MITOCW | watch?v=CTVFDb44ses

OLIVER DE WECK: So let me start. Session 3 is about system modeling languages. But before I start, I'd like to remind you that A1 is due today. The first assignment is due today. And I think neither [? Yuanna ?] or I or [INAUDIBLE], did you get a lot of questions about A1?

GUEST SPEAKER: No. I didn't get any questions.

OLIVER DE WECK: Yeah. So we didn't get many questions. So I interpret that as a positive, but I guess we'll find out. Everybody submitted.

Well, so we're hoping to have these graded in about a week, give you feedback. And we'll also post a master solution. And A2 is out right now. And the other good news is there is no new assignment today that's being-- the next A3 is going to go out next week.

Any questions about A1? Was there something that was particularly difficult or confusing? Or was it straightforward? Anybody want to comment on A1?

Wow. Sam. Do you want me-- push the button.

GUEST SPEAKER: No, our team thought it was fairly clear on what we had to do based on the lectures.

OLIVER DE WECK: OK.

GUEST SPEAKER: We didn't have any trouble.

OLIVER DE WECK: Good. All right. Well, let's keep going then.

So the V-Model is our roadmap for the class. We're starting to fill in the V. We're still in the upper left corner.

And today's lecture is actually a little different. It's sort of in the center of the V, system modeling languages, as a precondition or precursor to what we've been calling MBSE, Model-Based System Engineering. So what I'd like to cover today is why do we need-- or why have these system modeling languages emerged, particularly, what do we mean by ontology, semantics, and syntax?

And then we're going to give you a-- I would characterize this as a sampler of three different system modeling languages that have emerged really within the last 10, 15 years. The first one is called OPM, Object Process Methodology. The second one is called SySML, System Modeling Language. And then the third one is called Modelica. And then we'll sort of quickly wrap up with the question, you know, what does this mean now for system engineering today, and tomorrow in the future?

So I'd like to motivate this with a little exercise. So you remember Mr. Sticky from last time? You came up with some requirements. So it's kind of the simplest system I could think of here. So what we'd like to do is have you work in pairs again.

And the assignment here is to describe this system as clearly as you can, provide a description. So last time the assignment was right, a requirement, come up with some requirements that led to this design. But today I would like you to describe what the system is, how it functions, and so forth, as clearly as you can.

And I would like you to do this-- so hopefully you're on the Webex, logged into the Webex. I would like you to do this in teams of two. And as you're doing this, I don't know if you've noticed but on the Webex there's actually a Notepad feature.

Where is it? Tell me. Left? Right?

AUDIENCE: [INAUDIBLE]

OLIVER DE WECK: Oh, I see. So I can't share and use the Notepad at the same time.

AUDIENCE: [INAUDIBLE]

OLIVER DE WECK: Annotate.

AUDIENCE: [INAUDIBLE]

OLIVER DE WECK: Yeah, yeah.

AUDIENCE: [INAUDIBLE]

OLIVER DE WECK: Stop sharing. But then they can't see it.

AUDIENCE: [INAUDIBLE]

OLIVER DE WECK: Can they see this?

AUDIENCE: [INAUDIBLE]

OLIVER DE WECK: So you can write text. You can draw shapes. So the reason I want you to do this on the whiteboard is such that we can then go around and look at some examples.

So the assignment is take five minutes, turn to your partner and try to describe this system. And then we'll go around and look at some-- we'll sample people's descriptions. Go for it.

All right. So keep working on the assignment but do it locally on your computer, not on the whiteboard. And then we'll sort of discuss it and share it and maybe not use the whiteboard, because I didn't realize there's only one whiteboard that we all share.

I thought that you have individually the whiteboard and then you can sort of pass it on to different people. So if you do it locally on your machine, then we can share the screen so it'll work. So do it in do it in PowerPoint, or Word, or sketchpad, or anything you want. Sorry about that.

All right. So let's do this. We're going to sort of go back and forth between here and EPFL.

Let's start maybe over here with Narek. Are you ready? So I'm going to give you the ball, and then you can sort of explain how you guys describe the system.

NAREK SHOUGARIAN: So what we decided to do was identify the primary function of Mr. Sticky. So the primary function is to trap the fly, we thought. This is enabled by a couple of other functions that are sort of at a lower layer of abstraction. It's attracting the fly, immobilizing the fly, transporting the objects to where you need to immobilize the fly, and deploying Mr. Sticky.

And we mapped this to the physical forms that enable the function. So the canister, the physical form of the canister is helping with the transporting function. The sticky tape is helping with the immobilizing function. The scented material, we thought, would be helping for the attracting function. And the hook maybe on top that you use to hang it would help with the deploying.

OLIVER DE WECK: OK. Good. So nice function form separation. And you used primarily text, human language to describe it.

So let's see, at EPFL, who would like to share? And we'll give you the ball.

AUDIENCE: OK. We can try maybe.

OLIVER DE WECK: Who is speaking?

AUDIENCE: Maxim.

OLIVER DE WECK: Maxim, OK. Can we give the ball to Maxim?

AUDIENCE: Yes. Do you see something?

OLIVER DE WECK: Yeah, its good.

AUDIENCE: Oh, perfect. So we draw like the same system when deployed and undeployed. So we begin with a container containing basically the sticky setup rolled. Then when unrolled, we have the container that should be linked to the sticky setup, the [INAUDIBLE] whatever. And then we have like an external input from the insects that were, that come to-- we it sticked onto the--

OLIVER DE WECK: Yeah, go ahead

AUDIENCE: OK. So that's all.

OLIVER DE WECK: So I'll note here that you guys used graphics. You used some kind of graphical language. And at the highest level, you have-- it's like a state diagram, rolled, unrolled. So you showed the system in two different states. Very nice.

Somebody else here on the MIT side, and then we'll go back one more time. Who would like to share here? Lucille? OK. So let's--

AUDIENCE: So we did a diagram showing the use of Mr. Sticky. So we have a user because Mr. Sticky has to be rolled and unrolled-- well, unrolled. The user installs or disposes of Mr. Sticky.

The flies are attracted to-- or Mr. Sticky attracts the flies, and the flies stick to Mr. Sticky. And then we did compose Mr. Sticky into the different components that are below at a lower level. And yeah, so it's basically at a higher level of use diagram.

OLIVER DE WECK: OK. So here we have, again, a graphical description. The states are sort of implied. But you're focusing on decomposition, the elements. Very nice.

Anybody else at EPFL? Did anybody just write a paragraph of text, or more of, like, sentences?

AUDIENCE: [INAUDIBLE]

OLIVER DE WECK: At EPFL, who wants to share?

AUDIENCE: Chris here.

OLIVER DE WECK: Chris, OK.

AUDIENCE: So we didn't write the text. The text seems a bit heavy to convey a description in efficient terms. What we prefer to do is decompose in elements and for each element give some properties.

OLIVER DE WECK: OK. And did you do this in the form of a list, or in the form of a table? Or how did you actually describe it?

AUDIENCE: So wait. I'm trying to share the screen.

OLIVER DE WECK: OK.

AUDIENCE: All right. So we just worked on the first half here. We have the band. And we give here properties.

Those that have to be made of paper or soft material has to be 1 to 1.5 meters long, 3 to 5 centimeters large. It needs to have a coating which itself is a sticky material, and be centered in order to attract flies. It needs to have visible color. And then, well, the other part here, which

I'm highlighting, would be related to the packaging, so a self-sealing linear container with a single-use opening, including a hanger. We'd possibly give branding on the packaging and, well, the non-toxic material would refer to the sticky material.

OLIVER DE WECK: Good, good. Thank you very much. This is great.

So what you showed, that's more like a list. And I would describe-- this looks like what I would call a bill of materials. It's essentially a list of the primary elements of form of the system. But there are some attributes that are associated.

So this is a list format in the form of a bill of materials with attributes attached. So thank you very much. That's great.

So I'm going to share here again. And you're probably wondering what the heck? Why did we do this? Why did we do this exercise? And that's the point I want to make next.

So here's a very simple system. And we have four examples of descriptions that were quite different. And of course, if you had more time, they'd become more complete. But they would be really different.

So this is fundamentally the issue that we've been facing in systems engineering for a long time. The means for describing our artifacts, whether it's something as simple as a Mr. Sticky, or an airplane, or a spacecraft, or a medical device, or even a service, how would we describe it? Well, first of all, natural language, the human natural language.

And as we know, the human natural language is very rich. There's very different ways in which we can express, essentially, the same facts, the same things. That's a wonderful thing if you're a poet or a writer. But it makes system engineering challenging, because it gets confusing when we're describing the same thing in very different ways.

Or graphical, so we saw some boxes, and we saw some great examples, sketches, drawings. So fundamentally, the way we describe systems-- and this gets to the left half and right half of the brain-- is using language, words, sentences, lists, or graphical. Those are the two fundamental ways of describing systems.

And then we put all these descriptions together in what we've been calling documents. We aggregate this in documents. So examples of documents would be a requirements document.

That's what, essentially, you're doing in Assignment 2. Or a drawing package, even if it's in CAD, it's still essentially a document.

So typically in system engineering all of this gets assembled into what we call a TDP, Technical Data Package. And fundamentally, when you're designing a new system, you're producing a technical data package that has software drawings, descriptions. And that's the deliverable from the design process, is this TDP, Technical Data Package. And from that, you should then be able to build and operate the system with as few errors, mistakes, misunderstandings as possible. And fundamentally, as our systems have been getting more and more complex-- we're now talking about the systems that need the three, four, five layers of decomposition-- it's very easy to have errors, omissions, different interpretations of this information.

So that's fundamentally-- but there are advantages. I don't want to say it's categorically bad to use natural language and graphics. They're definitely an advantage, familiarity to the creator of the description.

So it's easy. It's comfortable. It feels familiar. And also it's not confining, so you can be quite creative by creating descriptions in this way.

But the list of disadvantages is quite long for allowing an arbitrary description, room for ambiguous interpretations and errors. It's quite difficult to update. So if you make a change in one description, that change will not automatically propagate to the other descriptions. Handing off these descriptions from one lifecycle phase to another, there's discontinuities in these hand-offs.

Uneven level of abstraction, so what I mean by that is you may describe one part of the system, and very detailed. So the last example we saw with the list, with the bill of materials, there was quite a bit of detail there on the scent and the attributes of the tape. But one of the other-- at least a couple of the other descriptions didn't have that level of detail when it came to the tape. So the level of abstraction could be quite different in the different ways to describe it.

And then for a complex system, you can imagine that the amount of volume of information grows a lot. And so you can walk into any program manager and systems engineers office and see bookshelves full of binders, dozens and dozens of binders with documents, thousands and thousands of pages. And many of them are never read. That's the big issue. So that's been the kind of way in which we've been doing system engineering traditionally. So the idea here is that in order to mitigate-- yes, Justice.

AUDIENCE: [INAUDIBLE]

OLIVER DE WECK: Phase A, conceptual design. Phase B, preliminary and detail design. Phase C is testing and launch. Phase E is operation.

So usually not the same people do conceptual design, preliminary design testing. So all the technical data package, these artifacts have to be transferred and handed off to new people who then work on the next phase. That's what I mean by hand-offs.

And, yeah, so the idea is in order to mitigate some of these disadvantages of natural language and graphical description, there has been, and this has been recognized for a long time, a need to be more precise, perhaps more confining, but to create languages that allow us to describe systems much more precisely. And so I'll mention a couple of the past efforts. And you can read about these.

Each of these has-- so I'll mention bond graphs first, 1960. This was actually invented here at MIT by a professor in mechanical engineering. His name was Henry Paynter. Professor Paynter created bond graphs.

You can think of bond graphs as block diagrams where different blocks have ports or interfaces where information, material, energy flows in and out. And you can compose a system out of these blocks. These bond graphs are essentially-- and Narik will talk about Modelica, which is sort of a modern version of bond graphs. It has other features, too. But this has been sort of one attempt.

Another one that's very well-known is IDEF, I-D-E-F, about 20 years later. This was created by the Air Force, the US Air Force. And this is essentially a description of systems that's very functionally oriented. So it shows you what functions are involved.

And we saw one of the descriptions was very functionally oriented, and how the functions of the system relate to each other. But generally these system languages have not fully been deployed. Some organizations use them, others don't use them.

And the main reason for that is twofold. First of all, some of these languages were incomplete.

They would focus only on one aspect, like functions or the block nature of the system, the block diagrams. And a lot of them were not executable. So they would be graphical, but you couldn't actually simulate and actually check whether that description was complete or accurate.

So since then-- and the other thing, of course, important is domain agnostic. So what I mean by this is that the system modeling language should be applicable for any kind of cyber physical system. Again, if you're designing a spacecraft, an aircraft, medical device, any kind of product, the language shouldn't have to be adapted. The language sort of covers all these applications. That's the idea.

So whatever language it is that you're using or developing, it has to have these three things. Any language has these three things. So the first is ontology.

And I reference here the Wikipedia articles on these things. Some of my colleagues-- in academia it's a big debate is Wikipedia a legitimate source of information or is it not? My position on this is that it is. I think Wikipedia is definitely not perfect, depending on what topic you're looking at, but it's a sort of self-correcting system.

So I actually go to Wikipedia, and then there's references. And you can dive deeper. So I give you the Wikipedia links here for these three things.

First, ontology. So ontology is a very fancy word. What ontology actually is-- Mark, why don't you come up here? You're an instructor today. Ontology, fundamentally, is describing the entities that are allowed to exist in the language, subjects, nouns, adverbs, what are the objects, the entities, that can exist.

So it's a kind of very abstract thing. But it's essentially the library of words and objects that are allowed to exist in that language and then how these entities can be grouped perhaps in a hierarchy and subdivided. So it essentially constrains the universe of things you can describe in that language. The shorter, the smaller your ontology is, the more confined the language.

Semantics. Semantics is basically a branch of science of philosophy, which is fundamentally assigning meaning to those objects that are described in the ontology. And so the way that we say this is that it's the relationship between the signifiers, so the signifiers are words, letters, symbols, graphical symbols.

So how do we describe a resistor, for example, in electrical engineering? A squiggly line. It's the zigzag line.

Well, if you don't know electrical engineering, you just see a squiggly line. It's meaningless to you. But if you know that semantically that means that's a resistor, that's the symbol for resistor, that's what we mean by semantics.

And then the third is syntax. What is syntax? It's the set of rules, the set of principles and processes by which the objects or the entities in the ontology can be combined to build up higher level information, like sentences, paragraphs, and so forth.

And so that's essentially the construction rules for the language. So every language has these three things. So any questions about this before we move into our first language?

So we're going to give three examples of system modeling languages, and you'll see the similarities and differences. But as you see these languages, keep in mind they all have ontology, semantics, and syntax. Any questions about that?

So OPM. Let me describe to you OPM. This is one of the younger languages.

And so OPM stands for Object Process Methodology. And it was created by Professor Dov Dori at Technion, a colleague of mine. Dov is essentially a computer scientist by training. And you'll see the heritage here of OPM.

And the big news here about OPM is OPM is not that well-known yet. If you ask around, not too many people know OPM. But I predict that in the next decade that will change very quickly. And one of the reasons is that OPM was just now adopted as an ISO standard. And if you know ISO, the International Standards Organization, they're located in Geneva, it's a big deal to become an ISO standard. It took like five years, the whole process, with committees and reviews.

And so the ISO standard is actually-- OPM was adopted as an ISO standard as a means to describe and develop other standards. So it's kind of a metalanguage because, as you can imagine, when you read different standards which, by the way, have a lot of influence, they're also written in natural language and graphics and lots of inconsistencies. So the idea is that future ISO standards should be written using OPM such that they're clear and consistent, and so forth.

So the history here is that if we go back further, UML, which is Unified Modeling Language which I'm not going to talk about today, is a software-- this is a language that was developed primarily for software engineering to consistently describe use cases, to consistently describe activity and flows in software, the structure of software. But it's really software centric. So from UML 2.0, we then sort of branched off into SysML, which Mark Chodas, who just joined us will talk about, and then OPM.

So these are sort of derivatives of UML. And there's a book, it's not one of the mandatory books for this class, OPM 2002. If you're really interested in OPM, I recommend you invest in that book. It's really very well written.

So let me give you an example of how OPM can be used. So like we said, typical product representations are sketches, engineering drawings, or UML diagrams and software, but the need for a unified representation. And fundamentally, we have functions and then we have objects, form and function, in systems.

And so what we would like to do, and the premise of OPM, is that we can show everything in one diagram type, so the functions, the functional attributes, the objects, and there's different types of objects, operands, system components, consumables, the attributes of those objects, and then the links. And I'll show you the different types of links between these that exist in OPM. So it's a generic modeling language. And it has been successfully applied to system architecting of complex products in different organizations.

I'm going to try to give you a pretty simple example here, which is a refrigerator. So we're going to look at a small kind of household-level refrigerator through the eyes of OPM.

All right. So here's the basic ontology of OPM. It's very, very simple. And that's the idea, is to have as few objects, as few entities as possible in the language to keep it simple.

So the first one is the idea of an object. What is an object? Objects are drawn as these rectangles. Objects are defined as entities that have the potential for stable, unconditional existence for some positive duration of time.

And objects have states within them. So what would be an example of an object that we've talked about today? Go ahead.

AUDIENCE: Maybe the sticky tape.

OLIVIER DE Yeah, so the sticky tape itself. That's an object. It exists unconditionally. It's there.

WECK:

And what's important is it could be a physical object. So it has a physical existence. But it could also be a sort of informational object. So for example, if you have an idea or a vision, that's an object, too. It's not physical in that sense, but it does exist as an informational object.

What are states? Let's see at EPFL. What would be an example of a state that's associated with an object? Yeah, that's OK. Can somebody give an example of a state?

AUDIENCE: Yeah. Rolled and unrolled for the sticky tape.

OLIVIER DE WECK: Rolled and unrolled. Exactly. Or furled and unfurled.

So that's kind of a binary state that could be halfway unrolled, or the sticky tape is full of flies, or it's kind of empty. Those would be describing the object in terms of what state it is in. Exactly. So the form is then the sum of all these objects. So that's one building block.

And then processes are the other. So what are processes? Oh, is there another example? Yeah, go ahead.

AUDIENCE: May I ask a question?

OLIVIER DE WECK: Is this [INAUDIBLE]?

AUDIENCE: Yeah, it's [INAUDIBLE].

OLIVIER DE WECK: Go ahead.

AUDIENCE: What do you mean by positive duration of time?

OLIVIER DE WECK: Well, meaning that the object could be created. It didn't exist before. It's created, and it exists. And then it could be destroyed again. It could disappear, or it could be consumed.

But it means that the object needs to exist for a non-zero period of time in order for it to be called an object, so objects in the world that can be described with OPM, fundamentally, objects can be created. Objects can be modified, particularly their states can be modified. And they can be destroyed or consumed.

That's basically it. That's a complete set. Does that make sense?

So processes are-- what are processes? Processes are really fundamentally different from objects. Processes are shown as these ellipses. And they're the patterns of transformation applied to one or more objects. And processes change states.

So processes, unlike objects, cannot exist on their own. Processes only make sense if they're associated with at least one object. So processes are essentially-- the functions that we develop in systems are processes that transform or create, destroy, or transform objects. So function emerges from one or more processes.

And then we have different links between objects and processes. I'll show you two examples here. One is the arrow with a pointy head. That could be a consumption or production type link, or a link with this little lollipop symbol. This is known as an instrument link.

And so the difference there is that if an object is linked to a process using an arrow, it means that something's happening to that object. It's being created, or destroyed, or modified. If an object is linked to a process using the lollipop symbol, the instrument link, it means that in order for that process to happen this object is needed, it's an instrument. But the object itself, the instrument, does not get modified in any way by the process. But the process couldn't happen if that object didn't exist. Do you see the difference?

And so one of the really, I think, important things about OPM, but any of the languages, is that every box, every arrow, every link has precise semantics. And usually when we kind of doodle, when we just think about-- we put arrows and links between boxes, we often don't really deeply think when I put a link in here, what does that link actually mean? What does that link imply?

So when you do system modeling using these languages, you become much, much more precise. Yes, please. And would you push the mic button? Yeah. Mark, go ahead.

AUDIENCE: Can an object be a process or a process be an object?

OLIVIER DE WECK: No.

AUDIENCE: So what about if you have-- I guess I'm thinking in terms of if there's some process for doing some procedure and like you're assembling a satellite or whatnot. You need to modify that process. So how is that sort of thing represented in OPM?

OLIVIER DE WECK: The process modifies objects, but processes cannot modify other processes because processes are fundamentally, in OPM, acting upon the objects. Now, processes can invoke each other. So if there's a sequence of processes-- you have to do this assembly step before this other assembly step-- you can have what's called an invocation link. But that's a logical dependency between processes. But fundamentally, the processes act through the objects in OPM.

AUDIENCE: OK.

OLIVIER DE WECK: So let's go into some more detail. So at a high level when you look at the economy, products-- yes, go ahead.

AUDIENCE: I have a question. Why create another language and not just stick with UML?

OLIVIER DE WECK: So good question. So we'll talk about SysML, which is very similar to UML. It's sort of generalized for cyber-physical systems, not just software.

So the reason that OPM was created, because UML was found to be somewhat too confining. This is more general and also the idea of a unified representation, one type of diagram and description for any application. So it's basically a kind of a more general version of UML.

But the other really important thing about OPM is that objects and processes are sort of-- the processes are often in object-oriented thinking. Processes are embedded inside objects. And in OPM, the processes have been emancipated to stand at the same level as objects. Those are the main differences.

So let me go in a little bit more detail. So if you think about the economy in general, goods and services, goods are objects, and services are processes. So if you buy a new iPad, or a new pencil, or whatever it is, you're actually buying an object.

You're purchasing an object. That's obvious. But why are you purchasing that object?

So let's say you're buying a new tablet. You're buying the tablet. But why are you buying the tablet? Sam, go ahead.

AUDIENCE: You're buying the tablet to perform an action for a process on something else, to do something.

OLIVIER DE WECK: Right. So what do tablets do? I mean, not stone tablets, but modern tablets.

AUDIENCE: They allow you to work with software, communicate.

OLIVIER DE WECK: Yeah, so they're information processing. They're information-processing devices. And there's an argument with tablets are great for consuming information. They're maybe not as good for generating new information. So fundamentally, you're purchasing the tablet, which is an object, in order to be able to do information processing and information consumption. So the process is then implicit.

What's an example-- if you purchase a service, what would be an example of a service? What would be an example of a service? Let's see at EPFL. What would be an example of a service that you could purchase?

AUDIENCE: Going to the dentist.

OLIVIER DE WECK: Going to the dentist. Yeah, one of our favorite things to do. Have you been there recently?

AUDIENCE: Yeah, one month ago.

OLIVIER DE WECK: So I don't want to be-- I don't want to violate your privacy, but can you share with us what happened at the dentist?

AUDIENCE: It's the yearly checkup. You have to check that there's no hidden-- I don't know how you call that in English, carries?

OLIVIER DE WECK: Yeah, cavities. Yeah.

AUDIENCE: Yeah, cavities. Check that wisdom tooth don't mess up what you've been working on tirelessly when you were younger with braces. And [INAUDIBLE] get checked.

OLIVIER DE WECK: Very good. So going to the dentist. The dentist provides a service which is either checking your teeth or filling cavities, which is a process. And all the objects, the chair on which you sit, the instruments-- I guess we still use gold sometimes in some places-- those are objects that are used in the performance of the service.

You see the relationship? So objects and processes always come in pairs. Thank you for that example.

So let me talk about the links in OPM briefly. So there are two types. There's the structural links, which link objects to objects. And we typically use arrows, you know, is related to or we can tag these as well.

So for example, something powers something else. This is known as a tagged link. It suppresses the processes.

And then there's these triangles that are-- essentially, there's a kind of hierarchy implied there and slightly different meanings. So the solid triangle means decomposition. So the higher level object is composed of lower-level objects. So that's, Mark, you mentioned assembly, you know, are you creating the bus of the spacecraft and it has a whole bunch of stuff in it? Well, you would use this filled in triangle to show that decomposition.

The second example is the characterization link. So this is essentially relating an attribute to its kind of master object. Specialization and generalization is the empty triangle.

And then this funny symbol here is instantiation. So essentially, you have a general object. And then you can instantiate that.

So I have two children. I have two children, which is general. And there's two instantiations of them.

One of them is called Gabrielle. And one of them is called Christian. And they're actual people. So that's the idea of instantiation.

Processes. Processes are these patterns of transformation. They're tricky. Processes are trickier to understand than the objects because we cannot hold or touch a process. It's fleeting.

And the creating change or destruction of objects is what processes do. They rely on at least one object in what we call the preprocess set. A process transforms at least one object.

And the time is implied. So processes take along a timeline. And in terms of the description, in English we use the so-called Gerund form. So all the processes, there's some examples on the right side, use the "ing" form of a verb.

So we can then put these together, objects and processes. So here's an example of a machine. This happens to be like a printer or copy machine. It has a main switch. The main switch has an attribute called Main Switch State, which can be on or off.

The process of switching transforms, in this case, the Main Switch State from on to off. Or we could go the other way. And in order for this to happen, we have here-- this is actually slightly different than the instrument link. This is a filled in lollipop, which is known as an agent link. So the operator is an active agent to carry out the switching process which changes the main switch state from on to off or off to on, and the Main Switch State is an attribute of the main switch.

So transporting. This is another example. Transporting changes the state of a person from being here, Location A, to being there, Location B.

So there are seven-- huh, coincidence, seven, seven-- object process links in OPM. So P changes-- the process changes the object, say from State A to B. That's the example we just looked at.

You can actually hide that. If you're really not interested in all the states and details, you hide the states. You don't want to see them.

And then you can replace that with what's called the affectee link, which is this two-sided arrow. And all you know is that this process is affecting that object. And it's a two-sided arrow. But you don't know exactly how but you know it's affected.

A resultee link-- so this is an arrow pointing from the process to the object-- means that the process of transporting produces emissions that weren't there before. So that's a resultee link. But the process of transporting requires or consumes energy. So the arrow is pointing from energy into the transporting process because it's being consumed.

I did mention the agent link. So there's an operator of a vehicle. And when we talk about autonomously driving vehicles, a big topic right now, actually, it was cool. At EPA this summer, there's the autonomous shuttle on the campus, the electric shuttle. Did anybody take that? Did you guys try that shuttle this summer?

AUDIENCE: [INAUDIBLE]

OLIVIER DE WECK: Yeah? Did you like it?

AUDIENCE: No, I didn't. But a friend is working in this kind of shuttle, like sitting for hours waiting for people [INAUDIBLE].

OLIVIER DE WECK: So fundamentally, I mean, if you want to think of this in OPM language, a driverless vehicle is basically eliminating this, no longer needing an operator with an agent link and replacing this with a piece of software, which would be an instrument link. So the instrument, the transporting process requires a vehicle. And then we have what's known as a conditional link. So this process can only occur if this object is in that particular state.

So this example here, obviously, ignores the existence of credit cards. So you can do the purchasing. The process of purchasing is conditional upon the state of the object money being in a state of enough for doing the purchase.

So here's an example of a little bit more complicated. This is a Level 0 OPM diagram of a car, of a vehicle. So you can see in the upper right is sort of a sketch of a vehicle. And it has these different attributes.

ED is engine displacement, height, ground clearance, overall length, wheelbase. There's a trailer here with a towing capacity. So the way the way you would interpret this is that we have a transporting process.

That's our master, sort of the highest level process. And it changes the attribute location for driver passengers and cargo from A to B. And that's, fundamentally, where the value is for the owner of the vehicle.

And then we can zoom in to the transporting process and look at subprocesses, towing, propelling, and housing. And if you think about what a vehicle does at the highest level, it protects you, it houses you, and it propels you. And then you can break those into more detail.

And then on the left side here we have, essentially, the elements of form. So the automobile, which is an instrument of the transporting process, can be decomposed into its major subsystems, powertrain, chassis, body, wheels. And each of those are characterized-- you see those attribute links-- characterized by things like fuel capacity, engine displacement.

This is the design domain we talked about last time. Ground clearance. And so those are the design variables. Those are the parts and assemblies.

And then on the right side, the processes, the internal processes can also be characterized by performance or functional attributes like towing capacity, fuel economy, acceleration. PV stands for Passenger Volume and cargo volume. And those are things when you're comparing different vehicles to purchase, those are the things you would compare vehicles against. So they're the internal functions and then the functional attributes.

And then up here there is the fuel and emissions and safety-related issues, which that's often where the governments intervene and then regulate. And this is sort of a highest level OPM of a vehicle. And then if you want to see more detail, you would start drilling down into these. And you'd have multiple levels of these, like a hierarchy of these diagrams. Yeah.

AUDIENCE: So here, what is the use of the-- or the meaning of the open arrows? And it looks like there's a couple of different arrows here than what we had in the other diagram.

OLIVIER DE WECK: Are you talking about these guys?

AUDIENCE: No.

OLIVIER DE WECK: Oh, these here. Yeah.

AUDIENCE: Yeah.

OLIVIER DE WECK: So it's just a visual represent-- there's no distinction on the arrows whether they're filled in or empty. That's just a kind of graphical thing. Yeah. So one of the-- yes. Veronica, do you want to push the--

AUDIENCE: How would you represent a process that creates kind of a temporary state? So if you're saying this is acted on an object, and this changes the form of the object but the object will ultimately return to its original state kind of absent of a reversing process, if it's a natural tendency for the object to return, how would you represent that change? Would you need to break it down as kind of a subprocess within the object?

OLIVIER DE Right. So I mean, and sometimes there's multiple, there's non-uniqueness in sort of

WECK: representing the same thing. But there's one process that brings you to the temporary state.

AUDIENCE: OK.

OLIVIER DE WECK: And then there would be a restoring process that restores you back to the original state.

AUDIENCE: Does the process have to be a separate plan within the system? Because there are certain objects that have a tendency-- I'm thinking primarily of kind of chemical states where reactions would happen naturally. It's kind of a specific thing, but I was thinking about how you might model different systems. And I was thinking about the engine of the car, just kind of how things might naturally return. So do you have to describe the process explicitly if it's not something that's inherently designed in, if it's kind of a-- if it will happen anyway.

OLIVIER DE WECK: I think, I want to say you have to explicitly define that.

AUDIENCE: OK.

OLIVIER DE WECK: So if it's a man-made process, so to speak, then that's a process you want to happen. And then if the restoring it back to some other state is a natural process, well, it exists. So it will restore the system to a prior state. That process would also have to be modeled.

AUDIENCE: Is there a distinction between how you would indicate a man-made process or a natural process?

OLIVIER DE WECK: Not fundamentally.

AUDIENCE: OK.

OLIVIER DE WECK: And in fact, OPM has been applied to modeling how a cell functions. So there's been some pretty recent work on-- cells are incredibly-- the biological engineering is just really complex. So there's some really recent work on describing even the RNA and cell division, using very much this language.

AUDIENCE: OK.

OLIVIER DE So it doesn't matter whether it's an artificial process or a natural process.

WECK:

AUDIENCE: Thank you.

OLIVIER DE WECK: Let me go a couple more minutes, and then we'll take a short break. And then we'll talk about SysML and Modelica. So the key thing in OPM is there's only one type of diagram. And there's also natural language that gets auto-generated.

And I'll show you this very quickly in the tool. So as you can imagine, as you're working on real systems, these diagrams, if you'd showed them on one sort of level, you'd have thousands of objects and links. It would be a mess.

So how does OPM handle complexity? There's three fundamental mechanisms. One is known as folding and unfolding. What does that mean? It's basically related to the structure. So folding/unfolding means that higher level objects, you can show the decomposition of the objects or you can hide it. That's known as folding and unfolding.

Then the second one is in-zooming or out-zooming. And so here's an example of a process and an instrument and an affectee that's affected by the process. And I want to know, what are the subprocesses in that process?

So you can zoom into this process, and it will expose the subprocesses that are happening inside. That's known as in-zooming. And then going back the other way is called out-zooming. And then the third one I've already mentioned, which is that states can be expressed or suppressed or hidden depending on your interest and what states of the system you want to look at.

So here's the sort of Level 0 OPM of our refrigerator. I said that was kind of our case study. So how does the refrigerator work at the sort of-- Level 0, that's what the stakeholder, what the customer sees. Don't care about the details of what's happening in the refrigerator.

So we have food-- and we'll get back to this I think next week in the kind of creativity concept generation. Why do we have refrigerators fundamentally? If you've heard this before, you keep quiet.

Maybe EPFL. Why do we have refrigerators? Any ideas? Go ahead.

AUDIENCE: Keep food cold.

OLIVIER DE WECK: Yeah, well, if you're a beer drinker, you want cold beer. But if you really think about it deeply, that's not really the primary reason. The primary reason is this state change, their shelf life. So the primary reason why you have refrigerators is to extend the shelf life of the food.

So speaking as a systems architect, system engineer, a refrigerator is a food spoilage rate reduction device. You see that? So the attribute of the food is the shelf life, and we're going to extend the shelf life of the food.

If you think about it sort of architecturally, that's why we have refrigerators. But I agree with you on the cold beer. We all agree we want cold beer, not warm beer. So you're right. You're right, too.

So the refrigerator essentially is an instrument of extending the food shelf life. So the food is the operand. The food is the operand. The extending of shelf life is what we call the primary value delivering process. That's where the value is.

The refrigerator itself is the product system. And then the operator sets the thermostat setting at which temperature the refrigerator should be. And then here we have the primary operating process, which is what allows us to keep the temperature of the food at that level.

And in order to do this, we consume electrical power. We produce waste heat. And we also require, or we convect that waste heat to the exterior air at a certain temperature.

How well do refrigerators work in a vacuum chamber? They don't. They don't. There's no way to-- well, I guess you could radiate the heat a little bit. But they're not going to work very well.

You're not going to have conduction because you're sort of in the middle of the vacuum chamber. You're not going to have convection. So you only have radiation. And that's not going to work very well.

So the exterior air is important for the refrigerator to work. So then you say, well, OK. That's fine. I buy that.

But now I want to really know, how does it really work? So you say operating. The refrigerator is operating.

But I want to do in-zooming and understand, how is it operating? So what's the key to refrigeration? What's the magic word there, or two magic words?

AUDIENCE: [INAUDIBLE]

OLIVIER DE WECK: Yeah, that's part of it. That's just a sliver of it. Heat exchange is part of it.

So the magic word is Carnot cycle. So here's a little graphic that sort of gets into it. So the Carnot cycle is actually a thermodynamic concept where you're compressing essentially a refrigerant. A coolant is being compressed, absorbs the heat from the inside and then expands and condenses and radiates that or convects that heat to the outside.

So here's a-- I don't know if you remember your thermodynamics. This is a classic PV diagram. You've got the four legs of the Carnot cycle. And actually, what's really nice here-- so we're going through this counter-clockwise. What's really nice about it is that every leg of the Carnot cycle is one of our processes.

So compressing is this leg here from D to B. Condensing is from B to A. Expanding from A to E. And then evaporation happens from E to D.

So the Carnot cycle can be decomposed into four subprocesses. These are the internal processes in the system that are governed by physics. So if we take that operating process that we looked at before, we can actually zoom in and see the subprocesses emerging. And so in cooling we have those four expanding, evaporating, compressing, condensing. But I'm adding the absorbing process, which is that the heat then needs to be absorbed by the exterior air.

We have to power the device. You can decompose that into grounding, protecting, supplying. Regulation, keeping it at the set point, you can decompose that process into sensing, switching, and setting the set point. And then we have supporting, which is we need to be opening and closing the refrigerator, retaining it, and then connecting all the pieces.

So at Level -1, we had one process at Level 0, which was operating. The refrigerator was operating. And then as we zoom into Level -1, four processes appear, powering, regulating, cooling, and supporting.

And then at Level -2, we have 15 subprocesses. So this is sort of a view at Level -1, our four subprocesses, cooling, powering, regulating, supporting. And then we can zoom in more.

So here's the general idea, and we've looked at many systems over the years, that most cyber-physical systems-- or it says optomechanical here, but I really mean it more generally-- have this kind of OPM structure. On the right side, we have the output that the customer, the stakeholder cares about, the operand. We have a set of specialized processes, and these can be often organized in a cascade. And then we have supporting processes, like powering, connecting, controlling, that provide support for the specialized processes. Most systems that we've seen have this generic architecture.

How do you generate an OPM? Fundamentally, you can do a top-down. So you start with your stakeholders. That's what we did in the first lecture.

Where's the value? You start thinking about requirements, what functions, how well the functions should be performed. And you sort of go down.

Or if you already have a system, you can actually reverse engineer that system and from bottom up, like we started doing for the Mr. Sticky, and that's fundamentally reverse engineering. So just for time, I'm going to skip this demo. But what I will do is I will post a video. I'm going to make a little video with the OPCAT demo and post that to Stellar so you can sort of watch that.

So this is one of the-- it's still not super mature, but it's a Java-based program called OPCAT that allows you to generate object process diagrams in a computer-supported environment and store them in an XML format, and so forth. It allows you to create a hierarchy. And the other thing that's very cool, it autogenerates text.

So the text is called OPL, Object Process Language. And right now you can go from the graphics to the text, but you can't go the other way. So they're complete sentences. It's not like an exciting novel when you read it. But it is semantically precise.

So we're going to switch to SysML. We're going to take a very short break. Are there any questions about OPM? In the system architecture class, we spend like five, six lectures on OPM and you get to do detailed exercises.

We kind of don't have time for this in this class. But hopefully you've seen what it is. And if I've whetted your appetite for OPM, then the goal has been met.

Any questions about OPM? Is it pretty clear? All right. So let's take-- yes.

AUDIENCE: Compared to Modelica, because I've seen one of the links that OPM is just for describing the system, though. It's not for making calculations or simulations.

OLIVIER DE WECK: That's correct. The latest versions of OPM you can do like a logical simulation. So you can say, OK, this process enables and does this state, so it's kind of a discrete logical. But usually it's not used for any mathematical calculations. The purpose of OPM is really to support conceptual design, early conceptual design. That's correct.

Mark. So just a couple of words about Mark. He's a doctoral student right now in the Space Systems Lab. He's been working a lot on an instrument called REXIS. I guess you're the chief system engineer for REXIS.

MARK CHODAS: Yeah, yeah.

OLIVIER DE WECK: And that was also the topic of his master's thesis. So Mark really knows what he's talking about. He knows SysML quite well. And thanks for doing this.

MARK CHODAS: So let me start by giving kind of a high-level overview of what SysML is and what it aims to do. So it's similar to OPM, but there are a couple important differences. SysML, as Olly said, it kind of is an extension or inherits a lot from UML. And its aim is to really provide a language that enables you to capture all the different aspects of information about a system in one place.

And this concept of Single Source Of Truth is something that I'll kind of try and emphasize during my presentation. The idea is if all your information is in this one model, then communication is easy. There's no ambiguity between versions. Everyone knows where to go to get the most up-to-date and correct piece of information. So that's one of the emphases of SysML.

SysML is a graphical language, similar to OPM. It's defined in terms of diagram types that I'll go into in a second. It has more than just one diagram type, as compared to OPM.

But basically it aims to do things like capture functional behavioral models, capture performance models, capture the structural topology of your system, the parts of your system and how they're all interconnected, and any other engineering analysis model is one of the big emphases with SysML, is integration with external analysis tools. So if you have a thermal tool, a structural tool, an electronics tool, or something like that, integrating this informational, descriptional model with that analysis model and enabling, making it easy to transfer information from your description model to your analysis model, do an analysis, and then incorporate those results back into your descriptive model is one of the things that SysML really is all about.

Then another thing, another difference compared to SysML from OPM is it incorporates requirements pretty explicitly. And that's one of the other areas that people are really interested in is if you have good modeling of requirements, what sort of information can you glean about your system that you couldn't otherwise?

How do I advance the slide? Oh, there we go. So as I said, SysML is composed of diagrams. I'll go into in a second kind of what each diagram, what all the diagram types are and what kind of their intent is.

But here's kind of a high-level overview. It might be a little bit difficult to read. So at the top you have a system model. You have requirements, diagrams, behavior, structure and parametrics.

Within requirements, there's actually a specific requirements diagram that's supposed to represent the relationship between requirements in your system. And I'll show an example of that. In behavior, there are diagrams to describe kind of the activity of your system, the sequence of events that may happen.

There's a state machine diagram if you want to model your system as a state and transition between those states, things like that. In the structure, there's diagrams that go over the decomposition of your system. What is your system? And what parts make up your system? Both the logical decomposition and the physical decomposition.

And then there's topology. How are they all connected? What are the characteristics of the interfaces? Things like that.

And then parametrics, which is kind of adding constraints and numbers to all these things, whether they be logical constraints, mathematical constraints, things like that. Similar to OPM, SysML has no built-in analysis capabilities, so you can't like run a model or calculate an equation in SysML. You can't really do that. But very frequently the tools that implement system that I'll show you have that kind of analysis capability built into the tools, as opposed to the language.

So you can do things like use a parametric diagram with a bunch of equations to create a system of equations that you then can solve, whether it be in the tool, or you can move it to an external tool, like Mathematica or something and solve it, and then bring that information back into your system. You also can do kind of sequence-based computation. If you have an activity diagram that says first you have to build this part of your system and then this part of your system, there's things in sequence, things in parallel. You can run simulations like that where it's all about, have you done everything you need to do to get to the next step? Things in a more logical flow, as opposed to actually mathematical equations, you can do those sorts of computations as well.

One note is that these diagrams, although they are the main way to define your system and interface with the model, are not the model themselves. So you can create links between diagrams. If an element shows up in one diagram and that element shows up in another diagram, that's the same element.

If you make changes in one diagram, that's going to propagate to all your diagrams. So there's kind of a database backend to this whole model that encompasses all the information. So instead of having a bunch of isolated block diagrams, they're really just views into this model that's hidden in a backend database.

So we'll talk a little bit about the applications of SysML. First is requirements engineering. As I said, when you can explicitly model requirements in the relationship between requirements and your system, you can do a lot more.

The way that it's typically done nowadays is with tons of documents-- I'm not sure if you've ever actually developed a system, but there's an ungodly amount of documents. I've experienced that firsthand. It's a real pain. There are tools like DOORS that will enable you to link requirements to other requirements, and things like that that help you manage your requirements.

But what if you had a really explicit tie between your requirements and your system? You can actually represent in SysML. And I'll show you a little bit about this.

You can represent in SysML a textural requirement, you know, the mass of your system must be less than 5 kilograms, or something like that. You can tie that requirement directly to the mass property of your system. You can [INAUDIBLE]. You can start building constraints.

Your requirements aren't just textual statements, they're actually constraints upon properties of your system. Those are the types of things you can start to do with SysML. Do you have a question?

AUDIENCE: [INAUDIBLE]

MARK CHODAS: Yeah. Yeah, so that isn't something that's built into the language. But that is something you can do with-- basically there's a whole API. And you can interface with the model. I'll show you this, actually, in my demo.

But you can build in rules and constraints that say, check, for example, that all my requirements, at least how they've been defined, are satisfied. You can run that check, and it will tell you have they been satisfied or not. And that's something that's really powerful that you can't really do with existing kind of techniques. Yeah.

AUDIENCE: Thank you. And can you also, for example, requirement changes, like the master system has to be this much, as opposed to this much, then would it go through and check until you have to now look at this, this, and this, and then that affect like-

MARK CHODAS: So you're getting down in the weeds. That's something that would be awesome if you could do. That really-- again, that's not something that SysML enables you to do natively. But it gives you the language and the syntax to be able to write queries that give you that type of information.

That's kind of where the cutting edge is right now is, can we do that? Can we get that type of information from a SysML model? That's something I'm really interested in for my PhD thesis. So yeah, that's something that I think is possible and would be really great to have in the development process of a system.

So the next bullet here is on a system description. So actually one of the fundamental questions is, how do you describe a domain-specific system within SysML? I'll show you that SysML has a pretty strong notion of inheritance, and classes, and things like.

It's object-oriented. And so one of the questions is-- I'm in the space system. So how do you describe a spacecraft in SysML?

SysML is very general. But how do you actually represent, for example, a CNDH system in SysML? What are the types of attributes that are typically found? How do you represent that?

How does it interface with other parts of your system? Those type of questions. That's another active area of research, domain-specific modeling.

And then finally, as I said, integration with external analysis tools. So there's quite a lot of papers in the literature about going from a SysML model to MATLAB to SDK to Thermal Desktop, external modeling tools, taking that information out of the model, doing an analysis, putting it back in the model. And actually, I think there I was going to talk about Simscape, which is a kind of analysis tool, external analysis tool. And there's actually been papers written on, how do you take SysML information from a SysML model, pull it into Simscape process it, and put it back in the model?

So let's talk about the diagrams. There are nine types of diagrams in SysML. And I'll try and just give you a brief explanation of what they do. I won't go into the syntax for all of them because there's quite a bit of detail in the syntax. But I'll show you some examples of a couple of them.

So we'll go from left to right. So there's two main classes, behavior and structure, similar to OPM. In the behavior diagram, in the behavior diagram category you have activity diagrams, which basically represent flows of activities. So you do this, then you do this, then you do this. Those can be tied to system elements. If this system element has this sort of function or property or performs this operation on another part of the system, you can represent that link as well.

There's a sequence diagram, which is more about logical ordering. So if you have, for example, a multi-threaded software system, and you have different threads that may need different other threads to communicate with them or finish their computation before that, that way you can execute things like that, you can do that sort of interfacing between different threads of activities in a sequence diagram. This is one of the diagram types that was kind of inherited directly from UML. So there is a very kind of strong software element to that diagram.

There's a state machine diagram. So obviously, state machines are very powerful. If your system has various states, if things in your system have various states, you can represent that in a state machine diagram and then talk about, what are the criteria for transitioning between states? What would trigger or cause a transition between states?

What are guards that must be met before you can transition states? Things like that. That's where you represent the state machine diagram.

These type of diagrams are very powerful for describing things like concept of operations. So there has been some work-- I did an internship at JPL a couple of summers ago. And they were trying to build up this capability to model a concept of operations for a spacecraft.

So what are all the power modes of everything? What are the time-- you know, it spends this amount of time in this power mode, then it transitions here. For example, I can give an orbit of a spacecraft. That's sort of the thing that you can do with this set of behavior diagrams.

And then use case analysis. Again, it's mostly focused on early concept development, stakeholders. How do they interface with the system? Where do they derive value?

How does the user interact with the system? Things like that. That's where you put a use case diagram.

Going over to structure, the block definition diagram is where you define the structure of your system. So the logical or physical decomposition, I'll show you an example of this. So if your system is a spacecraft, it has various subsystems, if you want to decompose it logically as a thermal subsystem, a structure subsystem, ADCS subsystem, things like that, you can also decompose it physically so your spacecraft has solar arrays, it has instruments, it has thrusters, things like that. You can represent those types of things in a block definition diagram.

And then internal block diagram is where you describe the ties with the interfaces between all the components of your system. And this can be at varying different levels of abstraction, as I'll show you. Parametric diagram is kind of a subdiagram type of the internal block diagram. So you can, again, put constraints, mathematical, logical, things like that, on your interfaces and begin to build up the infrastructure for doing computation in the model.

And then a packaging diagram is not terribly important. It's really focused on the organization of your model. How do you scope things? It's kind of a modeling diagram as opposed to a representation of your system.

Then last of all is the requirements diagram up top. So again, I'll show you a good example of this. But that's where you represent, how are your requirements related to your system? And then you can see here what's been modified and what's been taken from UML and the new diagram types of requirement and parametric.

There were a couple of diagram types that were eliminated from UML that were pretty software-specific. I think there is one called the deployment diagram, like how has your software been deployed across various servers or users? Things like that. That's not really--that's a pretty software-specific thing, so in a general system you might not care about that all that much. So that diagram was removed.

So let me quickly go over some of the syntax. So this is an Interface Block Diagram, an IBD. And this is the type of diagram that I find is really interesting representing these interfaces.

So here's the system engineering ontology we typically talk about. So this is basically a model of an avionics board. So you have things like voltage converters. You have memory, volatile, nonvolatile. We have an FPGA, which is our main computational unit.

We talk about these as being parts of a system in terms of system engineering ontology. Then these are these interfaces or these lines right here. So in SysML, the way we talk about it is these are part properties of the system. It's kind of like an instantiation type of thing. What we're saying is all of these parts can represent independent of each other.

And then you define a property of that part as being part of a different part, if that makes any sense at all. So for example, this is a board. This main electronics board is this whole block. And then that has some blocks within this block, which represent the subparts that make up the board. And these are called part properties of this overall block.

We have these green little boxes, which are called ports. And again, that's to support this kind of system-independent modeling. So you can model like a voltage converter independent of any type of system as maybe an input voltage and output voltage. You can define what ranges those are, things like that. And you can model those interfaces using ports.

And then these connectors, which are called connectors in SysML which represent the interfaces, represent how each part is tied into the kind of larger system. And you could, for example, check that you don't have any empty ports. If a part needs an input voltage, you could run a script that checks that all the parts have all their ports kind of satisfied. That's something you can do with SysML.

So before I get into the case study, I want to talk a little bit about what you're going to see. As Olly said, I work on something called REXIS, which is the REgolith X-Ray Imaging Spectrometer. It's an x-ray spectrometer that's flying on NASA's OSIRIS-REx asteroid sample return mission. I've been working on it since 2011, when I was a senior here all throughout my master's, and then now for my PhD.

Basically we're going to measure x-rays that are fluoresced from the asteroid surface in order to tell what the elemental composition of the asteroid is. So that's our main science goal. And that will basically enable us to categorize where the asteroid is within the different meteorite types that have been defined on the ground based upon existing meteorite samples.

So what I did for my master's thesis was modeled the design history of REXIS. So how has our design evolved from the very beginning where it was very open ended, very abstract, and you'll see this, to the current design, which current in this case was CDR which was over a year ago now. Right now the current state of REXIS is we're almost ready to mount to the spacecraft. So it's very exciting.

Just to give you an idea of a timeline, this is something I'm sure Olly will talk about in this course, is the flow through the system development lifecycles. So we have a system requirements review back in January of 2012, system-- I think it's definition review, April 2012, preliminary design review January of 2013, and then critical design review February of 2014. So I created models at each of these design points in SysML and looked at, what are the lessons we could have learned?

We didn't use the SysML in REXIS. I was kind of looking back historically, what if we had used it? Could we have designed our system better in any sort of way?

So here is kind of a CAD representation of how a design evolved. And I think you can kind of get the idea. Back at SRR, a lot of things we didn't really know what they would look like. We didn't know what the interfaces would be. We didn't know what all the parts would be.

We had a little more development for SDR. You can see there's more arrows. The CAD is a little bit more detailed.

PDR, we had even more detail. This was actually like a buildable design. This turned out to not even be buildable.

And then we had more evolution between PDR and CDR to get to pretty much where the design is. There's actually been some evolution after this as well, as sometimes happens with a new system. But you can see just graphically kind of the increase in level of detail and level of fidelity of the state of our system throughout its development cycle. And I'll show you that and how that looks in the SysML model as well.

So one of the things you can do if you have a SysML model, is as I talked about, you can run queries on it and pull out information that's very difficult or impossible to get with our current deployment practices. So this is just looking at the different subassemblies within REXIS. What are the number of parts in each of the subassemblies?

So you can see the general trend is up for all of them, as you would expect. Some jump up very high. Some kind of stay basically where they were. But in general, they all increase. And this is something you might be able to do with looking at like a parts list or something like that with current methodologies.

But it would be very hard to get this information, which is the number of ports per assembly. So each interface has two ports. So these numbers divided by 2 basically equal the number of interfaces that we have in each subassembly.

And you can see, again, there's a general trend of increasing number of ports as you go through the lifecycle. So this is a piece of information you might want to use if you want to manage the complexity of your system. And you say, this subassembly is getting-- it's way too many interfaces, way too many parts. It's way too complex. We need to think about how we've logically arranged our system and maybe how can we rearrange it to make it more understandable and easier to work with.

And then you can divide the two. And you end up looking at how many-- this is ports per part in each subassembly at each of the design reviews. And you can see trends here, too, which are interesting.

So you can see in the beginning we didn't really know what we were doing. Some of these had a lot of parts, ports per part. Some of these had very few.

They all ended up stabilizing kind of in between the three and five ports per part range. And then you can look at the literature and say, well, typically systems tend to be between five and six ports per part. So what does that mean? Does that mean that our system didn't model it correctly? That's one possibility. It didn't model it to the lowest level of fidelity possible.

Does it mean that our system is too simple? Does it mean that we're missing something that we haven't thought about? Does it mean that our system, which was intended to be simple and cheap and implementable by students is actually achieving that goal because it's beneath what you typically expect?

Those are the types of questions you can ask with this data. But this data is not easy to get with the current methodologies. So this kind of very simple queries you can do gives you power on managing complexity in your system.

So let me now transition quickly to the demo. Let's see. There's no sound.

AUDIENCE: I know, but I need to sync it up to make it [INAUDIBLE].

OLIVIER DE WECK: So while Mark is setting up for the demo, are there any questions about SysML so far, any observations you guys have? Maybe at EPFL. Do you see the similarities and differences between OPM and SysML? What's the biggest difference between the two? There's two really important differences.

GUEST SPEAKER: No questions? Yes.

AUDIENCE: I was going to ask--

OLIVIER DE WECK: Hang on. Hang on. Is there anybody at EPFL who wants to comment on this?

AUDIENCE: No, there is no comment from EPFL.

OLIVIER DE WECK: OK. Good. That's fine. That's fine. Mark, are you set up?

GUEST SPEAKER: Almost.

OLIVIER DE WECK: What I would say is-- so first of all, OPM has only one type of diagram, and then you go really deep, sort of a deep hierarchy. SysML has nine different types of diagrams split between behavioral and structural. So that's one difference. And then the other is that the SysML is fundamentally object-oriented because it comes from object-oriented thinking and software, whereas OPM has objects and processes sort of at the same level. Those are two of the most important differences.

MARK CHODAS: All right. So what I'm showing you-- can everyone see the screen? Just let me know if you can't see the screen.

What I'm showing you right now is a tool called MagicDraw. There are basically a variety of tools provided by commercial vendors that enable you to build and work with a SysML model.

So SysML is a language, and then it's implemented in tools. And this just happens to be one that's fairly well, fairly widely utilized. Unfortunately, it's quite expensive.

But anyway, so what I'm showing you right now, and this is going to be a little bit difficult because, as you can see, you need a big screen. What I'm showing you right now is a requirements diagram. So you can see that right here. It's a requirements diagram.

And each of these blocks, as you can see by the tag here, is a requirement. So this is one of our operating criterias. Well operating temperature of all our components shall be maintained within operability limits. Straightforward requirement.

And you can create these satisfy relationships between that requirement and the components in the system that must satisfy that requirement. So right now, this is being done at SRR, as you can see up here. So this is very early in the design. So we don't have it broken down fully to all the components.

But we have-- here is our main electronics board. It has to satisfy that requirement. Here's our radiation cover. It has to satisfy that requirement. So you can tie these two things, the requirement and the part of the system that must satisfy the requirement.

And here I've tied it to parts. So these are called blocks, which are the fundamental unit in SysML, is a block. So these represent parts of our system. But you could tie it to a property of that part if you wanted to.

So then you get into the situation I talked about earlier where you have the requirement is on the mass of it must be less than this. You can tie it to the mass of the system itself, as opposed to the system. You can tie it to the actual property, which is very useful.

So let me quickly show you one of the cool things. If I delete these requirements, I'm not actually-- you can see there's no requirements. I'm not actually deleting them from the system itself, but I'm just doing it as I'm removing it from the diagram.

But if you wanted to, you should be able to look at related elements of the different blocks. So I just clicked on a block, and I can choose to show all the requirements, all the things that are satisfied, the requirements of that component of the system satisfies. And these requirements pop up.

So this is showing how the diagrams themselves aren't the model. There's actually backend to the model. And you can kind of work in the diagram and show or hide things however you want. But the information is actually kept behind the diagram, so to speak.

So my research looked into typologies. So let me just kind of show you a little bit about what I did. So this is a block definition diagram, again, defining all the parts of your system. And I just want to give you kind of a high level idea of the type of things that we saw.

So it's big, first of all, very big. These are all the parts of the system. So we start high like the mission context, and the mission context contains things like the environment and the spacecraft and then REXIS. And then you can break down REXIS. We have these various subassemblies broken down to parts. And that's how you get this tree structure.

And then this tree structure can be tied together. I can zoom in, but it's big again. So these are all the parts of our system. Let me give you an example.

So we have a couple of boards that we call our detector electronics. And those detector electronics have various ports. One thing they had to do, they had this port in here and this interface here, which connects to our CCDs. So this is showing how you can build interfaces and SysML. So here are the green boxes, again, or the ports. The lines are the connectors, and they're defining all the interfaces.

And you can see at a high level how complicated things get very quickly. This is the earliest, most abstract version of our design. And it already has a lot of complication. One thing you can do-- I talked about running scripts. I'll show you how that works. It's quite easy.

So I just ran a script on the model that told me to find the number of parts in the scope that I defined and the number of ports. And here's the output of that model right here. So the script itself is not even very complicated. It's like 50, 60 lines of code. And immediately I can pull out how many ports, how many parts, things like that, information down my system.

So let me kind of take you briefly through the development process. So that was, again, the highest, the most abstract, the earliest version of our system, SRR. This is SDR. You can see it's starting to get a bit more complex. And then you can go over to PDR, and it starts to get really scary.

And you can go to CDR, and it's just a nightmare. So I created all these systems, all these models by hand. You can actually build the model with the script, if you would like. You can basically do things like define a pattern and then apply that pattern to all the parts of that type.

That's all possible through the API.

And just to show you how much of a nightmare it was at CDR, let me run the same script on the CDR model. You have 230 parts and 900 interfaces. And this was not even modeling to the lowest level of fidelity possible. I didn't, for example, model all the components, all the capacitors, resistors, op amps, and stuff on the board.

And now you can already see it's quite large. Kind of the idea behind these models would be to extend this to the lowest level in a real system and use this, basically use all the capabilities that you had with the model to really manage your complexity in a way that is just not possible currently. And there's no way you can really mentally keep track of all these interfaces and understanding how your system is working.

So having this modeling capability and querying capability is really, really powerful. Yeah?

AUDIENCE: What was used during REXIS for the systems engineering? You had applied this after the fact. So what was used to create these block diagram or [INAUDIBLE].

MARK CHODAS: Do you mean during when we were developing REXIS?

AUDIENCE: Yeah.

MARK CHODAS: We basically didn't have this. So we were relying-- as you would typically do upon the capabilities of the system engineer or the team-- you'd to have documents. We have a ton of documents.

But I was talking about how things weren't buildable. I found a situation where because of the way we had done our thermal system, we were dumping spacecraft heat to space, which you don't want to do. I can explain why, but you don't want to do that. And that's something we didn't realize at the time. If we had tried to build it, we would have had this property of the system that we didn't know actually existed until I went back and looked at the model. So we definitely miss things. And this should have improved the design process, if we had been using it.

OLIVIER DE WECK: Great, so Mark, sort of to wrap up, because we've got to switch over to [INAUDIBLE], what's your recommendation for, let's say, students in the class got sort of intrigued by SysML. What's the next step?

MARK CHODAS: There's a couple different ways. Certainly if you're interested, let me know, and I can give you resources to further your understanding. As I said, working with these tools can be expensive to get these tools. So I can help you understand what that would take. There are beginning to be some companies that will do like SysML training courses that will sit down for a day or a week and teach you SysML, basically, how to work with the model, how to build the model.

I took one of those courses. It was really great. Yeah, talk to me if you're interested, and I can steer you down the right path.

OLIVIER DE WECK: Great, thanks, Mark.

MARK CHODAS: Sure.

OLIVIER DE WECK: Very good. So we did in 20 minutes what usually takes about a week, right?

MARK CHODAS: There's a lot more depth. There's a lot more depth.

OLIVIER DE WECK: Great, so we're going to switch over now to Modelica, which I think is-- we're going to maybe run a couple of minutes over today, but I think it's important we cover all three languages. So let's get the slides back up. And I'm going to switch here to Narek. He's another doctoral student in the group. And so introduce yourself and then tell us about Modelica.

NAREK SHOUGARIAN: Thank you.

OLIVIER DE WECK: Are you using this computer for slides or this computer?

NAREK SHOUGARIAN: For the demo, I'm going to use this one.

OLIVIER DE WECK: This one?

NAREK SHOUGARIAN: This one.

OLIVIER DE WECK: This one.

NAREK SHOUGARIAN: Thank you. So hello, everyone. My name is Narek. I'm a doctoral student at AeroAstro here. My background is in gas turbine engines, and specifically I've been looking at concept generation of gas turbine engines, automated concept generation. And the way I got acquainted with Modelica was that I needed to be able to rapidly reconfigure different concepts and simulate them mathematically, do physics-based simulations.

So in contrast to the first two languages, Modelica is a lot more about rapidly being able to build physical models of systems and reconfiguring them and reusing them for later on.

So like I mentioned, Modelica is primarily about modeling physics-based modeling of systems and rapidly being able to reuse models and reconfigure them. It's a language, again. It's not a tool like the first two that you heard about. There are many different tools which implement this language. But I'm going to start off with just describing how the language works and then go on to describe which tools that you can use.

In contrast to SysML, there are a couple of really good free tools that you can use and rapidly get into. And there are a lot of libraries that you can use with hundreds, even thousands, of actually basic components that you can use for modeling. So it's a declarative language. And what I mean by declarative is that in sequential sort of programming, you write commands. And you make assignments to various parameters.

Here, you just describe the governing equations of the components you want to simulate. And there's no particular order in which you do this. The models are acausal. There's no direction to flows. All you really need to do is describe what ports, like Mark mentioned. It's similar in this situation-- what kind of ports you have, what kind of interfaces the components can have with other components, and the governing equations and the parameters and variables.

It's a multi-domain modeling language, so it's like agnostic to what kind of domain you're working in. It's not particular to electrical engineering. For example, I'm going to be showing an electrical engineering example and also show a gas turbine engine example with aeroelastic vibrations. So you don't necessarily-- you're not attached to any specific domain. It's also object-orientated, and it enables you to decompose systems into subsystems or recombine them and look at them at various levels of abstraction.

It's designed to be efficient. So this is a quote from Professor Peter Fritzen at the Linkopings University. So these are just about the scale of problems that you can solve with the Modelica language. Obviously, it depends on what kind of equations you're talking about, but it's designed to be a very efficient way of simulating systems.

So I really want to talk a little bit more about this idea of acausal modeling that I mentioned before. So on the left hand side is an assignment. And that's typically what you do when you program in MATLAB, just the MATLAB normal scripts. f is assigned to ma or p is assigned the value of rho rt for the equation of state of a gas. And what that means is that you know what the mass and acceleration are, and you figure out the force. You assign that value to the force. In Modelica, there's almost none of this. It's equations not assignments. And what I mean by that is this equation can be written in any which way, as long as your system has the same number of equations as unknowns.

The tool that you're using will interpret the language and will solve your problem. So you can write this in any which way you want in any order. As long as your problem is properly constrained, the tool that you're using will interpret it and solve the problem for you.

So just to go into a little bit more detail, all of Modelica's and also other acausal modeling language that I'll mention in a little bit-- which one of them is Simscape-- have essentially three parts. They're designed to be extremely simple. The first, like Mark mentioned, are ports. It's essentially identical to SysML in a way. Ports are the ways with which components can share information, material, or energy, for example. You can define any kind of port you want. All you need to really do is define what kind of parameters it carries-- like for gas, for example, temperature, pressure, and mass flow; or for electrical ports, voltage and current.

The second part of any kind of model that you're building in Modelica or Simscape, which I'll mention a little later, are variables and parameters. So you just declare those. And the third part are governing equations. The point is that there's nothing else. It's just that. And I'll just brief briefly go through a very, very simple example.

So for example, a capacitor. This is the entire code for a capacitor, and this is what will generate a visual image of a capacitor with the correct ports for you. First you have pins that carry voltage and current. So the key thing to notice here is that there are fundamentally two types of variables-- flow variables and normal variables here.

Flow variables are ones for which the Kirchoff's current law applies. So mass flow, for example, every time you connect 15 components together in a network, mass flow into that network needs to be conserved. So the sum of mass flows into any node has to be zero. That doesn't apply to the standard variables.

So then you essentially define the parameters, the variables. And you need to define the governing equations. And that's the capacitor for you. There are slightly more complicated components that you can use, for example a pressure drop component. Did I lose my sharing? No.

In this case, the main thing to take away from this one is that if you have complicated mathematics describing the fluid mechanics in a component, you can actually initialized with one model and then go to a full turbulent simulation. That's what this is doing here.

So just to get to the tool side of the equation, the language I just showed you, it's the same across all the tools. But there are many different tools which you can use free and commercial to actually run these models. The main one that you'll be using if you want to get deeper into this is OpenModelica. It's free. It's actually become quite user-friendly. And there's a link in the slides from which you can download it.

There's one from Wolfram. It's integrated with Mathematica, which is quite useful. And there's a free trial of it as well. There's Dymola. There are other ones. But mainly, I think OpenModelica is the one that you guys will be using.

OLIVIER DE WECK: OK, so in other words, I think we're going to wrap up and then have you back next week.

NAREK SHOUGARIAN: OK.

OLIVIER DE WECK: Are you around next week?

NAREK SHOUGARIAN: Yeah, yeah. sure.

OLIVIER DE WECK: Just to stay around. So basically we're going to finish this lecture next week. I think it's important enough that you really see the demo and see sort of the-- and it actually ties in kind of nicely with next week's topic is concept generation. This is the next step in the V. And since

Narek, your research is also on concept generation, it'll tie in nicely.

So I think we're going to stop here for today. So you heard about the general idea of system modeling languages that are rigorous, that have ontology, syntax and, semantics. There's different of these that have been proposed, developed. Some are used more than others. There's really important differences between them. So OPM is very conceptual. SysML is based on UML and has these different type of diagrams and can really help you flesh out your design in more detail. And then Modelica allows you to build these blocks. It's acausal or declarative. And you can actually simulate the physics of the system pretty readily.

So the big picture here, just to wrap up here, the big picture is the following. And we'll come back to this next week. The big picture is basically that system engineering is in a transition phase. The classic way of doing system engineering really for the last 50 years is on the left, document-centric. Write your requirements. Do your drawings. Even CAD, you know, computer-aided design, is great, but it only essentially does the mechanical part of the design.

And so the result of that is as you get-- even REXIS, REXIS is a box, like shoe-box size, basically. And it's going to fit on a much bigger spacecraft. And you saw how much complexity is there. And it gets very, very difficult to manage all this information, to prevent errors, oversights, any change that you make. It doesn't propagate automatically in these documents.

So the transition is happening to the right side, a model-centric way to do system engineering. And think of paperless engineering. Everything you're doing is in a model. The models are linked. The models are executable. The models automatically propagate any changes that you make in requirements or design. We're not quite there yet. But that is where things are moving.

And so keep that in mind. So there's no new assignment this week. So next week, we have A-2, which is do, the requirements. Please let us know if you have any-- we're here for you. So I'm going to have office hours now on the WebEx. You have the link.

If you have any questions about A-2, don't be shy to email me or [? Yuanna ?] or [? Leislu ?] at EPFL. We're really here to answer your questions. So next week topic, we're going to finish on Modelica, and concept generation is going to be sort of our-- creativity concept generation is our main topic next week.