Public Certification of Software and its necessity in Computable Laws

FV Time as the first application

Mireia González Bedmar
Guillermo Errezil Alberdi

Formal Vindications S.L.

28 April 2022

Proyecto RTC6740-2017–7 financiado por MCIN/AEI/10.13039/501100011033 y por FEDER Una manera de hacer Europa

The result of 5 years of research

Public Certification of Software: A new concept to reach higher certification standards

FV Time Library: a complete and succesful publicly certified software

A historical example

A bill in Kansas contained the following sentence:

Excerpt of a bill in Kansas

When two trains approach a crossing, both shall stop, and neither shall go ahead until the other had passed by.

But what can be done if this is in a law enforced by software?

1. Follow the law and have the trains stop forever.
2. Self-fix: For example by giving preference to a train on one side. The software would be **breaking the law**, and **the software engineers would be legislating at their will** (consciously or otherwise).
3. Engage with the policy makers to amend the law.

Everyone is taking path 2. We are here to show how 3 can be done.

Questions

When implementing a law, two questions arise.

- Is it possible to implement in code anything that is written in natural language?
- Should the IT engineers make decisions and become the real lawmakers?

Motivation

The precursors of our team worked in the transportation sector: truck drivers, tachographs...

We call **computable law** any law that is meant to be implemented as software. In our sector we have two EU regulations:

- R. 3821/85 Update to the digital tachograph: a technical specification on how to build it
- R. 561/06 Determination of several kinds of infraction in driving times

Engineers can implement both regulations and then...

Finally, there would be no room for interpretation of the law and logs. Software can do this for us!

End of story?

From second to minute resolution

All European tachographs need to define and count driving time according to computable law 3821/85:

(50) This function shall output activity changes to the recording functions at a resolution of one minute.

(51) Given a calendar minute, if DRIVING is registered as the activity of both the immediately preceding and the immediately succeeding minute, the whole minute shall be regarded as DRIVING.

(52) Given a calendar minute that is not regarded as DRIVING according to requirement 051, the whole minute shall be regarded to be of the same type of activity as the longest continuous activity within the minute (or the latest of the equally long activities).

According to our investigation, no tachograph follows this law.

Industrial Software Homologation: Theory and case study Analysis of the European tachograph technology with EU transport Regulations 3821/85, 799/2016, and 561/06 and their consequences for Europeans citizens

Conclusion

Conclusion: we are at the Kansas train scenario.

- The law in natural language is not compatible with mathematics and hence, coding.
- Engineers become the real lawmakers.

It's all the way down...

This happens with EVERY specification given in natural language.

There always are **inconsistencies** and **ambiguities**.

No 100% guarantee that your natural-language specification has one and only one interpretation.

Example of infraction

Article 7

After a driving period of four and a half hours a driver shall take an uninterrupted break of not less than 45 minutes, unless he takes a rest period.

This break may be replaced by a break of at least 15 minutes followed by a break of at least 30 minutes each distributed over the period in such a way as to comply with the provisions of the first paragraph.

Article 7 in code

How can the equivalence between Article 7 and this code be proven? Simply impossible.

```
private int GetMaximumPauseIx(List<BaseEvent> simplifiedActivities, int beginEventIx, int endEventIx)
{
    int maximumPauseIx = -1;
    TimeSpan maxPauseDuration = TimeSpan.Zero;

    for (int i = beginEventIx; i <= endEventIx; i++)
    {
        BaseEvent item = simplifiedActivities[i];
        if (item.SimpliActivity == SimplifiedActivity.SPause)
        {
            if (maxPauseDuration < item.Duration)
            {
                maximumPauseIx = i;
                maxPauseDuration = item.Duration;
            }
        }
    }
    return maximumPauseIx;
}
```

What does software really do?

We live in the era of software.

Some software has a critical use: DNA sequencing, auto-pilot of a plane, analysing tachograph registered data...

How can we prove software does what it's supposed to do?

How can we even express what software is supposed to do?

Our solution

We present:

1. The notion of Formal Verification of Software.
2. One step further: the notion of Public Certification of Software.
3. Our own concrete model to reach it.
4. A particular example using that model: FV Time.

Formally verified software

Formally Verified Software Components

- Σ Specification: mathematical description of what the software is supposed to do
- Π Implementation: the code, i.e., the actual steps the software performs
- Δ Proof of correctness: mathematical proof that Π meets Σ

But the formal specification Σ is written in a formal language (logic/mathematics). So...

How to make software specifications accessible?

Our solution

Publicly Certified Software must satisfy the following requirements:

1. It is based on a formal specification (in a formal language).
2. The formal specification is mathematically proven to follow all basic mathematical properties (unambiguity, consistency...).
3. The formal specification follows physical and computational limits (it is a representation of some part of the physical world and can be computed).
4. The software is mathematically proven to follow the formal specification.
5. The formal specification has an equivalent formulation (an interpretation) closer to natural language.
6. The previous items can be checked by anyone with enough knowledge, and it has been done at least once by some party.

If a project is limited to 1-4, then we can call it "privately certified" by means of formal verification.

Our particular model

Our concept of Publicly Certified Software is given by a $(4 + 1)$-tuple $(\Sigma, \Pi, \Delta, \Lambda, \Phi)$, where:

Publicly Certified Software Components

- (Σ, Π, Δ) **Formally verified** triple written in Coq
- Λ Formally verified **extraction** to a language suitable for computation (OCaml)
- Φ **Interpretation** in several abstraction-levels of language (from natural to formal language)

Σ is written formally: it can be proven but not understood by most.

Φ is a version of Σ written for **human understanding**.

Φ is a guide on how to understand Σ.

Problem

Φ needs to solve the most important issues of Formal Verification.

- A formal-language specification is not usually directly understandable for humans. As our CEO puts it:

> "If you understand it, then you can't prove it.
> If you can prove it, then you can't understand it."

- **The intuitive specification we have in mind** when designing software may or **may not correspond to the formal specification** we write. Same if we write a natural-language version of the specification.
  1. Natural language can be and is ambiguous.
  2. The gap between different abstraction-levels of language cannot be completely overcome. No mathematical proof! However, public certification can narrow it considerably.

Then how can we jump from Matrix to the real world?

Time Managers

But remember... We were implementing computable law 561.

We needed to manage time: conversions, durations...

What is a Time Manager?

A software that deals with dates and times in different formats.

- Convert from a timestamp to date format
- Define a calendar and the correspondence between timestamps and calendars
- Arithmetical operations: add seconds, add hours to a date, compute durations between 2 dates...

Time managers are everywhere: Microsoft, Linux, Apple, Android. Why did we decide to make our own?

What is UTC?

UTC is the *de facto* time standard.

Many laws enforce its use (including EU 561/06, EU 3821/85, USA 49 CFR Part 395), NASA enforces its use.

Regulations are in UTC.

UTC is based on atomic clocks that count the number of seconds. It has a particularity, **leap seconds**. There have been 27 leap seconds as of today.

Microsoft, Android, iOS, Linux...

None of them follow UTC.

Kansas scenario: software engineers are making the legal decisions.

"Just 27-second difference, no big deal"? Our team proved that, when applying Requirements (51)-(52) (see page 6), the same driver logs can give 100% driving in Unix and 0% in UTC, and vice versa.

FV Time

Nobody was following the law, so we decided to make our own time manager.

First Publicly Certified Software for dealing with UTC conversions and arithmetic.

FV Time includes:

1. Conversion between date-time (`Y-M-D h:m:s`) and timestamp (the number of seconds elapsed since 1-1-1970 00:00:00).
   The UTC standard calendar is the **Gregorian calendar**.
2. A standard for durations, `formalTime`, defining units like formal minutes, hours, days...
   There are no standards for durations (time spans). In UTC any unit above the second has variable duration (e.g., minutes can have 61 seconds!), so we made a consistent standard: **Formal calendar**.

Formal time

In Unix “add 1 month” is not well defined because months have different durations. But “add 1 day/hour/minute” is okay.

In UTC none is well-defined. Due to leap seconds, a minute can have 59, 60 or 61 seconds! Similarly hours, days...

Thus we defined the formal time units:

- 1 formal minute = 60 seconds
- 1 formal hour = 60 formal minutes
- 1 formal day = 24 formal hours
- 1 formal month = 30 formal days
- 1 formal year = 365 formal days

FV Time equivalence with Microsoft DateTime

Microsoft method	**FV update to UTC**	**FV new algorithm** with arithmetical properties: 1 – 1 = 0
`AddDays`	`shift_utc_days`	`add_formal_days`
`AddHours`	`shift_utc_hours`	`add_formal_hours`
`AddMinutes`	`shift_utc_minutes`	`add_formal_minutes`
`AddMonths`	`shift_utc_months`	`add_formal_months`
`AddSeconds`	`shift_utc_seconds`	`add_formal_seconds`
`AddYears`	`shift_utc_years`	`add_formal_years`

AddMonths
<2019-01-31 14:00:00> 1
= <2019-02-28 14:00:00>

shift_utc_months
<2019-01-31 14:00:00> 1
= <2019-02-28 14:00:00>

add_formal_months
<2019-01-31 14:00:00> 1
= <2019-03-02 14:00:00>

AddMonths <2019-02-28
14:00:00> -1 =
<2019-01-28 14:00:00>

shift_utc_months
<2019-02-28 14:00:00> -1
= <2019-01-28 14:00:00>

add_formal_months
<2019-02-28 14:00:00> -1
= <2019-01-31 14:00:00>

Specification vs implementation

The implementation or code is a set of instructions for the computer, it performs concrete operations.

Looking at the code, one cannot know what the software is supposed to do.

The specification must express what the software is supposed to do in a conceptual way. Not operations, but concepts! Human-friendly.

However, the formal specification is in formal language (hard to understand for humans).

Let's see an example.

Example

Suppose we want to define the function `utc_timestamp_plain`.

First intuitive specification

Given a time t, returns the number of seconds elapsed since the Unix epoch (1-1-1970 00:00:00).

Example: Implementation

Our code for that is:

```
Definition utc_timestamp_plain (t : time) : uint :=
  timestamp hardcodedls t - timestamp hardcodedls Unix_epoch.
```

where

```
Definition timestamp (ls : list (date * B )) (t : time) : uint :=
  let Dts := datestamp t * 86400 in
  let hts := hour t * 3600 in
  let mts := minute t * 60 in
  Dts + hts + mts + second t + offset_rd ls (leapSeconds_offsets ls) t.

Definition Unix_epoch := Time (Date 1970 1 1) 0 0 0.
```

where

```
Definition datestamp (d : date) : uint :=
  let y' := year d - (month d <=? 2) in
  let era := y' / 400 in
  let yoe := y' mod 400 in
  let m' := month'_of_month (month d) in
  let doy := first_day_of_month' m' + day d - 1 in
  let doe := doe_of_yoe doy yoe in
  era * 146097 + doe - 306.
```

etc.

Example: Specification

We could write a specification very similar to the code and show that they are equivalent.

But what happens then with the intuitive, natural-language specification?

Can we say that the code meets the natural-language specification?

No way. They are too far away.

Let's take another approach.

Example: Formal specification

Our formal version of the specification says:

Formal specification in Coq

```
#|[pred t' | (epoch <= t' < t)%O]|
```

This can be read as:

Formal specification in a descriptive level of language

Given a time *t*, returns the cardinality of the set of times that are equal or after the Unix epoch (1-1-1970 00:00:00) and before *t*.

Now an expert can honestly say that this corresponds to the intuitive specification we had:

First intuitive specification

Given a time *t*, returns the number of seconds elapsed since the Unix epoch (1-1-1970 00:00:00).

Interpretation: abstraction-levels of language

Public certification exists to make sure that, in a particular project:

- Formal verification has been indeed done, and
- There is a correspondence between the formal specification and the natural language claims about the software.

To do that our guidelines ask that for every function, besides the formal specification, descriptions in different levels of language are given:

0. **Pure natural language**: accessible for non-experts.
1. **Hybrid language**: acceptable in a mathematics textbook.
2. **Mathematical language**: acceptable in rigorous formalization.

Interpretation

Example from FV Time:

Specification of `utc_timestamp_plain` (version with no error handling)

A. Descriptive language, abstraction level 0: pure natural language

A function which takes a `time` and returns the number of seconds elapsed since the `Unix_epoch` in UTC to that time.

B. Descriptive language, abstraction level 1: hybrid

A function named `utc_timestamp_plain` with input a time t and output a natural number representing the cardinality of the set of valid times t' such that $t' < t$.

Interpretation

C. Descriptive language, abstraction level 2: mathematical language

A function

$$\texttt{utc_timestamp_plain} : \texttt{time} \to \texttt{uint}$$

such that $\forall t$: `time`, `valid_time` t:

$$\texttt{utc_timestamp_plain}\ t = |\{t' \mid \texttt{valid_time}\ t' \wedge t' < t\}|,$$

where $<$ corresponds to the order `lt_time_plain`.

Σ (Coq)

`utc_timestamp_plain`

- **`utc_timestamp_plainR`**

In summary

Checked by human experts

Mathematical proof, computer-checked

Interpretation Φ (different descriptive levels of language)

Human-like: conceptual, expressed for non-experts

Formal specification (formal language)

Math-like: abstract, conceptual, expressed for experts

Implementation (code)

Machine-like: operations, efficiency, expressed for computers

Public Certification

Formal Verification

Future work

We are currently working on:

1. The world-wide public certification of the transportation regulations for hours of service and driving time. Of course FV Time is embedded. For this project it is extremely critical to jump from formal specification to natural language because drivers and enforcement agencies need to understand the law.

> Every computable law and technical specification affecting civil rights should go through the process of public certification, for a more just society.

2. The formal verification of the SHA256 algorithm, as it appears specified by the National Institute of Standards and Technology of the USA. In this case the jump to natural language is not possible because the specification is purely mathematical. Other levels of language are central.

Contact details

FV webpage

FV's general contact:
info@formalv.com

Speaker's contact:
mireia.gbedmar@formalv.com