The Kopetz Principle

Edgar Daylight

13 September 2019

Abstract

"Mind the gap!" says the engineer to the computer scientist; that is, the gap between a mathematical model of a physical system on the one hand and the system itself on the other hand. This cautionary remark refers to what is called *the Kopetz Principle*. The principle comes from Kopetz, Parnas, Lee, and other engineers and is often (ineffectively) conveyed to theoretically-inclined researchers in formal verification. Historically, the principle came to prominence independently of similar developments made in the philosophy of computing (e.g. by Fetzer and Colburn). In the present document I share my experience as a safety engineer in industry to underscore the relevance of the Kopetz Principle and hint at the need to examine the principle more closely, both by historians and philosophers. Feedback from the reader is most welcome, also and especially if s/he opposes the distinction between abstract and physical objects.

1 Personal Anecdote

I started my consultancy work in 2017 as a safety engineer at a reputable company. I relied on Nancy Leveson's *Engineering a Safer World* [6] and conducted a Hazard & Risk Analysis. Based on the company's remote-control device, its electric vehicle, documentation, `C` code, and many oral discussions, I derived various new hazards for the company. One hazard pertains to a highly unreliable wireless channel:

> Outdated steering commands, sent from the remote-control device to the vehicle are not discarded, causing unpredictable system behavior.

This hazard led to a new safety specification for the company:

> The vehicle shall not execute outdated steering commands.

Subsequently, I put on my mathematical hat in order to formalize the notion of "outdated" steering command. Following Peter Naur's personal advice (from 2011), I chose a formal method that was not necessarily familiar to me but which is highly appropriate for the problem in hand; i.e., a process algebra, `mCRL2`, which allowed me to model the communication channel, along with a

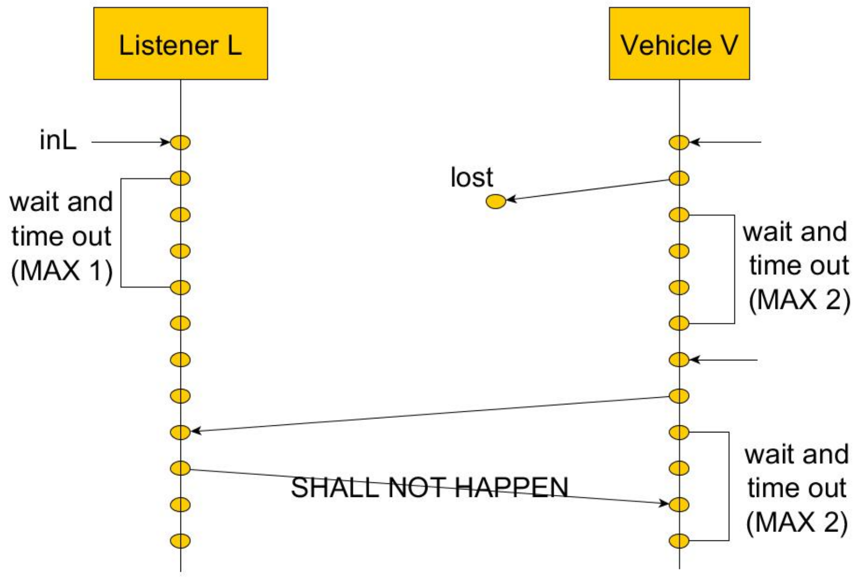
Listener L
Vehicle V
inL
lost
wait and time out (MAX 1)
wait and time out (MAX 2)
wait and time out (MAX 2)
~~SHALL NOT HAPPEN~~

Figure 1: Sequence diagram. Time increases as events take place from top to bottom.

Listener L process (running on the remote-control device) and a Vehicle V process (running on the vehicle). On the one hand, L repeatedly receives steering advice from the user of the remote-control device. That advice needs to cross the channel and reach V. On the other hand, V frequently receives sensor values from the vehicle (including speed and acceleration parameters) and needs to send these to L. Eventually, I defined process L like this:

$$L = inL \cdot LWait(MAX1) + listen \cdot L$$
$$LWait(0) = i \cdot L$$
$$L(Wait(0 < n) = i \cdot LWait(n-1) + listen \cdot (i + i \cdot request) \cdot L \qquad [0,1]$$

For experts, this algebraic specification is almost, but not quite, in compliance with the official `mCRL2` handbook [3]. The previous specification captures:

- The possibility that packets sent from L to V on the *request* channel get lost.
- L's countdown mechanism, starting from $MAX1$ and counting to 0.
- A priority list $[0,1]$ on the last line, stating that the first summand has lower priority than the second.

Based on several more discussions, I derived a definition for process V, and finally placed L and V in parallel composition: $L \parallel V$.

The result of using the `mCRL2` tools on my model $L \parallel V$ is remarkable. For, I obtained three different notions of what an "outdated" steering command can actually mean in practice. For instance, one tool allowed me to derive the diagram in Figure 1, in which a packet (containing vehicle parameters) sent from V to L gets *lost*. Thanks to the inclusion of the $MAX1$ time-out on the L side, the steering advice received via inL from the human user is considered *obsolete* when a *later* packet from V does arrive on L (thereby granting L the freedom to piggy back steering advice). The result is that the obsolete advice is *not* sent over the channel to V, as is desirable.

To cut to the chase, the $MAX1$ time-out is the first safety mechanism I added to the company's software. Similar investigations led to a $MAX2$ time-out mechanism on process V. Moreover, based on model checking, these two precautions still turned out to be insufficient. I also had to add time stamps to the packets that are sent from V to L, and which eventually piggy back to V. Then the real-time clock on the vehicle is consulted in order to compare the current real time with the received time stamp, and so on. In a word, mathematical reasoning helped the company a lot, even allowing the engineers to contemplate the superfluousness of the second safety precaution, given that the first and the third suffice in theory.

But, using a particular model only gets you so far. In a later phase of consultancy I started scrutinizing the assumptions underlying my model $L \parallel V$, and began to draw sequence diagrams manually, based on the realistic possibility that the remote-control device was made by malicious hackers instead of the company's engineers. Specifically, I examined the scenario in which a malicious Listener L process repeatedly and *ad infinitum* sends steering requests to the Vehicle V process. V has to be able to deal with this *denial-of-service* attack in some way. Further contemplation led to the inclusion of authentication techniques, which again can be modeled with `mCRL2`, but which also result in a far less simple overall solution, given that V has to meet hard real-time constraints. Even after including authentication, it is easy to contemplate detrimental scenarios. For example, one malicious hacker could emulate $10,000$ different L processes and incite another denial-of-service attack.

2 The Kopetz Principle

My consultancy work led to two concerns that I wish to disseminate for obvious, safety-related reasons. My first concern (mentioned only in passing) is about the deployment of self-driving cars that are *supposed* to be *safer* than the vehicles we use today. Yet here is my most worrisome hazard in this regard:

> A seemingly insignificant software error in an automated, connected car, can — when detected by a malicious hacker — result in a crash of that car and thousands of other crashes.

Having cars mechanically attached to railroads is one of many ways to at least reduce the impact of this hazard. Mechanically and drastically constraining the cars' maximum speed, is yet another.

My second concern is about '*complete* formal verification of a cyber-physical system.' (If that *were* feasible, then my most worrisome hazard could perhaps be resolved.) Besides observing that most automotive companies hardly use formal methods in stark contrast to the aviation industry, there are at least two reasons to be highly skeptical about the viability of complete verification. First, coming up with all hazards and related specifications is a well-known and unresolved problem: the lack of specifications makes the notion of formal proof meaningless [7, p.1334]. Second, I introduce (and shall elaborate on) the **Kopetz Principle**:

> Many (predictive) properties that we assert about systems (determinism, timeliness, reliability, and correctness) are in fact not properties of an implemented system, but rather properties of a *model* of the system.
>
> We can make definitive statements about *models*, from which we can infer properties of system realizations. The validity of this inference depends on *model fidelity*, which is always *approximate*.

With the exception of the underlined words which I have added myself, this fragment comes from Edward Lee's plenary talk 'Verifying Real-Time Software is Not Reasonable (Today),' presented at the Haifa Verification Conference in 2012. The fragment is Lee's paraphrasing of yet another expert in real-time systems: Hermann Kopetz.

The Kopetz Principle has been conveyed many times in the past, and not only with regard to model checking, but also machine-checked proofs of, say, `C` code. My favorite example is the Chaudhuri-Parnas exchange. Chaudhuri et al. have studied properties of programs that are necessary conditions for a correct program to possess in their mathematical framework. In their article 'Continuity and Robustness of Programs' [1], Chaudhuri et al. discuss a way to prove mathematical continuity of programs. But, to put it candidly, Chaudhuri et al. do not seem to be aware that they are not proving something about their programs, but that they are proving something about some model of their programs. This is pointed out by Parnas in a reaction to their article. Parnas writes:

> Rather than verify properties of an actual program, it examined models of programs. Models often have properties real mechanisms do not have, and it is possible to verify the correctness of a model of a program even if the actual program will fail. [8]

Similarly, while I have praised the `mCRL2` work of Jan Friso Groote, I now take the liberty to question his philosophy. In his words:

> The model can be investigated to prove that the system always satisfies certain requirements [2, p.4].

This fragment illustrates a conceptual oversight between the properties proved on a mathematical model on the one hand and the actual, running system on the other hand. Finally, the Kopetz Principle can also be used to scrutinize the following common line of reasoning in formal verification:

> [Static verification] is done on the basis of models of the program code or on the code itself. [4, p.614]
>
> [Modal logics] operate on idealized system models rather than on actual code. [4, p.616]

But the "actual code" is merely a textual representation of another *model* of the cyber-physical system in hand. Or, as Lee stresses in his talks, *a C computer program is a model* too and *not* a system realization [5, p.4839].

To recapitulate: the best we can do is prove properties of one or more mathematical models of the running system, *we cannot guarantee correctness of the cyber-physical system itself.* That's why my most worrisome hazard pertaining to automated cars (and, by extension, to drones and the like) should be worrisome to all of us. If there is one law about software, it is that we can never be sure that it is bug-free, let alone 100% secure. Having bugs in a text editor or an unmanned Mars Lander is less problematic than in fast moving objects that we will use on a daily basis.

Computer scientists can continue searching for their holy grail: *complete* formal verification of engineered artifacts. And although we will never find it, better tools will surface on our endless quest for perfection. Unfortunately, these very tools can be used by hackers to find vulnerabilities in, say, a communication protocol of a self-driving car. Groote has illustrated precisely this possibility when he found a deadlock in the sliding window protocol and after that protocol had been deployed and repeatedly analyzed for over a decade [3, p.107]. I take this and other feats in formal verification to support my skepticism as regards to the alleged viability of self-driving cars and what have you.

Acknowledgments

I thank Edward Lee and Michael Jackson for commenting on a draft version of this document.

References

[1] S. Chaudhuri, S. Gulwani, and R. Lublinerman. Continuity & robustness of programs. *Communications of the ACM*, 55(8):107–115, 2012.

[2] J.F. Groote, T.W.D.M. Kouters, and A. Osaiweran. Specification guidelines to avoid the state space explosion problem. In *Software Testing, Verification and Reliability*, volume 25, pages 4–33. John Wiley and Sons, Ltd., 2014.

[3] J.F. Groote and M.R. Mousavi. *Modeling and Analysis of Communicating Systems.* MIT Press, 2014.

[4] M. Huisman, H. Bos, S. Brinkkemper, A. van Deursen, J.F. Groote, P. Lago, J. van de Pol, and E. Visser. Software that meets its intent. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*, volume ISoLA 2016. Lecture Notes in Computer Science, vol 9953. Springer, 2016.

[5] E.A. Lee. The past, present and future of cyber-physical systems: A focus on models. *Sensors*, 15:4837–4869, 2015.

[6] N.G. Leveson. *Engineering a Safer World: Systems Thinking Applied to Safety*. MIT Press, January 2012.

[7] D.L. Parnas. Software aspects of strategic defense systems. *Communications of the ACM*, 28(12):1326–1335, December 1985.

[8] D.L. Parnas. On Proving Continuity of Programs. *Letter to the Editor of the Communications of the ACM*, 55(11):9, November 2012.