FROM LINEAR SYSTEMS TO DISCRETE-EVENT SYSTEMS

ECE Dept, U. of Toronto, 2012.03.22

Thank you for turning up, on this nice summer's day.

I'm going to talk about control theory and design. Probably to most people here, "control" means differential equations, Laplace transforms, Nyquist plots, and so on. Good things, of course.

***

The talk today is about none of these, and almost no background is needed at all. Generally speaking, it's about "cybernetics", whatever that is.

The "cyber" prefix is quite popular nowadays, so you have "cyberspace", and so on, with whatever meaning you want to give it.

***

The root word "kubernetes" goes back around 3000 years. What the Greeks meant by it was simply "steersman", as in this Homeric ship. The kubernetes is at the stern, with a double steering paddle.

***

"Cybernetics" came into English with Norbert Wiener's book in 1948, much cited – but not so widely read.

Its main theme is the universality of feedback.

***

A few years later, “cybernetics” reappeared in the title of Qian Xuesen’s book. Qian pinned down “engineering cybernetics” as an “engineering science”. That concept inspired my outlook as a graduate student at Cambridge in the late 50’s, and the activity of many researchers in control ever since.

***

Moving on, this talk is about a control theory for discrete-event systems.

Discrete-event systems are things like manufacturing systems, traffic systems, logistic systems, and so forth.

The "events" might be: a workpiece arrives at a workcell, a machine breaks down, or a vehicle enters a queue at an intersection.

The systems are event-driven rather than clock driven, with state spaces that are discrete rather than continuous.

So for a while they lay outside the traditional domain of control theory.

On the other hand, standard control was extremely suggestive in getting started.

Here's some history, starting around 1980.

***

In discrete-event systems, there were lots of practical problems, and an active community doing computer modeling and simulation.

From operations research to software engineering, there was a raft of techniques, and a hugely varied literature.

Formal methods in today's sense were just beginning to appear, but were only being done then by a few specialists.

***

In all this, you could see that control problems were certainly implicit.

But for instance, performance issues and logic issues tended to be mixed up together.

Also, systems reasoning tended to be done directly with programming models.

There was little by way of a standard control model or systematic approach to control design.

***

In a snapshot about 1980 you'd see a well-established catalog of control ideas.

In the state-space framework for controlled dynamic systems, the key concepts were, and still are: stability, controllability, observability and optimality.

Controllability is a property of a system that enables you to improve its dynamic behavior using feedback.

Observability guarantees that you can deploy such feedback by processing the system's output.

And optimality gives rise to formal methods of control synthesis.

In an ideal world, what you do is define the control problem formally, check for controllability and observability, press the button, and you're done.

In some ways control synthesis can be seen as building dynamic invariants – for instance like rotating an unstable subspace of the state space into a suitable hiding place, where you can’t see it at the system output and it doesn’t affect tracking error.

Here partial order by subspace inclusion is important, and leads to the notion of supremal subspaces - that are as large as possible within the problem constraints.

So that was our point of view from control.

***

We felt there was a need for a control approach, with some general plant model that was discrete, asynchronous, nondeterministic, and supported synthesis.

Optimistically, it would also be useful in design.

***

What we came up with was not startlingly new.

Finite automata for the system components, and regular languages for the system behavior.

Finally, a simple control technology, inspired by the Petri net idea of enabling or disabling the “firing”, or execution of transitions.

***

By the way, automata were invented by Hephaistos, the Greek god of engineers. Here are Hephaistos' party tables, making up a team of autonomous robots, and a "wonder to behold". That was 3000 years ago.

***

The initial response to our efforts was somewhat lacking, shall we say, in warmth.

Some control people thought automata were too hard, and a waste of time.

Other people, in computer science, dismissed it all as hopelessly trite - either it had all been done, or it wasn't worth doing anyway.

Finally, though, we snuck in, thanks to the word "optimal", which can be magic in control.

***

Moving on, here's the approach.

This machine is a labeled transition system.

The green transitions are called "uncontrollable", happening or not as the machine wishes.

While working, it could either complete the job, or else break down.

The red transitions are "controllable", meaning they can be enabled or disabled by an external controller, which we call a

"supervisor".

A supervisor could hold the machine at "Idle", or else at "Down" until a repair technician became available.

So the label alphabet is partitioned as shown.

The state with an incoming arrow is initial.

Any state with an outgoing arrow is called "marked".

When it reaches a marked state, the system outputs a beep, to announce that something useful has been accomplished, in this case another machine cycle.

Of course after beeping it can still keep moving around its graph, and eventually, perhaps, beep again.

***

Here is how entering and saving the DES MACH would be done in TCT.

***

The machine behaviors are the languages generated when the machine moves around its transition graph.

You have the set of all finite strings it can generate - this is the "closed behavior".

You also have the set of all strings that hit a marked state and so cause a beep - this is the "marked behavior".

The setup makes it easy to define a property of "weak liveness",

meaning no matter what happens you can always get to a marked state, and therefore accomplish something useful.

We call this "nonblocking".

Formally it means that you can always complete a string in the closed behavior to a string in the marked behavior.

Thus the closed behavior is exactly the so-called "prefix-closure" of the marked behavior.

***

We need to build complex systems from simple components.

This is done with the "synchronous product". It just forms a product of automata, synchronizing the component automata on their shared events.

It's more elegant, though, to work with the languages, just sets of strings, and use the so-called "natural projection" operator.

***

Now we can set up a control problem.

Here the plant is a "Transfer Line".

We have two machines and test unit, linked by buffers.

A tested workpiece is either exported, or, if found to be faulty, is sent back for re-working.

So we have a block diagram, as usual in control theory.

The arrows are nothing but synchronization on shared events, so the components link together naturally under synchronous product.

The "object to be controlled", or "plant", is the synchronous product of the three "active" elements.

Call this "TL".

TL may look innocent, but note the material feedback loop.

In feedback there lurks instability.

And in fact, TL can block - can grind to a halt - if you try to run it too fast.

***

As for specifications, let's suppose we just want to prevent the buffers from overflowing or underflowing.

We can form the overall specification in the same sort of way, as a synchronous product of partial specifications.

Call this "BUFFSPEC".

***

Now we pose some general questions.

By means of our control technology, and watching the plant behavior as it evolves in time, can we achieve "safety" - no buffer overflow - and also "liveness" - no blocking?

The trivial answer is "Yes" - by shutting the system down so it does nothing.

So we also ask for "maximal permissiveness" - the largest, or freest, behavior possible.

If there is a solution, can we synthesize it?

And if so, without unreasonable effort?

***

Here is the problem in formal terms.

It can be stated very simply and clearly in languages.

***

The formal solution depends on a key definition and a key result, both with their roots in standard control theory.

For our control technology, there is a language definition of "controllability".

It just says that if you're already within a controllable language, you'll be in no danger of skidding out of it on an uncontrollable event.

A controllable language is invariant under the uncontrollable flow, so to speak.

It turns out that controllability is exactly the property that allows you to implement a language as the behavior of the plant connected to some feedback controller.

And now the magic is, there is a maximally permissive way of doing this.

This optimal behavior is the "supremal controllable sublanguage".

All we have to do is compute it, set up the feedback loop, and we're done.

***

The picture in languages is nice and simple.

We have the language of all possible strings at the top, and the empty language, with no strings, at the bottom.

Working down, we have the two data languages, supplied by the user.

Then we do the optimization step, to produce this supremal controllable sublanguage.

Call it K_sup.

There may be less permissive solutions further down, but we don't need them.

Everything can be said in terms of languages.

The rest of it is just computation and, if you will, the "mechanical engineering" of automata, hopefully left as an exercise.

***

Here is the result.

It's a standard feedback loop, where the controller is exactly an automaton called SUPER, which simply represents the language

K_sup.

***

It turns out that SUPER is often much more complicated than necessary to do its job. This is because it incorporates both control constraints and plant constraints. Because the plant is already in the feedback loop, we don't need to incorporate its structure in SUPER. If we project out the structure of PLANT from SUPER we can often get a much simpler controller, say SIMSUP, which does exactly the same job.

***

That would be end of story.

Except that, we wanted the approach to apply to real systems.

Real systems get large - in terms of state sizes - very quickly.

You have the notorious "exponential state explosion".

***

We'll look at one computational approach to this issue.

It's an adaptation of methods well-known in the computing literature.

The key is to retain system product structure, so plant and spec are now both represented as vectors of state variables.

Then we look for SUPER as a predicate over product state sets as shown.

But instead of forming these products explicitly, and trying to store the predicate SUPER extensionally, we use algorithmic representations in the form of IDDs - integer decision diagrams.

***

To recall what these are, suppose you have a function, f, of some discrete variables.

You could always represent f as a tree, except that the tree would get large exponentially with the number of variables, defeating the purpose.

But it turns out that if you're clever in the way you order the variables, and are a bit lucky, then the tree will collapse into something much simpler, namely the IDD, which can be built from scratch.

The IDDs can be manipulated to mimic operations among the various functions.

How much effort that takes will depend on the number of nodes in the IDDs.

***

To see how effective this can be, let's look at this workcell.

We have 4 machines, 2 robots, and 2 part types, red and green, with assigned routes.

***

The safety specifications include the production sequences.

In this cell the robots are shared by the machines.

Sharing is often a recipe for deadlock.

For instance, Robot 1 can get confused and grind to a halt.

So on top of safety we have to impose liveness in the form of nonblocking.

***

Here are some numerical results.

K is a scaling parameter - the capacity of each machine and robot – ranging from 1 to 50.

State size goes up rapidly with K, roughly as K to the tenth.

Naive computation soon becomes unmanageable.

However, in the IDD computations, the node count increases much less quickly, about as K squared.

And even for K = 50, probably larger than this workcell would ever become in real life, the time and memory are both quite reasonable.

Storing the control function is not very demanding at all.

***

The key parameter is the number of IDD nodes.

And this only grows about linearly in the number of components.

The bottom line is that we have found that our theory is computable for many practical systems of industrial size.

***

A feedback implementation would be standard, IDDs now computing the control enablements and disablements from the system state.

***

As well as smart computation, an important way to manage control complexity is through suitable architecture.

One scheme is the classical, pyramidal, hierarchy, as explored, for instance, by Herbert Simon and Michael Mesarovich.

There's a manager at the top, and an operator or operators at the bottom.

As you go up the hierarchy, the levels increase in scope, which could be defined in different but somewhat equivalent ways.

It seems that the scope ratio, from one level to the next one higher, is often around 5-to-1.

So if you had 20,000 employees, you might expect to organize them in 7 levels.

***

Let's look at two adjacent levels.

We have the low-level world, say the real world, and the operators coping with it.

We also have the manager, who does his planning in terms of an abstract model.

Having made a plan, the manager hopes it will be executed around the hierarchical loop, via the chain: command, control, and report.

If in fact that happens, namely what's reported up actually agrees with what was planned, we say there's "hierarchical consistency".

In algebraic terms, the mapping "plan" factors through "report", in such a way that the bold-arrow diagram commutes.

As we know, this is by no means automatic: "The best-laid plans oft go astray."

***

We can formalize in terms of our language setup.

We have a high-level alphabet, say capital Tau, of a priori "significant events" for the manager.

Then "report" will be some mapping, say theta, from the low-level language L over SIGMA to the corresponding high-level language M over TAU.

And for "command" we can take the inverse-image mapping of theta.

***

Now we can express hierarchical consistency.

The idea is that our "supcon" theory should work at both levels, for

the manager as well as the operator.

The manager's "plan" is now the supC of his high-level specification.

The operator implements the supC of the manager's "command", theta-inverse.

The operator's result gets reported back up by theta.

"Consistency" is exactly this factorization of mappings.

It will usually require some rather careful refinement of theta.

***

This is how it looks for the toy Transfer Line.

We bring in some high-level events for the manager, for instance that a faulty workpiece is sent back.

***

Here's the state diagram of TL under low-level control as described earlier.

At each of the enlarged states, we have a state output over the alphabet Tau.

This state-output machine represents the reporter theta-map.

The simple abstraction it generates is the high-level model for the manager.

In this model red and green mean the same as before - our

standard control technology.

***

So the manager can compute his plan in the high-level model with assurance that his commands will be executed around the hierarchical loop.

***

Several extensions of the base model have been worked out, contributing greater flexibility.

The big item is a variety of architectures, especially combining hierarchy with decentralization.

[You can have forced events - essentially forcing by preemption.

In a timed extension, "forcing" amounts to preempting the tick of a clock.

You can have a more demanding form of liveness than just nonblocking, liveness in the form of "eventuality".

You can have "fairness", in the sense that the strong can be prevented from locking out the weak.

An interesting open question is how to compute in a more flexible modeling framework - where you have not only plain automata, but also Petri-like components consisting of integer or boolean state vectors.]

***

There's been a decent range of applications.

For instance the AIP system is an experimental, but realistic, industrial assembly process (in Grenoble, France).

Using our approach, controls for the AIP were worked out and implemented on the system by Bertil Brandin.

A recent commercial application in the telephone industry was reported by Michael Seidl, in a paper in Control Systems Technology.

***

Here's some recent work by a group in Eindhoven, on the design of a patient support system for an MRI scanner. The uncontrolled system has over 6 billion states, too big to compute with all at once. But by going to a highly modular architecture the designers were able to arrive at an "optimal" and nonblocking design quite successfully. We're pleased that they used our stuff.

***

By way of conclusions on the status of SCT, we can say...

The results are synthetic, and general within the class of finite-state transition models (which includes, for example, bounded Petri nets).

The results are correct by construction, and computable for 'large' systems approaching realistic industrial size.

The story already incorporates decentralized, hierarchical, and distributed architectures; extensions are under current research.

The mathematics is essentially rather simple, with teaching

material freely available on the Internet.

As challenges to future research, there are many things on the agenda.

For instance, there is the problem of understanding a complex controller.

It may be "correct", and even "optimal", but what is it "really" doing?

And can we formulate general laws of control architecture, whatever we might mean by that, exactly?
