no hard mathematical argument showing that two workers suffice. Decreasing the granularity of the Promela model might allow more workers to be simulated. Using a swarm-based approach[16] might turn out to be a useful bug-hunting technique with larger models for both Spin and Divine.

The Divine models still deserve more attention. Firstly, measuring the effects of different kinds of code optimizations on the number and size of Divine states would be an interesting topic for a quantitative study. Secondly, using monitors to perform LTL model checking on ADAPRO was not yet attempted, because our Divine models were too large even without LTL properties. Thirdly, verifying ADAPRO under a weak memory model was also not attempted, since Divine does not yet support such a feature.

REFERENCES

[1] Borja Fernández Adiego, Dániel Darvas, Jean-Charles Tournier, Enrique Blanco Viñuela, and Víctor M. González Suárez. 2014. Bringing Automated Model Checking to PLC Program Development — A CERN Case Study —. *IFAC Proceedings Volumes* 47, 2 (2014), 394 – 399. https://doi.org/10.3182/20140514-3-FR-4046.00051 12th IFAC International Workshop on Discrete Event Systems (2014).

[2] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking.* MIT Press.

[3] Zuzana Baranová, Jiří Barnat, Katarína Kejstová, Tadeáš Kučera, Henrich Lauko, Jan Mrázek, Petr Ročkai, and Vladimír Štill. 2017. Model Checking of C and C++ with DIVINE 4. In *Automated Technology for Verification and Analysis (ATVA 2017) (LNCS)*, Vol. 10482. Springer, 201–207.

[4] Mordechai Ben-Ari. 2008. *Principles of the Spin Model Checker.* Springer. https://doi.org/10.1007/978-1-84628-770-1

[5] S. Chandra, P. Godefroid, and C. Palm. 2002. Software model checking in practice: an industrial case study. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002.* 431–441.

[6] Zhe Chen, Yi Gu, Zhiqiu Huang, Jun Zheng, Chang Liu, and Ziyi Liu. 2015. Model checking aircraft controller software: a case study. *Software: Practice and Experience* 45, 7 (2015), 989–1017. https://doi.org/10.1002/spe.2242 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2242

[7] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. 2002. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002) (LNCS)*, Vol. 2404. Springer, Copenhagen, Denmark.

[8] P. de la Cámara, M. M. Gallardo, P. Merino, and D. Sanán. 2005. Model Checking Software with Well-defined APIs: The Socket Case. In *Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems (FMICS '05)*. ACM, New York, NY, USA, 17–26. https://doi.org/10.1145/1081180.1081184

[9] B Abelev et al and. 2014. Upgrade of the ALICE Experiment: Letter Of Intent. *Journal of Physics G: Nuclear and Particle Physics* 41, 8 (jul 2014), 087001. https://doi.org/10.1088/0954-3899/41/8/087001

[10] John Fitzgerald, Juan Bicarregui, Peter Gorm Larsen, and Jim Woodcock. 2013. *Industrial Deployment of Formal Methods: Trends and Challenges.* Springer Berlin Heidelberg, Berlin, Heidelberg, 123–143. https://doi.org/10.1007/978-3-642-33170-1_10

[11] B. Franek and C. Gaspar. 2004. SMI++ object oriented framework for designing and implementing distributed control systems. In *IEEE Symposium Conference Record Nuclear Science 2004.*, Vol. 3. 1831–1835 Vol. 3. https://doi.org/10.1109/NSSMIC.2004.1462600

[12] Xiang Gan, Jori Dubrovin, and Keijo Heljanko. 2014. A symbolic model checking approach to verifying satellite onboard software. *Science of Computer Programming* 82 (2014), 44 – 55. https://doi.org/10.1016/j.scico.2013.03.005 Special Issue on Automated Verification of Critical Systems (AVoCS'11).

[13] C. Gaspar, M. Dönszelmann, and Ph. Charpentier. 2001. DIM, a portable, light weight package for information publishing, data transfer and inter-process communication. *Computer Physics Communications* 140, 1 (2001), 102 – 109. https://doi.org/10.1016/S0010-4655(01)00260-0 CHEP2000.

[14] Gerard J. Holzmann. 1997. The model checker SPIN. *IEEE Transactions on Software Engineering* 23, 5 (May 1997), 279–295. https://doi.org/10.1109/32.588521

[15] Gerard J. Holzmann. 2001. Economics of Software Verification. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '01)*. ACM, New York, NY, USA, 80–89. https://doi.org/10.1145/379605.379681

[16] Gerard J. Holzmann, R. Joshi, and A. Groce. 2008. Swarm Verification. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. 1–6. https://doi.org/10.1109/ASE.2008.9

[17] Gerard J. Holzmann and M. H. Smith. 1999. A practical method for verifying event-driven software. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*. 597–607. https://doi.org/10.1145/302405.302710

[18] Yi Ling Hwong, Jeroen J.A. Keiren, Vincent J.J. Kusters, Sander Leemans, and Tim A.C. Willemse. 2013. Formalising and analysing the control software of the Compact Muon Solenoid Experiment at the Large Hadron Collider. *Science of Computer Programming* 78, 12 (2013), 2435 – 2452. https://doi.org/10.1016/j.scico.2012.11.009 Special Section on International Software Product Line Conference 2010 and Fundamentals of Software Engineering (selected papers of FSEN 2011).

[19] Ranjit Jhala and Rupak Majumdar. 2009. Software Model Checking. *ACM Comput. Surv.* 41, 4, Article 21 (Oct 2009), 54 pages. https://doi.org/10.1145/1592434.1592438

[20] Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* C-28, 9 (Sep. 1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

[21] Leslie Lamport. 2002. *Specifying systems: the TLA+ language and tools for hardware and software engineers.* Addison-Wesley.

[22] J.L. Lång et al. 2018. ADAPOS: An Architecture for Publishing ALICE DCS Conditions Data. In *Proc. of International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'17), Barcelona, Spain, 8-13 October 2017 (International Conference on Accelerator and Large Experimental Control Systems)*. JACoW, Geneva, Switzerland, 482–485. https://doi.org/10.18429/JACoW-ICALEPCS2017-TUPHA042

[23] Amir Pnueli. 1977. The temporal logic of programs. , 46-57 pages. https://doi.org/10.1109/SFCS.1977.32

[24] Petr Rockai, Ivana Cerná, and Jiri Barnat. 2017. DiVM: Model Checking with LLVM and Graph Memory. *CoRR* abs/1703.05341 (2017). arXiv:1703.05341 http://arxiv.org/abs/1703.05341

[25] Kristin Y. Rozier. 2011. Linear Temporal Logic Symbolic Model Checking. *Computer Science Review* 5, 2 (2011), 163 – 203. https://doi.org/10.1016/j.cosrev.2010.06.002

[26] Jeannette M. Wing and Mandana Vaziri-Farahani. 1997. A case study in model checking software systems. *Science of Computer Programming* 28, 2 (1997), 273 – 299. https://doi.org/10.1016/S0167-6423(96)00020-2 Formal Specifications: Foundations, Methods, Tools and Applications.