**TLA$^+$ Video Course – Lecture 1**

Leslie Lamport

# INTRODUCTION TO TLA$^+$

This video should be viewed in conjunction with a Web page.
To find that page, search the Web for *TLA+ Video Course* .

The TLA$^+$ Video Course,
Lecture 1
Introduction to TLA$^+$

# WHAT IS TLA$^+$?

A language for high-level modeling of digital systems.

TLA+ is a language for high-level modeling of digital systems.

A language for high-level modeling of digital systems.

Has tools for checking those models.

TLA+ is a language for high-level modeling of digital systems.

It has tools for checking those high-level models.

A language for high-level modeling of digital systems.

Has tools for checking those models.

Most important tool: the TLC model checker.

TLA+ is a language for high-level modeling of digital systems.

It has tools for checking those high-level models.

The most important of these tools is the TLC model checker.

A language for high-level modeling of digital systems.

Has tools for checking those models.

Most important tool: the TLC model checker.

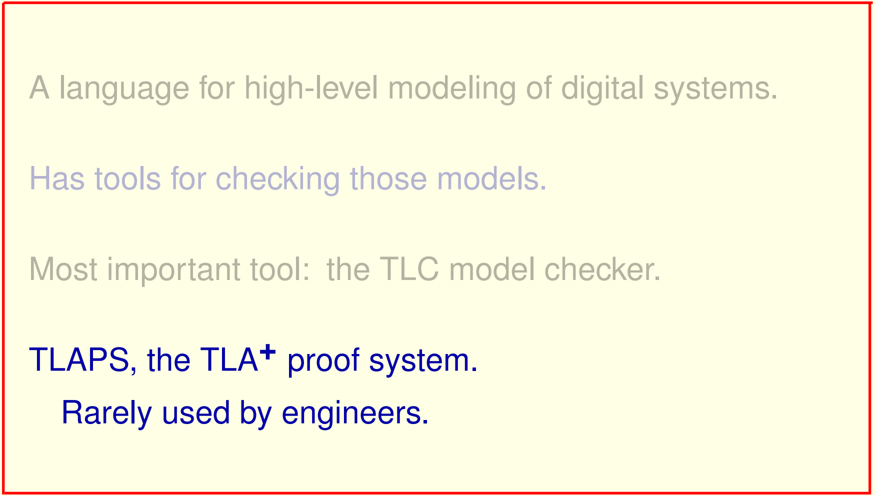
TLAPS, the TLA$^+$ proof system.

TLA+ is a language for high-level modeling of digital systems.

It has tools for checking those high-level models.

The most important of these tools is the TLC model checker.

Another tool is TLAPS, the TLA+ proof system.
But writing and checking proofs is a lot of work,



A language for high-level modeling of digital systems.

Has tools for checking those models.

Most important tool: the TLC model checker.

TLAPS, the TLA$^+$ proof system.

Rarely used by engineers.

TLA+ is a language for high-level modeling of digital systems.

It has tools for checking those high-level models.

The most important of these tools is the TLC model checker.

Another tool is TLAPS, the TLA+ proof system.
But writing and checking proofs is a lot of work,
and it will rarely be used by engineers.

[ slide 7 ]

A language for high-level modeling of digital systems.

Has tools for checking those models.

Most important tool: the TLC model checker.

The TLA$^+$ Toolbox, an IDE.

There's also the TLA$^+$ Toolbox, an Integrated Development Environment for writing specifications and running the tools on them.

A language for high-level modeling of digital systems.

A language for high-level modeling of digital systems.

Digital systems include

Digital systems include

A language for high-level modeling of digital systems.

Digital systems include

- Algorithms

Digital systems include algorithms,

A language for high-level modeling of digital systems.

Digital systems include

- Algorithms
- Programs

Digital systems include algorithms, programs,

A language for high-level modeling of digital systems.

Digital systems include

- Algorithms
- Programs
- Computer systems

Digital systems include algorithms, programs, and computer systems.

A language for high-level modeling of digital systems.

High-level means

Digital systems include algorithms, programs, and computer systems.

For those digital systems, high-level means

A language for high-level modeling of digital systems.

High-level means

- At the design level

Digital systems include algorithms, programs, and computer systems.

For those digital systems, high-level means at the design level,

A language for high-level modeling of digital systems.

High-level means

- At the design level
- Above the code level

Digital systems include algorithms, programs, and computer systems.

For those digital systems, high-level means at the design level, above the code level.

To model critical parts of digital systems

TLA+ is used to model critical parts of digital systems

While

To model critical parts of digital systems

Abstract away

TLA+ is used to model critical parts of digital systems

While abstracting away both

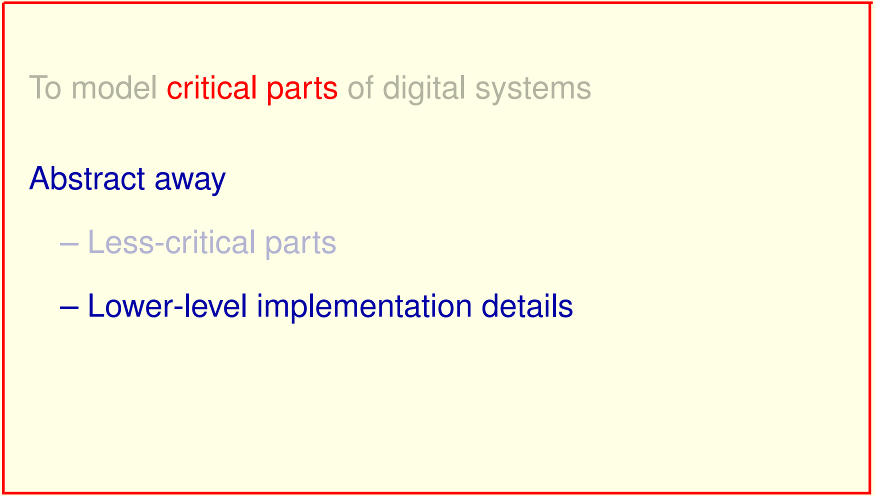
To model critical parts of digital systems

Abstract away

- Less-critical parts

TLA+ is used to model critical parts of digital systems

While abstracting away both less-critical parts



To model critical parts of digital systems

Abstract away

- Less-critical parts
- Lower-level implementation details

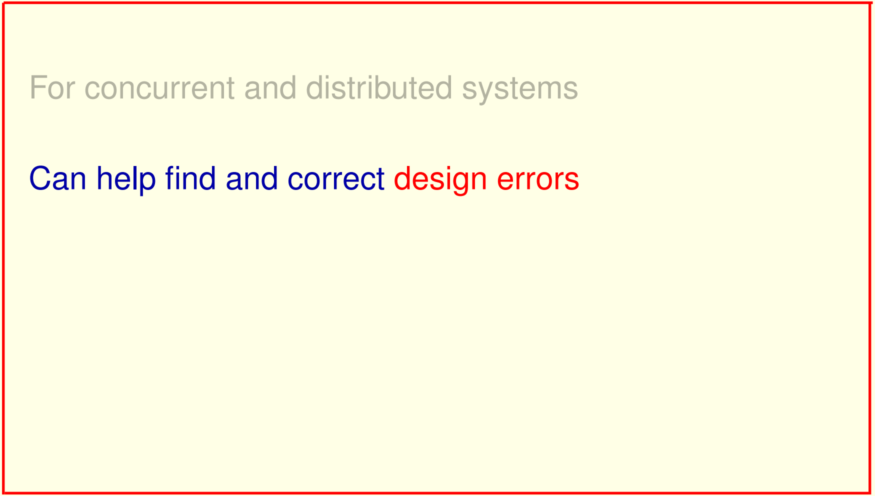
TLA+ is used to model critical parts of digital systems

While abstracting away both less-critical parts

and lower-level implementation details.

# For concurrent and distributed systems

TLA+ was designed for modeling concurrent and distributed systems.



For concurrent and distributed systems

Can help find and correct design errors

TLA+ was designed for modeling concurrent and distributed systems.

It can help you find and correct design errors

[ slide 22 ]

For concurrent and distributed systems

Can help find and correct design errors

- Errors hard to find by testing

TLA+ was designed for modeling concurrent and distributed systems.

It can help you find and correct design errors

– including errors that are extremely difficult to detect by testing

For concurrent and distributed systems

Can help find and correct design errors

- Errors hard to find by testing
- Before writing any code

TLA+ was designed for modeling concurrent and distributed systems.

It can help you find and correct design errors

– including errors that are extremely difficult to detect by testing

– before you write a single line of code

[ slide 24 ]

# For concurrent and distributed systems

## For concurrent and distributed systems

If you design them

If you design such systems

[ slide 26 ]

For concurrent and distributed systems

If you design them and care if they work,

If you design such systems

and care about whether they work properly,

For concurrent and distributed systems

If you design them and care if they work,

you should use TLA$^+$

If you design such systems

and care about whether they work properly,

you should be using TLA+

[ slide 28 ]

For concurrent and distributed systems

If you design them and care if they work,

you should use TLA$^+$ or some other method

If you design such systems

and care about whether they work properly,

you should be using TLA+ or some other method

For concurrent and distributed systems

If you design them and care if they work,

you should use TLA$^+$ or some other method

to precisely specify and check your designs.

If you design such systems

and care about whether they work properly,

you should be using TLA+ or some other method

for precisely specifying and checking your designs.

[ slide 30 ]

I don’t know exactly what else TLA$^+$ is useful for.

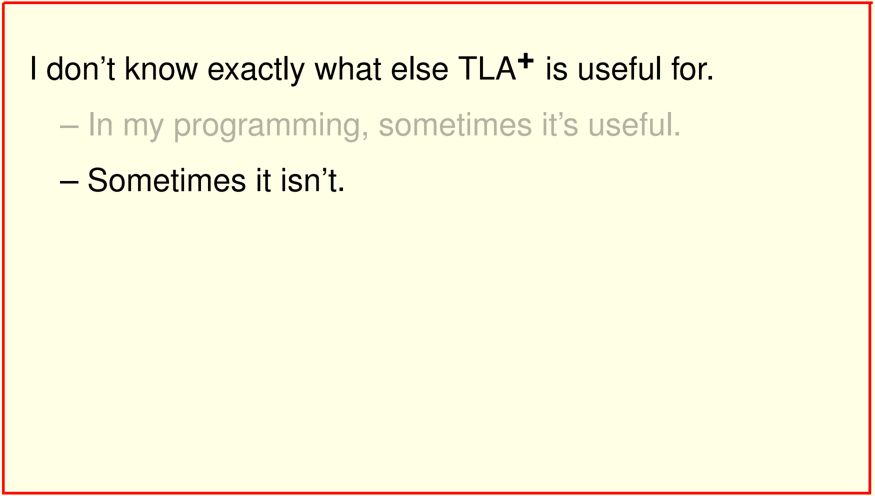
I don’t know exactly what else TLA+ is good for.

I don’t know exactly what else TLA$^+$ is useful for.

- In my programming, sometimes it’s useful.

I don’t know exactly what else TLA+ is good for.

In the small amount of programming that I do, sometimes I find it to be very useful.



I don’t know exactly what else TLA$^+$ is useful for.

– In my programming, sometimes it’s useful.

– Sometimes it isn’t.

I don’t know exactly what else TLA+ is good for.

In the small amount of programming that I do, sometimes I find it to be very useful.
Sometimes it’s of no use.

I don’t know exactly what else TLA$^+$ is useful for.

- In my programming, sometimes it’s useful.
- Sometimes it isn’t.

For most computer programmers and engineers

I don’t know exactly what else TLA+ is good for.

In the small amount of programming that I do, sometimes I find it to be very useful. Sometimes it’s of no use.
I do know that for most computer programmers and engineers

I don’t know exactly what else TLA$^+$ is useful for.

- In my programming, sometimes it’s useful.
- Sometimes it isn’t.

For most computer programmers and engineers

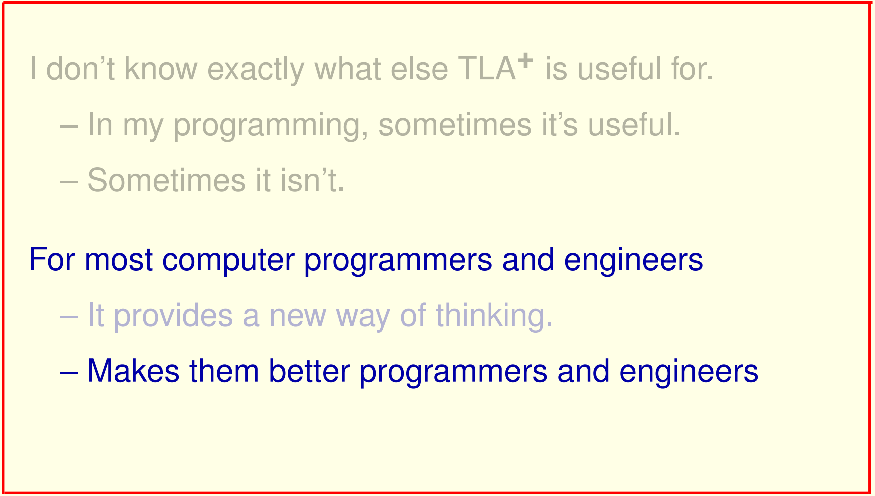
- It provides a new way of thinking.

I don’t know exactly what else TLA+ is good for.

In the small amount of programming that I do, sometimes I find it to be very useful.
Sometimes it’s of no use.
I do know that for most computer programmers and engineers

TLA+ provides a new way of thinking about what they do.

[ slide 35 ]



I don’t know exactly what else TLA$^+$ is useful for.

- In my programming, sometimes it’s useful.
- Sometimes it isn’t.

For most computer programmers and engineers

- It provides a new way of thinking.
- Makes them better programmers and engineers

I don’t know exactly what else TLA+ is good for.

In the small amount of programming that I do, sometimes I find it to be very useful. Sometimes it’s of no use.

I do know that for most computer programmers and engineers

TLA+ provides a new way of thinking about what they do.

And that this way of thinking can make them better programmers and engineers,

[ slide 36 ]

I don’t know exactly what else TLA$^+$ is useful for.

– In my programming, sometimes it’s useful.

– Sometimes it isn’t.

For most computer programmers and engineers

– It provides a new way of thinking.

– Makes them better programmers and engineers even when TLA$^+$ and its tools are not useful.

I don’t know exactly what else TLA+ is good for.

In the small amount of programming that I do, sometimes I find it to be very useful.
Sometimes it’s of no use.
I do know that for most computer programmers and engineers

TLA+ provides a new way of thinking about what they do.

And that this way of thinking can make them better programmers and engineers,

even in cases where the TLA+ language and tools are not useful.

# ABSTRACTION

Abstraction.

# Who am I?

What kind of clown am I?. . . claiming that I know what can make you think better?

Why should you pay attention to me?

# Who am I?

What kind of clown am I?. . . claiming that I know what can make you think better?

Why should you pay attention to me?

This is not the time to be modest.

## Who am I?

I’ve done seminal research in the theory of concurrent and distributed systems.

I have done seminal research in the theory of concurrent and distributed systems,

## Who am I?

I’ve done seminal research in the theory of concurrent and distributed systems.

I won the Turing award.

I have done seminal research in the theory of concurrent and distributed systems, for which I won the Turing Award.

Who am I?

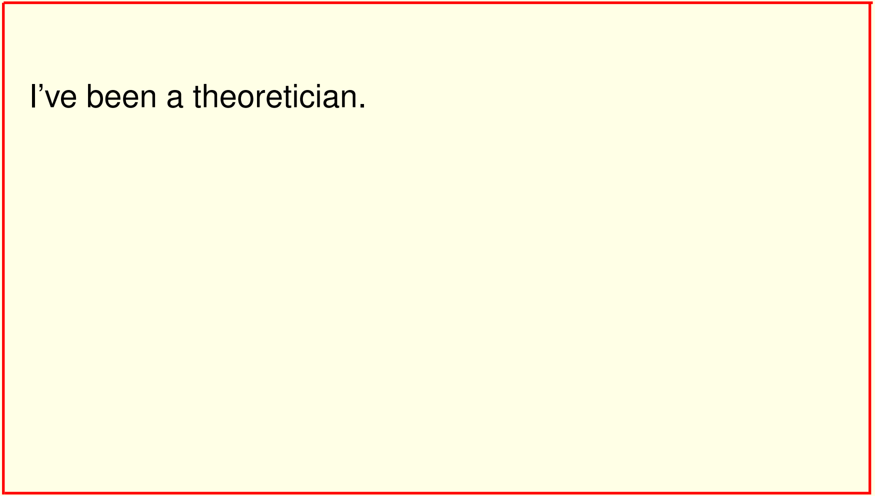
I’ve done seminal research in the theory
of concurrent and distributed systems.

I won the Turing award.

Look me up on the Web.

I have done seminal research in the theory of concurrent and distributed systems, for which I won the Turing Award.

You can look me up on the web.



I’ve been a theoretician.

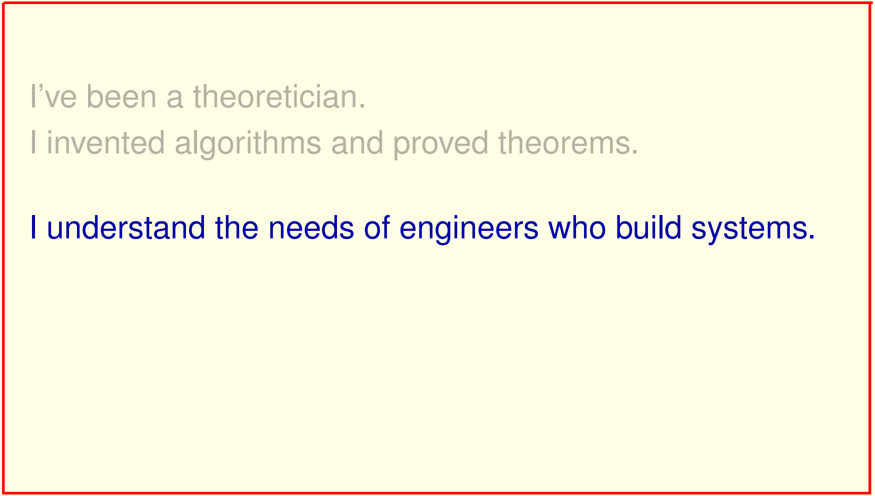
I’ve been a theoretician.

I’ve been a theoretician.

I invented algorithms and proved theorems.

I’ve been a theoretician.

I’ve invented algorithms and proved theorems.



I’ve been a theoretician.

I invented algorithms and proved theorems.

I understand the needs of engineers who build systems.

I’ve been a theoretician.
I’ve invented algorithms and proved theorems.

But I understand the needs of computer engineers who build real systems,

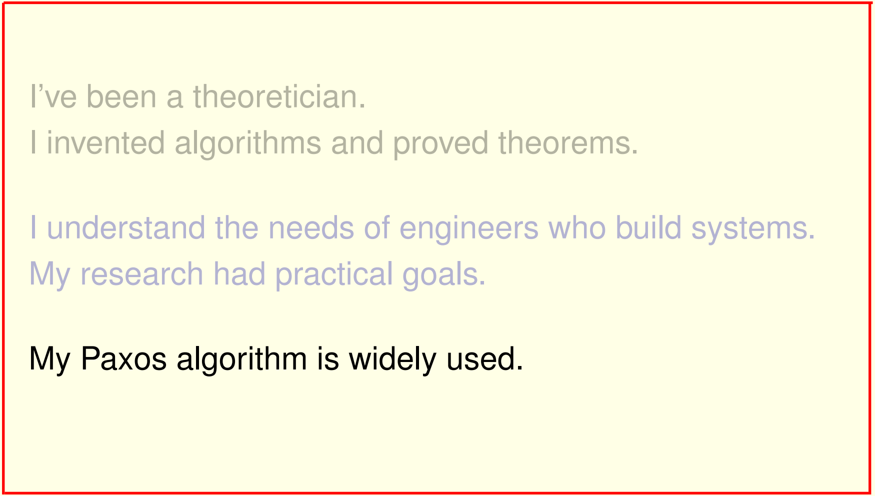
I’ve been a theoretician.
I invented algorithms and proved theorems.

I understand the needs of engineers who build systems.
My research had practical goals.

I’ve been a theoretician.
I’ve invented algorithms and proved theorems.

But I understand the needs of computer engineers who build real systems, and my research always had practical goals.

[ slide 47 ]



I’ve been a theoretician.

I invented algorithms and proved theorems.

I understand the needs of engineers who build systems.

My research had practical goals.

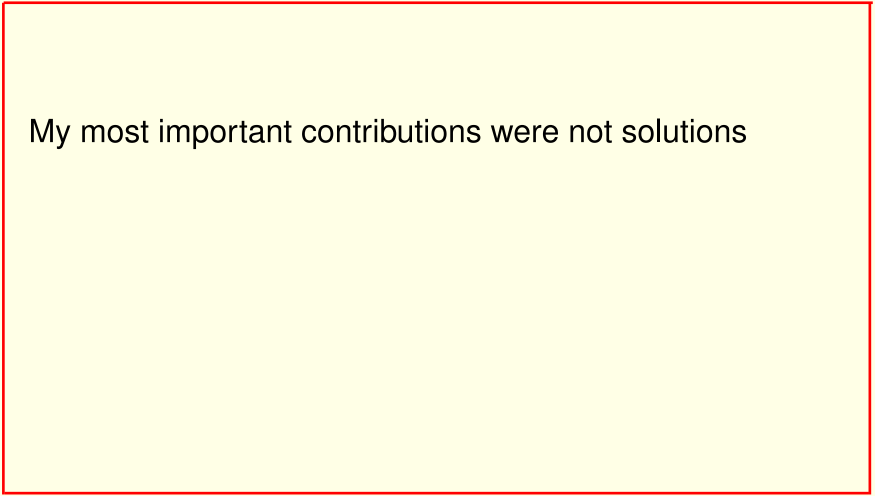
My Paxos algorithm is widely used.

I’ve been a theoretician.
I’ve invented algorithms and proved theorems.

But I understand the needs of computer engineers who build real systems, and my research always had practical goals.

My Paxos algorithm is widely used to implement fault-tolerant distributed systems.

[ slide 48 ]



My most important contributions were not solutions

My most important contributions have not been solutions to recognized problems,

My most important contributions were not solutions,

but recognizing important problems

My most important contributions have not been solutions to recognized problems,

but rather precisely stating important problems that had not been recognized

My most important contributions were not solutions,
but recognizing important problems
that had been obscured by details.

My most important contributions have not been solutions to recognized problems,
but rather precisely stating important problems that had not been recognized
because they were obscured by irrelevant details.

This ability to recognize fundamental ideas is why

My most important contributions were not solutions,
but recognizing important problems
that had been obscured by details.

My *Time Clocks* paper is still read after 40 years.

My most important contributions have not been solutions to recognized problems,
but rather precisely stating important problems that had not been recognized because they were obscured by irrelevant details.

This ability to recognize fundamental ideas is why my *Time Clocks* paper is still required reading in university system-building courses, 40 years after it was written.

Simplifying by removing details is called **abstraction**.

The process of simplification by removing irrelevant details is called **abstraction**

Simplifying by removing details is called **abstraction**.

Abstraction is the most important part of engineering.

The process of simplification by removing irrelevant details is called **abstraction**

Abstraction is perhaps the most important part of engineering.

Simplifying by removing details is called **abstraction**.

Abstraction is the most important part of engineering.

It lets us understand complex systems.

The process of simplification by removing irrelevant details is called **abstraction**

Abstraction is perhaps the most important part of engineering.

Only through abstraction can we understand complex systems.

Simplifying by removing details is called **abstraction**.

Abstraction is the most important part of engineering.

It lets us understand complex systems.

We can’t get systems right if we don’t understand them.

The process of simplification by removing irrelevant details is called **abstraction**

Abstraction is perhaps the most important part of engineering.

Only through abstraction can we understand complex systems.

And we can’t get systems right if we don’t understand them.

I’m very good at abstraction, and I believe that

[ slide 56 ]

Simplifying by removing details is called **abstraction**.

Abstraction is the most important part of engineering.

It lets us understand complex systems.

We can’t get systems right if we don’t understand them.

TLA$^+$ will teach you to be better at abstraction.

The process of simplification by removing irrelevant details is called **abstraction**

Abstraction is perhaps the most important part of engineering.

Only through abstraction can we understand complex systems.

And we can’t get systems right if we don’t understand them.

I’m very good at abstraction, and I believe that
using TLA+ will teach you to be better at it.

Brannon Batson:

Here’s what Brannon Batson, a former Intel Engineer, said:

[ slide 58 ]

Brannon Batson:

All quotes are edited.

Here's what Brannon Batson, a former Intel Engineer, said:

[When quoting what people wrote, I have made some small changes to eliminate unimportant details while preserving the intended meaning.]

Brannon said:

Brannon Batson:

> The hard part of learning to write TLA$^+$ specs is learning to think abstractly about the system.

Here’s what Brannon Batson, a former Intel Engineer, said:

[When quoting what people wrote, I have made some small changes to eliminate unimportant details while preserving the intended meaning.]

Brannon said: The hard part of learning to write TLA+ specs is learning to think abstractly about the system.

Brannon Batson:

> The hard part of learning to write TLA$^+$ specs is learning to think abstractly about the system.

With experience, engineers learn how to do it.

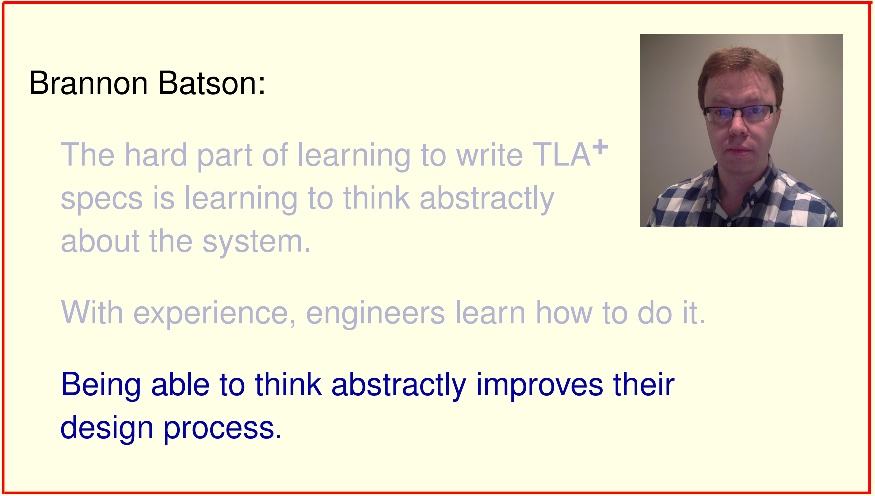
Here’s what Brannon Batson, a former Intel Engineer, said:

[When quoting what people wrote, I have made some small changes to eliminate unimportant details while preserving the intended meaning.]

Brannon said:  The hard part of learning to write TLA+ specs is learning to think abstractly about the system.

With experience, engineers learn how to do it.

[ slide 61 ]



Brannon Batson:

The hard part of learning to write TLA$^+$ specs is learning to think abstractly about the system.

With experience, engineers learn how to do it.

Being able to think abstractly improves their design process.

Here's what Brannon Batson, a former Intel Engineer, said:

[When quoting what people wrote, I have made some small changes to eliminate unimportant details while preserving the intended meaning.]

Brannon said:  The hard part of learning to write TLA+ specs is learning to think abstractly about the system.

With experience, engineers learn how to do it.

Being able to think abstractly improves their design process.

[ slide 62 ]

# WHAT ENGINEERS SAY

This is pretty abstract.

All this talk of abstraction is pretty abstract.

This is pretty abstract.

## Engineers

If you’re an engineer, you’re probably not looking for an education in abstraction. You

This is pretty abstract.

Engineers want help designing systems now.

If you’re an engineer, you’re probably not looking for an education in abstraction. You want something that will help you design systems now.

This is pretty abstract.

Engineers want help designing systems now.

What do engineers think of TLA$^+$ ?

If you’re an engineer, you’re probably not looking for an education in abstraction. You want something that will help you design systems now.

So, you probably want to know more about what engineers think of TLA+.

This is pretty abstract.

Engineers want help designing systems now.

What do engineers think of TLA$^+$ ?

Engineers in industry don’t say what they do.

If you’re an engineer, you’re probably not looking for an education in abstraction. You want something that will help you design systems now.

So, you probably want to know more about what engineers think of TLA+.

Engineers in industry who build complex systems usually don’t write about how they do it. Fortunately, there was

This is pretty abstract.

Engineers want help designing systems now.

What do engineers think of TLA+ ?

Engineers in industry don’t say what they do.

An exception: a team at Amazon Web Services.

If you’re an engineer, you’re probably not looking for an education in abstraction. You want something that will help you design systems now.

So, you probably want to know more about what engineers think of TLA+.

Engineers in industry who build complex systems usually don’t write about how they do it. Fortunately, there was one exception: a team at Amazon Web Services.

[ slide 69 ]

## Amazon a leader in cloud services.

Amazon is currently the leading provider of cloud services,

Amazon a leader in cloud services.

Amazon Web Services builds their cloud infrastructure.

Amazon is currently the leading provider of cloud services,

and Amazon Web Services builds and maintains their cloud infrastructure.

Amazon a leader in cloud services.

Amazon Web Services builds their cloud infrastructure.

They build large, complex systems.

Amazon is currently the leading provider of cloud services,
and Amazon Web Services builds and maintains their cloud infrastructure.

They build large, complex systems,

Amazon a leader in cloud services.

Amazon Web Services builds their cloud infrastructure.

They build large, complex systems.

They care about getting them right.

Amazon is currently the leading provider of cloud services,

and Amazon Web Services builds and maintains their cloud infrastructure.

They build large, complex systems, and they care about getting them right because their customers rely on them.

I have no official connection with Amazon.

I have no official connection with Amazon,

I have no official connection with Amazon.

Chris Newcombe contacted me

I have no official connection with Amazon,

but Chris Newcombe who was then at Amazon Web Services contacted me with some questions about TLA+

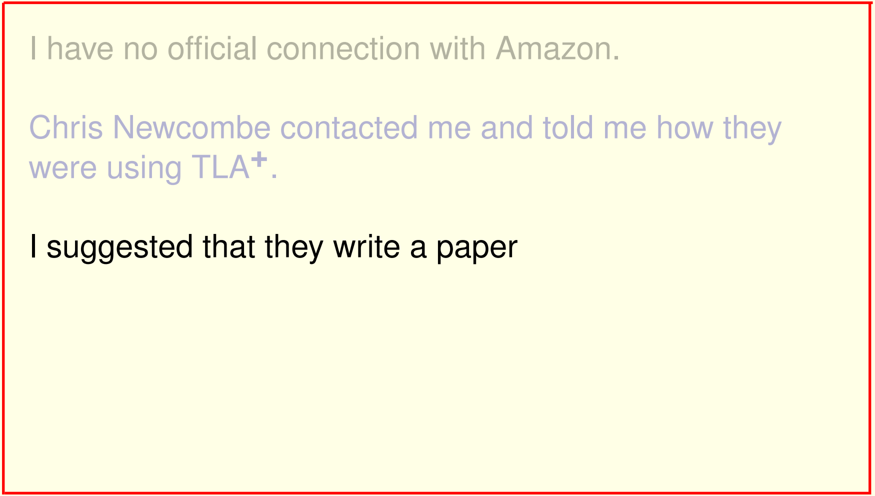
I have no official connection with Amazon.

Chris Newcombe contacted me and told me how they were using TLA$^+$.

I have no official connection with Amazon,

but Chris Newcombe who was then at Amazon Web Services contacted me with some questions about TLA+

and told me a little about how they were using it.



I have no official connection with Amazon.

Chris Newcombe contacted me and told me how they were using TLA$^+$.

I suggested that they write a paper

I have no official connection with Amazon,

but Chris Newcombe who was then at Amazon Web Services contacted me with some questions about TLA+

and told me a little about how they were using it.

I suggested that they write a paper about their experience,

[ slide 77 ]

I have no official connection with Amazon.

Chris Newcombe contacted me and told me how they were using TLA$^+$.

I suggested that they write a paper, and they wrote:

I have no official connection with Amazon,

but Chris Newcombe who was then at Amazon Web Services contacted me with some questions about TLA+

and told me a little about how they were using it.

I suggested that they write a paper about their experience, and the result was:

[ slide 78 ]

I have no official connection with Amazon.

Chris Newcombe contacted me and told me how they were using TLA$^+$.

I suggested that they write a paper, and they wrote:

**How Amazon Web Services Uses Formal Methods**

Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu,
Marc Brooker, and Michael Deardeuff

*Communications of the ACM*

April 2015, Vol. 58, No. 4, pages 66–73

*How Amazon Web Services Uses Formal Methods* by Chris and others.

I have no official connection with Amazon.

Chris Newcombe contacted me and told me how they were using TLA$^+$.

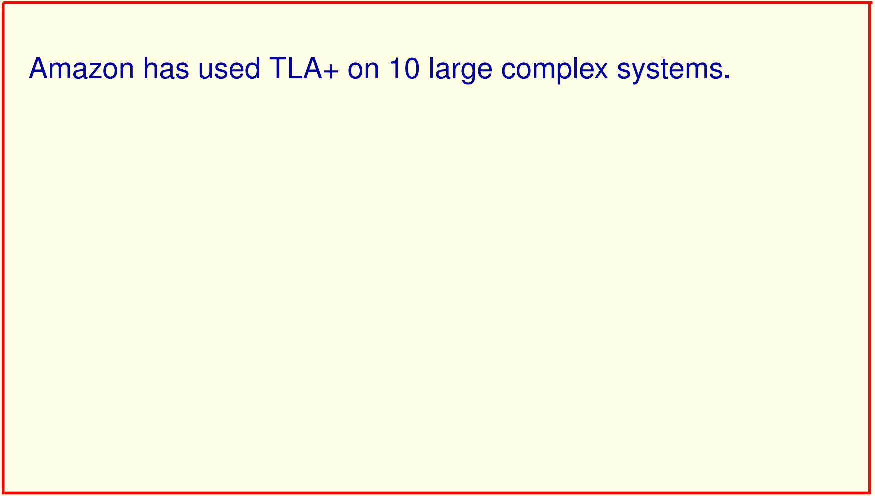
I suggested that they write a paper, and they wrote:

**How Amazon Web Services Uses Formal Methods**

Here’s a quote from the paper:

*How Amazon Web Services Uses Formal Methods* by Chris and others.

Here’s a quote from the paper



Amazon has used TLA+ on 10 large complex systems.

Amazon engineers have used TLA$^+$ on 10 large complex real-world systems.

Amazon has used TLA+ on 10 large complex systems.

In each, TLA$^+$ has added significant value,

Amazon engineers have used TLA$^+$ on 10 large complex real-world systems.

In each, TLA$^+$ has added significant value,

Amazon has used TLA+ on 10 large complex systems.

In each, TLA+ has added significant value, either finding subtle bugs we would not have found by other means,

Amazon engineers have used TLA+ on 10 large complex real-world systems.

In each, TLA+ has added significant value,

either finding subtle bugs we are sure we would not have found by other means,

[ slide 83 ]

Amazon has used TLA+ on 10 large complex systems.

In each, TLA$^+$ has added significant value, either finding subtle bugs we would not have found by other means,

or giving us enough confidence to make aggressive optimizations without sacrificing correctness.

or giving us enough understanding and confidence to make aggressive performance optimizations without sacrificing correctness.

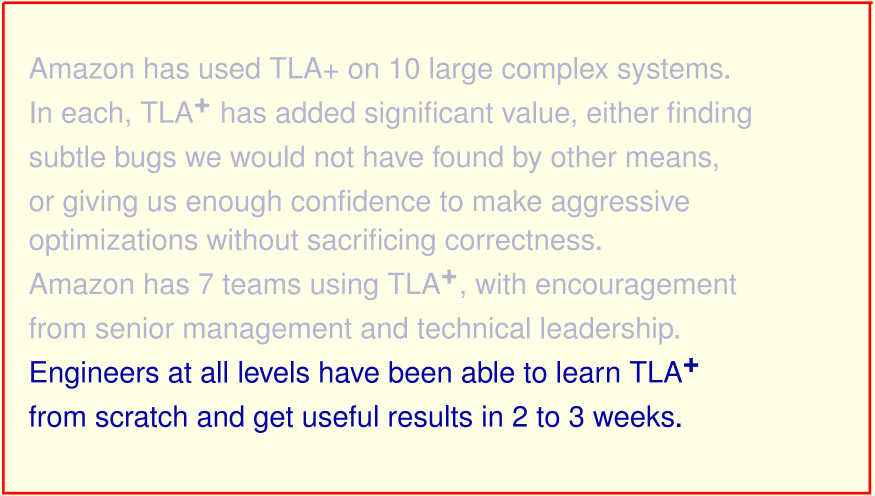
Amazon has used TLA+ on 10 large complex systems.

In each, TLA$^+$ has added significant value, either finding subtle bugs we would not have found by other means, or giving us enough confidence to make aggressive optimizations without sacrificing correctness.

Amazon has 7 teams using TLA$^+$, with encouragement from senior management and technical leadership.

or giving us enough understanding and confidence to make aggressive performance optimizations without sacrificing correctness.

Amazon now has seven teams using TLA+, with encouragement from senior management and technical leadership.



Amazon has used TLA+ on 10 large complex systems.

In each, TLA$^+$ has added significant value, either finding subtle bugs we would not have found by other means, or giving us enough confidence to make aggressive optimizations without sacrificing correctness.

Amazon has 7 teams using TLA$^+$, with encouragement from senior management and technical leadership.

Engineers at all levels have been able to learn TLA$^+$ from scratch and get useful results in 2 to 3 weeks.

or giving us enough understanding and confidence to make aggressive performance optimizations without sacrificing correctness.

Amazon now has seven teams using TLA+, with encouragement from senior management and technical leadership.

Engineers from entry level to principal have been able to learn TLA+ from scratch and get useful results in two to three weeks.

Amazon has used TLA$^+$ on 10 large complex systems.

In each, TLA$^+$ has added significant value, either finding subtle bugs we would not have found by other means, or giving us enough confidence to make aggressive optimizations without sacrificing correctness.

Amazon has 7 teams using TLA$^+$, with encouragement from senior management and technical leadership.

Engineers at all levels have been able to learn TLA$^+$ from scratch and get useful results in 2 to 3 weeks.

They used TLA+ on 10 systems.

Amazon has used TLA$^+$ on 10 large complex systems.

In each, TLA$^+$ has added significant value, either finding subtle bugs we would not have found by other means, or givin essive optimiz

Written in late 2013.

Amazo ragement from senior management and technical leadership. Engineers at all levels have been able to learn TLA$^+$ from scratch and get useful results in 2 to 3 weeks.

They used TLA+ on 10 systems.

That number 10 came from late 2013.

Amazon has used TLA$^+$ on ~~10~~ 14 large complex systems.

In each, TLA$^+$ has added significant value, either finding subtle bugs we would not have found by other means,

or givin essive

optimiz

In 2014.

Amazo ragement from senior management and technical leadership.

Engineers at all levels have been able to learn TLA$^+$ from scratch and get useful results in 2 to 3 weeks.

They used TLA+ on 10 systems.

That number 10 came from late 2013.

By some time in 2014 they had used it in 14 systems.

Amazon has used TLA$^+$ on ? ~~10~~ large complex systems.

In each, TLA$^+$ has added significant value, either finding subtle bugs we would not have found by other means,

or givin essive

optimiz

Now.

Amazo ragement

from senior management and technical leadership.

Engineers at all levels have been able to learn TLA$^+$ from scratch and get useful results in 2 to 3 weeks.

They used TLA+ on 10 systems.

That number 10 came from late 2013.

By some time in 2014 they had used it in 14 systems.

I don’t have any later information.

[ slide 90 ]

Amazon has used TLA$^+$ on large complex systems. In each, TLA$^+$ has added significant value, either finding subtle bugs we would not have found by other means, or givi ssive optimi

You can download the paper.

Amazo agement from senior management and technical leadership. Engineers at all levels have been able to learn TLA$^+$ from scratch and get useful results in 2 to 3 weeks.

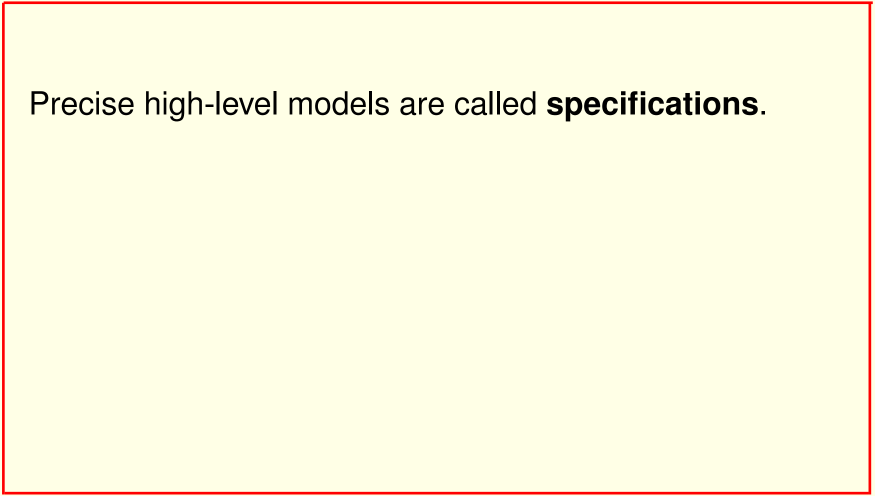
They used TLA+ on 10 systems.

That number 10 came from late 2013.

By some time in 2014 they had used it in 14 systems.

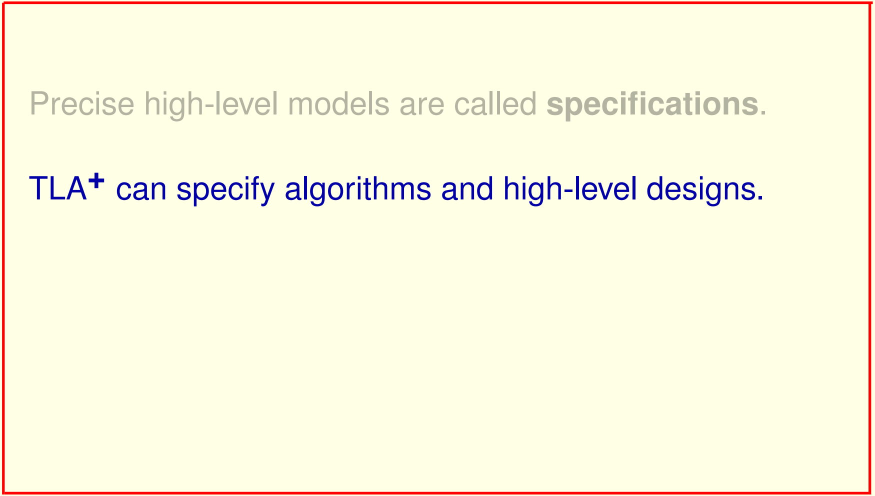
I don’t have any later information.

You might want to download the complete paper now.



Precise high-level models are called **specifications**.

Precise high-level models are called specifications.

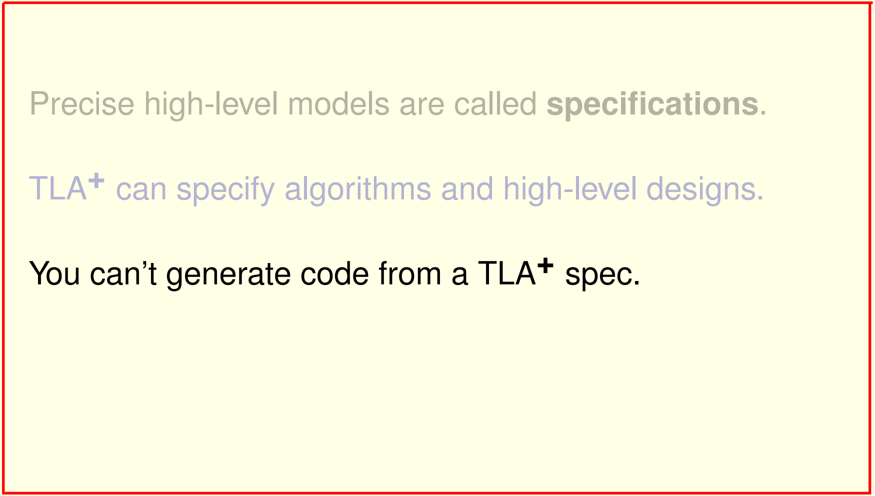


Precise high-level models are called **specifications**.

TLA$^+$ can specify algorithms and high-level designs.

Precise high-level models are called specifications.

TLA+ can specify algorithms and high-level system designs.



Precise high-level models are called **specifications**.

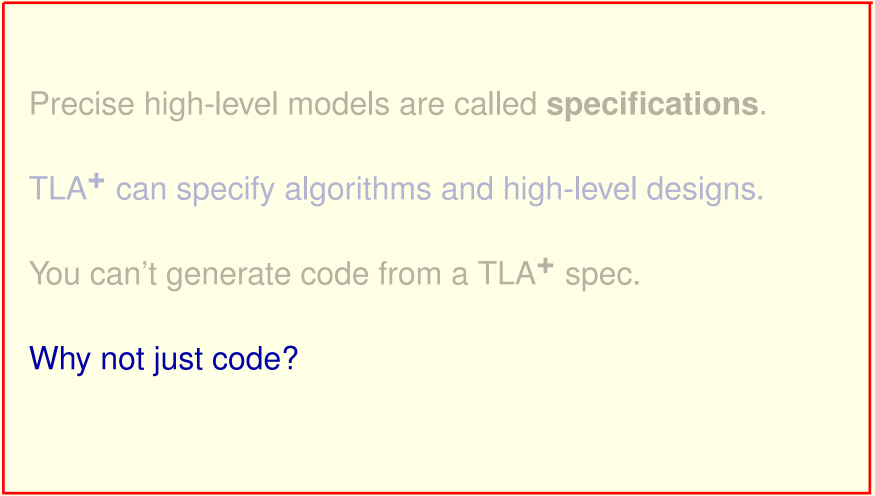
TLA$^+$ can specify algorithms and high-level designs.

You can’t generate code from a TLA$^+$ spec.

Precise high-level models are called specifications.

TLA+ can specify algorithms and high-level system designs.

But you can’t automatically generate code from a TLA+ specification.
So, what good is a spec if it doesn’t produce code?



Precise high-level models are called **specifications**.

TLA$^+$ can specify algorithms and high-level designs.

You can’t generate code from a TLA$^+$ spec.

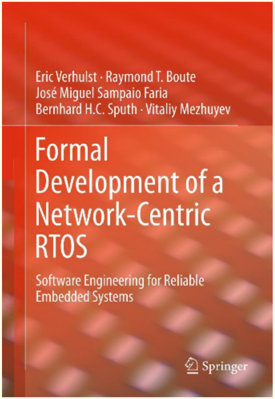
Why not just code?

Precise high-level models are called specifications.

TLA+ can specify algorithms and high-level system designs.

But you can’t automatically generate code from a TLA+ specification.
So, what good is a spec if it doesn’t produce code?

Instead of writing a spec, why not spend the time improving the code?



Eric Verhulst · Raymond T. Boute
José Miguel Sampaio Faria
Bernhard H.C. Sputh · Vitaliy Mezhuyev

Formal Development of a Network-Centric RTOS

Software Engineering for Reliable Embedded Systems

Springer

A real-time operating system designed using TLA$^+$.

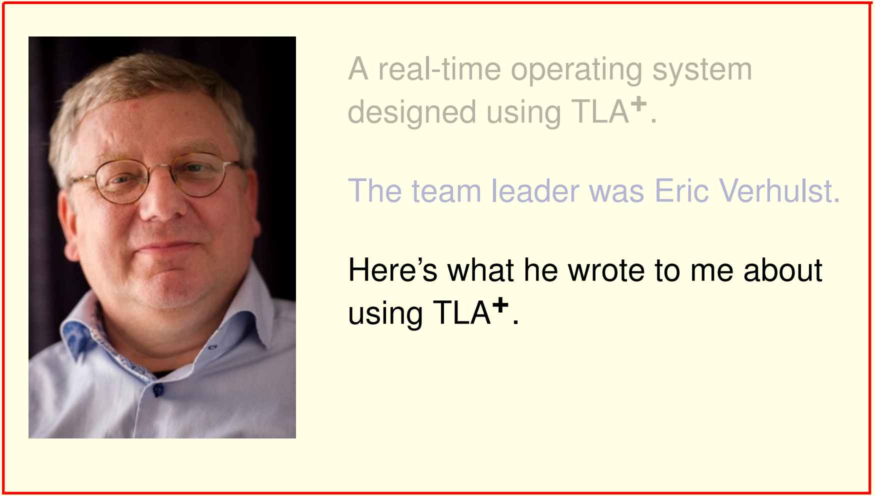
This book describes how a real-time operating system was designed using TLA+.

A real-time operating system designed using TLA$^+$.

The team leader was Eric Verhulst.

This book describes how a real-time operating system was designed using TLA+.

The team that built the system was led by Eric Verhulst.



A real-time operating system designed using TLA$^+$.

The team leader was Eric Verhulst.

Here’s what he wrote to me about using TLA$^+$.

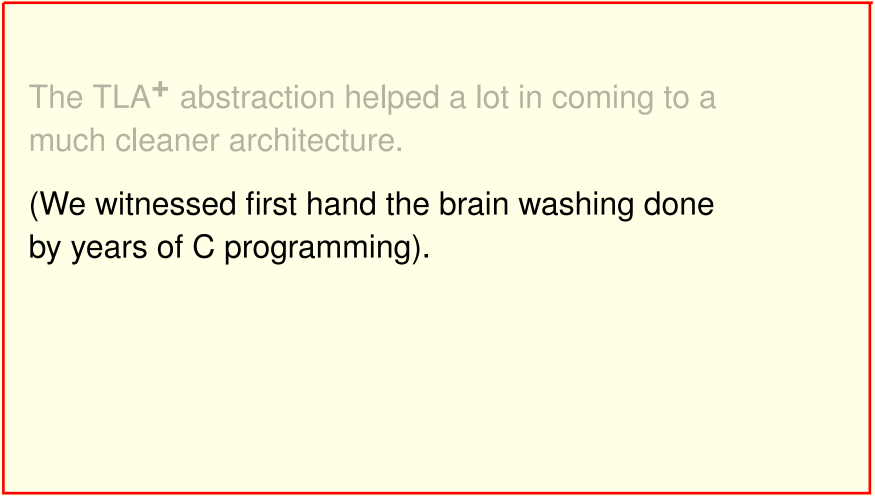
This book describes how a real-time operating system was designed using TLA+.

The team that built the system was led by Eric Verhulst.

Here is what he wrote to me about their experience using TLA+.

The TLA$^+$ abstraction helped a lot in coming to a much cleaner architecture.

The TLA+ abstraction helped a lot in coming to a much cleaner architecture.



The TLA$^+$ abstraction helped a lot in coming to a much cleaner architecture.

(We witnessed first hand the brain washing done by years of C programming).

The TLA+ abstraction helped a lot in coming to a much cleaner architecture.

(We witnessed first hand the brain washing done by years of C programming).

The TLA$^+$ abstraction helped a lot in coming to a much cleaner architecture.

(We witnessed first hand the brain washing done by years of C programming).

One of the results was that the code size is about 10× less than the previous version.

The TLA+ abstraction helped a lot in coming to a much cleaner architecture.

(We witnessed first hand the brain washing done by years of C programming).

One of the results was that the code size is about 10 times less than the previous version.

The code size is 10× less than the previous version.

10 times less than the previous version.

The code size is 10× less than the previous version.

The previous version flew on the Rosetta spacecraft.

10 times less than the previous version.

The previous version was flown on the European Space Agency’s Rosetta spacecraft that orbited a comet from 2014 to 2016.

The code size is 10× less than the previous version.

The previous version flew on the Rosetta spacecraft.

When building a spacecraft, you try to make the code small.

10 times less than the previous version.

The previous version was flown on the European Space Agency's Rosetta spacecraft that orbited a comet from 2014 to 2016.

And when you're building a spacecraft, you try to make the code that goes into it small.

[ slide 104 ]

You don’t get a 10× reduction in code size by better coding.

You don’t get a factor of 10 reduction in the size of code that flies in a spacecraft by better coding.

You don’t get a 10× reduction in code size by better coding.

You get it by “coming to a cleaner architecture”.

You don’t get a factor of 10 reduction in the size of code that flies in a spacecraft by better coding.

You get it by "coming to a cleaner architecture".

You don’t get a 10× reduction in code size by better coding.

You get it by “coming to a cleaner architecture”.

An architecture is a higher-level specification

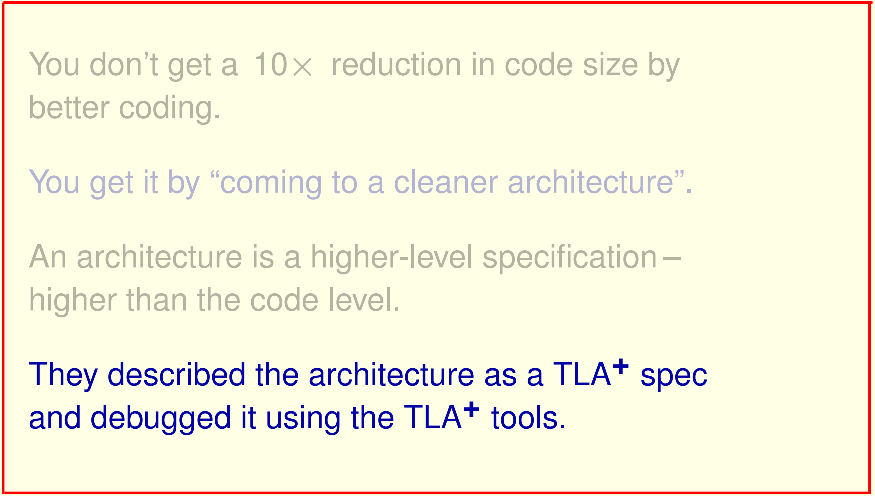
And an architecture is described by a higher-level specification –

You don’t get a 10× reduction in code size by better coding.

You get it by “coming to a cleaner architecture”.

An architecture is a higher-level specification – higher than the code level.

And an architecture is described by a higher-level specification – higher than the code level.



You don't get a 10× reduction in code size by better coding.

You get it by "coming to a cleaner architecture".

An architecture is a higher-level specification – higher than the code level.

They described the architecture as a TLA$^+$ spec and debugged it using the TLA$^+$ tools.

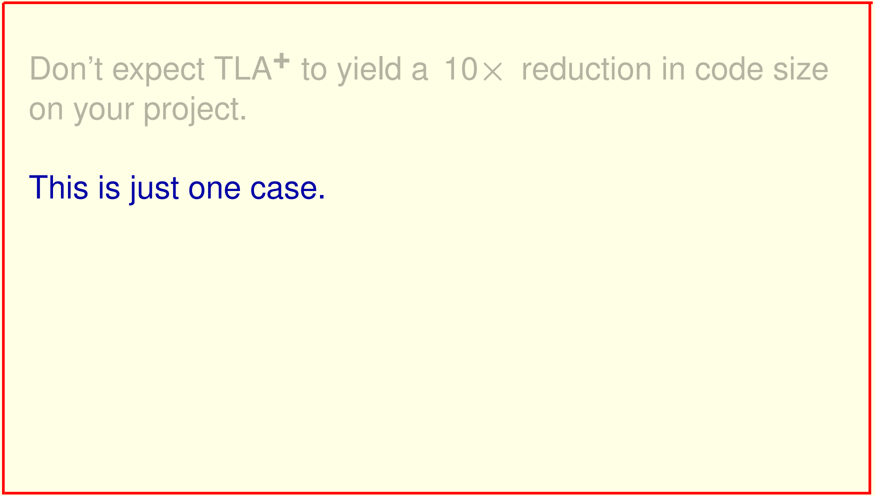
And an architecture is described by a higher-level specification – higher than the code level.

They described the architecture as a TLA+ spec and debugged it using the TLA+ tools.

[ slide 109 ]

Don’t expect TLA$^+$ to yield a 10× reduction in code size on your project.

Don’t expect TLA+ to yield such a reduction in code size on your project.



Don’t expect TLA$^+$ to yield a 10× reduction in code size on your project.

This is just one case.

Don’t expect TLA+ to yield such a reduction in code size on your project.

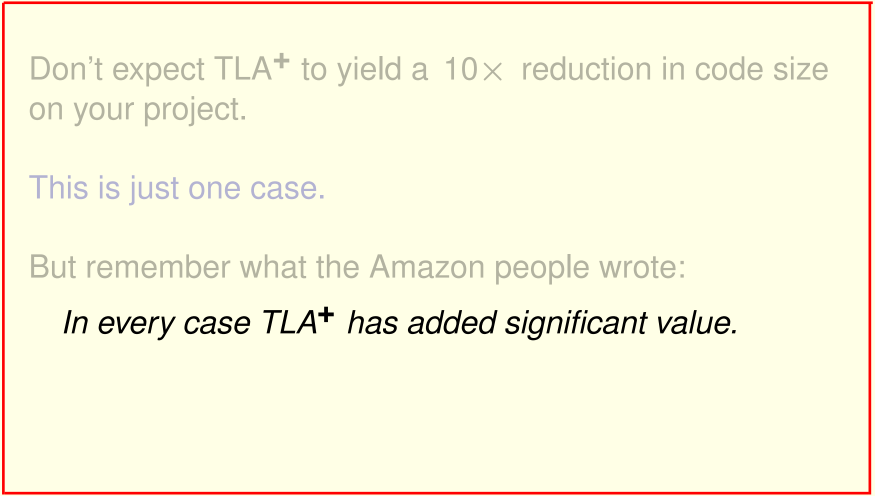
This is just one case.

Don’t expect TLA$^+$ to yield a 10× reduction in code size on your project.

This is just one case.

But remember what the Amazon people wrote:

But remember what the Amazon Web Services people wrote:



Don’t expect TLA$^+$ to yield a $10\times$ reduction in code size on your project.

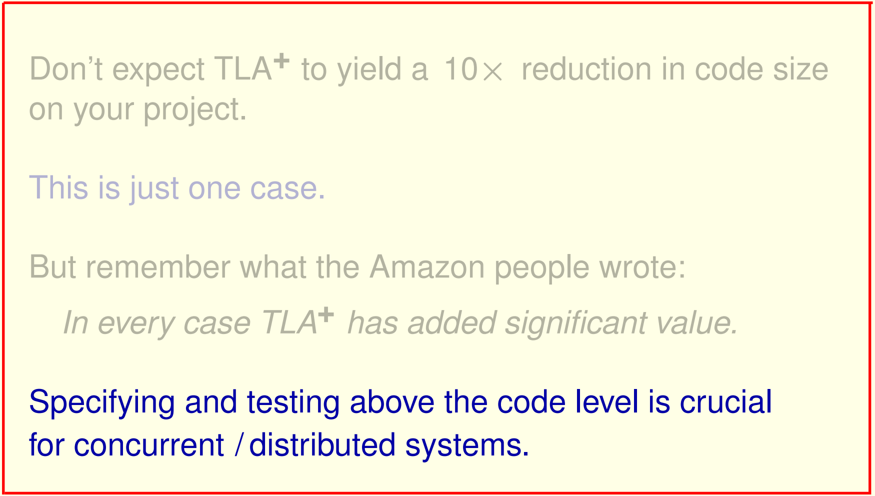
This is just one case.

But remember what the Amazon people wrote:

*In every case TLA$^+$ has added significant value.*

But remember what the Amazon Web Services people wrote:

In every case TLA+ has added significant value.



Don’t expect TLA$^+$ to yield a 10× reduction in code size on your project.

This is just one case.

But remember what the Amazon people wrote:

> *In every case TLA$^+$ has added significant value.*

Specifying and testing above the code level is crucial for concurrent / distributed systems.

But remember what the Amazon Web Services people wrote:

> In every case TLA+ has added significant value.

Specifying and testing above the code level is crucial when designing concurrent or distributed systems.

# WHAT CAN YOU CHECK WITH TLA$^+$?

We use TLA$^+$ to ensure the systems we build “work right”.

We use TLA+ to try to ensure that the systems we build “work right”.

We use TLA$^+$ to ensure the systems we build “work right”.

Working right means satisfying certain properties.

We use TLA+ to try to ensure that the systems we build “work right”.

Working right means satisfying certain properties.

We use TLA$^+$ to ensure the systems we build “work right”.

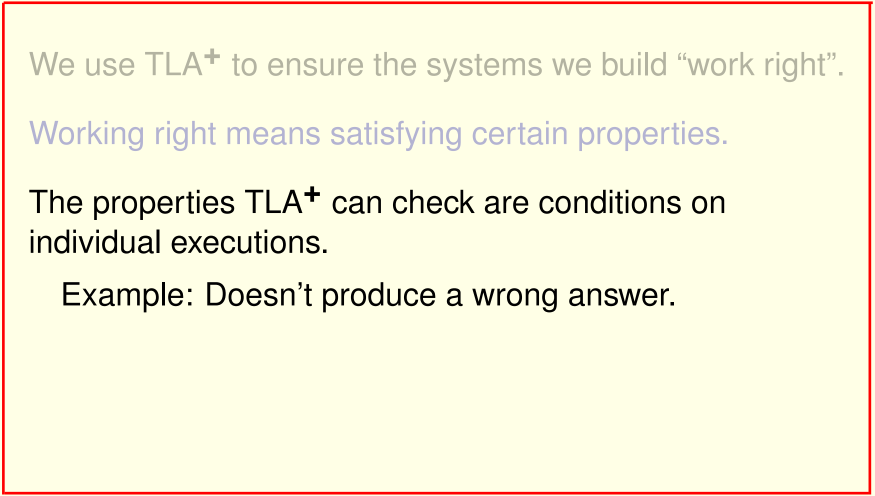
Working right means satisfying certain properties.

The properties TLA$^+$ can check are conditions on individual executions.

We use TLA+ to try to ensure that the systems we build “work right”.

Working right means satisfying certain properties.

The class of properties that we can check with TLA+ are ones that express conditions on individual executions.



We use TLA$^+$ to ensure the systems we build “work right”.

Working right means satisfying certain properties.

The properties TLA$^+$ can check are conditions on individual executions.

Example: Doesn’t produce a wrong answer.

We use TLA+ to try to ensure that the systems we build “work right”.

Working right means satisfying certain properties.

The class of properties that we can check with TLA+ are ones that express conditions on individual executions.

One example of such a property is: The system doesn’t produce a wrong answer.

[ slide 119 ]

We use TLA$^+$ to ensure the systems we build “work right”.

Working right means satisfying certain properties.

The properties TLA$^+$ can check are conditions on individual executions.

> Example: Doesn’t produce a wrong answer.
>
> Can examine an individual execution and see if it produced a wrong answer.

You can examine an individual execution and see whether or not it produced a wrong answer.

We use TLA$^+$ to ensure the systems we build “work right”.

Working right means satisfying certain properties.

The properties TLA$^+$ can check are conditions on individual executions.

Example: Doesn’t produce a wrong answer.

Can examine an individual execution and see
if it produced a wrong answer.

System satisfies property if every execution does.

You can examine an individual execution and see whether or not it produced a wrong answer.

The system satisfies this property if and only if every possible execution satisfies it.

We use TLA$^+$ to ensure the systems we build “work right”.

Working right means satisfying certain properties.

The properties TLA$^+$ can check are conditions on individual executions.

A property TLA$^+$ can’t check

Here’s a property that TLA+ can’t check

We use TLA$^+$ to ensure the systems we build “work right”.

Working right means satisfying certain properties.

The properties TLA$^+$ can check are conditions on individual executions.

A property TLA$^+$ can’t check

99% of executions produce the right answer.

Here’s a property that TLA+ can’t check

99% of system executions produce the right answer

We use TLA$^+$ to ensure the systems we build “work right”.

Working right means satisfying certain properties.

The properties TLA$^+$ can check are conditions on individual executions.

A property TLA$^+$ can’t check

99% of executions produce the right answer.

Not a condition on a single execution.

Here’s a property that TLA+ can’t check

99% of system executions produce the right answer

This doesn’t express a condition on an individual execution

# THE BASIC ABSTRACTION

## The basic abstraction underlying TLA$^+$

The basic abstraction underlying TLA+ is that

## The basic abstraction underlying TLA$^+$

An execution of a system is represented as a sequence of discrete steps.

The basic abstraction underlying TLA+ is that an execution of a digital system is represented as a sequence of discrete steps.

There are three important ideas there

## The basic abstraction underlying TLA$^+$

An execution of a system is represented as a sequence of discrete steps.

The basic abstraction underlying TLA+ is that  an execution of a digital system is represented as a sequence of discrete steps.

There are three important ideas there  **sequence**

## The basic abstraction underlying TLA$^+$

An execution of a system is represented as a sequence of discrete steps.

The basic abstraction underlying TLA+ is that an execution of a digital system is represented as a sequence of discrete steps.

There are three important ideas there **sequence** **discrete**,

## The basic abstraction underlying TLA$^+$

An execution of a system is represented as a sequence of discrete steps.

The basic abstraction underlying TLA+ is that an execution of a digital system is represented as a sequence of discrete steps.

There are three important ideas there **sequence** **discrete**, and **step**.

An execution of a system is represented as a sequence of discrete steps.

The basic abstraction underlying TLA+ is that an execution of a digital system is represented as a sequence of discrete steps.

There are three important ideas there **sequence** **discrete**, and **step**. Let’s start with **discrete**

An execution of a system is represented as a sequence of discrete steps.

Digital system: we can abstract its continuous evolution as a sequence of discrete events.

The basic abstraction underlying TLA+ is that an execution of a digital system is represented as a sequence of discrete steps.

There are three important ideas there **sequence** **discrete**, and **step**. Let's start with **discrete**

A digital system is a physical system in which we can abstract its continuous evolution as a sequence of discrete events.

An execution of a system is represented as a sequence of discrete steps.

Digital system: we can abstract its continuous evolution as a sequence of discrete events.

The first digital system.

The first self-powered digital system was the escapement clock.

An execution of a system is represented
as a sequence of discrete steps.

Digital system: we can abstract its continuous evolution as a sequence of discrete events.

The first digital system.

A continuous physical system.

The first self-powered digital system was the escapement clock.

Although all the parts of the clock move continuously,

[ slide 134 ]

An execution of a system is represented as a sequence of discrete steps.

Digital system: we can abstract its continuous evolution as a sequence of discrete events.

The first digital system.

A continuous physical system.

Can describe it by an abstract clock advancing in discrete ticks.

The first self-powered digital system was the escapement clock.

Although all the parts of the clock move continuously, we can describe the real clock by an abstract clock that advances in discrete ticks.

An execution of a system is represented
as a sequence of discrete steps.

Digital system: we can abstract its continuous evolution as a sequence of discrete events.

A Computer

The first self-powered digital system was the escapement clock.

Although all the parts of the clock move continuously,  we can describe the real clock by an abstract clock that advances in discrete ticks.

A computer

An execution of a system is represented as a sequence of discrete steps.

Digital system: we can abstract its continuous evolution as a sequence of discrete events.

A Computer

A continuous physical system.

The first self-powered digital system was the escapement clock.

Although all the parts of the clock move continuously, we can describe the real clock by an abstract clock that advances in discrete ticks.

A computer is also a continuously evolving physical system,

An execution of a system is represented as a sequence of discrete steps.

Digital system: we can abstract its continuous evolution as a sequence of discrete events.

A Computer

A continuous physical system.

Can abstract execution of a program as discrete steps.

The first self-powered digital system was the escapement clock.

Although all the parts of the clock move continuously, we can describe the real clock by an abstract clock that advances in discrete ticks.

A computer is also a continuously evolving physical system, designed so we can abstract the execution of a program by the computer as consisting of discrete steps.

[ slide 138 ]

An execution of a system is represented as a sequence of discrete steps.

What about "sequence"?

An execution of a system is represented as a sequence of discrete steps.

Strange to describe a *concurrent* system as a *sequence* of steps.

What about “sequence”?

It may seem strange to describe a *concurrent* system, in which multiple things can be going on at the same time, as a *sequence* of steps.

An execution of a system is represented as a sequence of discrete steps.

Strange to describe a *concurrent* system as a *sequence* of steps.

It’s possible.

What about “sequence”?

It may seem strange to describe a *concurrent* system, in which multiple things can be going on at the same time, as a *sequence* of steps.

We know that it’s possible to do this

An execution of a system is represented
as a sequence of discrete steps.

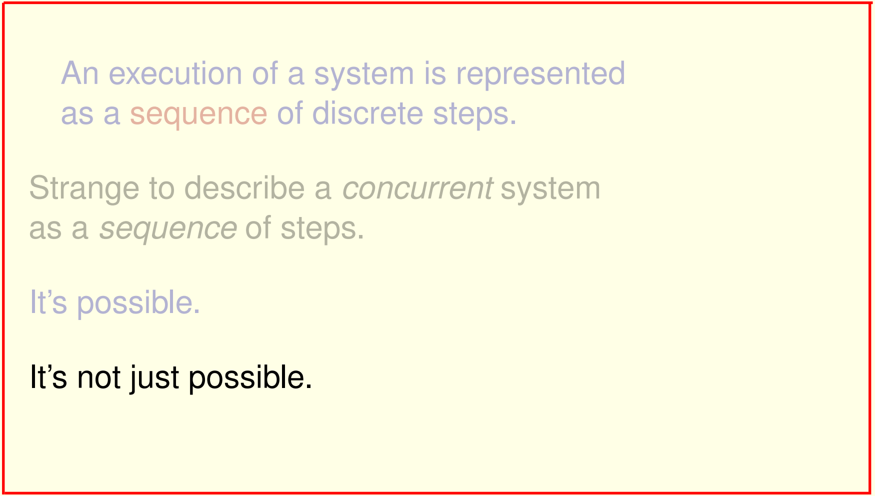
Strange to describe a *concurrent* system
as a *sequence* of steps.

It's possible. We can simulate a concurrent system
with a sequential program.

What about "sequence"?

It may seem strange to describe a *concurrent* system, in which multiple things can be going on at the same time, as a *sequence* of steps.

We know that it's possible to do this because we can simulate a concurrent system with a sequential program.



An execution of a system is represented as a sequence of discrete steps.

Strange to describe a *concurrent* system as a *sequence* of steps.

It's possible.

It's not just possible.

As you learn TLA+, you'll see that it's not just possible.

An execution of a system is represented as a sequence of discrete steps.

Strange to describe a *concurrent* system as a *sequence* of steps.

It's possible.

It's not just possible. It's simple and works well.

As you learn TLA+, you'll see that it's not just possible. It's simple and it works really well.

An execution of a system is represented as a sequence of discrete steps.

What’s a step?

An execution of a system is represented as a sequence of discrete steps.

TLA$^+$ describes a step as a state change.

What’s a step?

TLA+ describes a step as a state change.

An execution of a system is represented as a sequence of discrete steps.

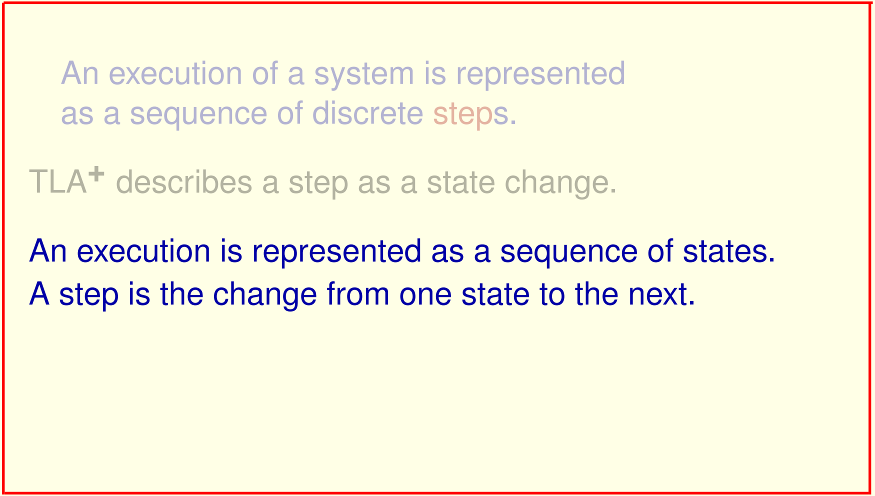
TLA$^+$ describes a step as a state change.

An execution is represented as a sequence of states.

What’s a step?

TLA+ describes a step as a state change.

An execution is represented as a sequence of states,



An execution of a system is represented as a sequence of discrete steps.

TLA$^+$ describes a step as a state change.

An execution is represented as a sequence of states.
A step is the change from one state to the next.

What’s a step?

TLA+ describes a step as a state change.

An execution is represented as a sequence of states,
and a step is the change from one state to the next.

I find this very natural because

An execution of a system is represented
as a sequence of discrete steps.

TLA$^+$ describes a step as a state change.

An execution is represented as a sequence of states.

A step is the change from one state to the next.

Science models systems by a state changing
with time

in most sciences, systems are modeled by a state that changes with time.

An execution of a system is represented as a sequence of discrete steps.

TLA$^+$ describes a step as a state change.

An execution is represented as a sequence of states.

A step is the change from one state to the next.

Science models systems by a state changing with time, usually continuously.

in most sciences, systems are modeled by a state that changes with time. Most often, the state is described as changing continuously.

An execution of a system is represented
as a sequence of discrete steps.

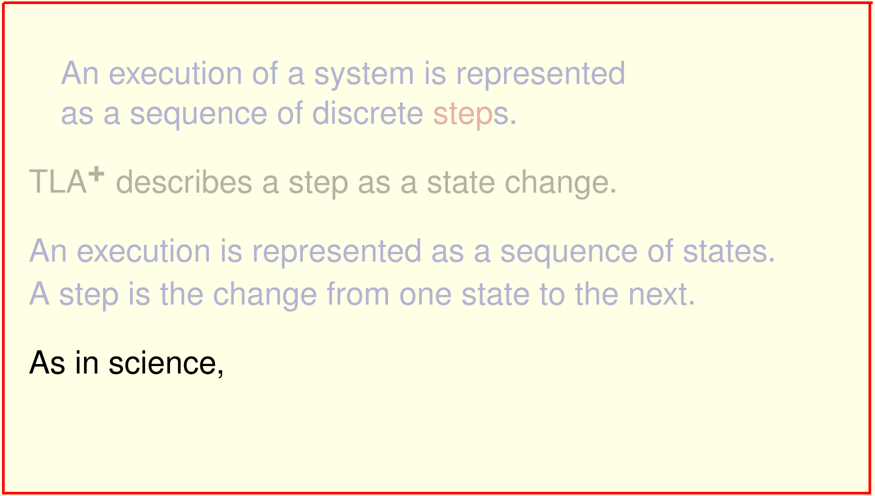
TLA$^+$ describes a step as a state change.

An execution is represented as a sequence of states.
A step is the change from one state to the next.

We model digital systems by a state changing
in discrete steps.

in most sciences, systems are modeled by a state that changes with time. Most often, the state is described as changing continuously.

But we model digital systems by a state that changes in discrete steps.



An execution of a system is represented
as a sequence of discrete steps.

TLA$^+$ describes a step as a state change.

An execution is represented as a sequence of states.
A step is the change from one state to the next.

As in science,

in most sciences, systems are modeled by a state that changes with time. Most often, the state is described as changing continuously.

But we model digital systems by a state that changes in discrete steps.

As in most sciences,

An execution of a system is represented
as a sequence of discrete steps.

TLA$^+$ describes a step as a state change.

An execution is represented as a sequence of states.
A step is the change from one state to the next.

As in science, TLA$^+$ describes a state as an assignment of values to variables.

in most sciences, systems are modeled by a state that changes with time. Most often, the state is described as changing continuously.

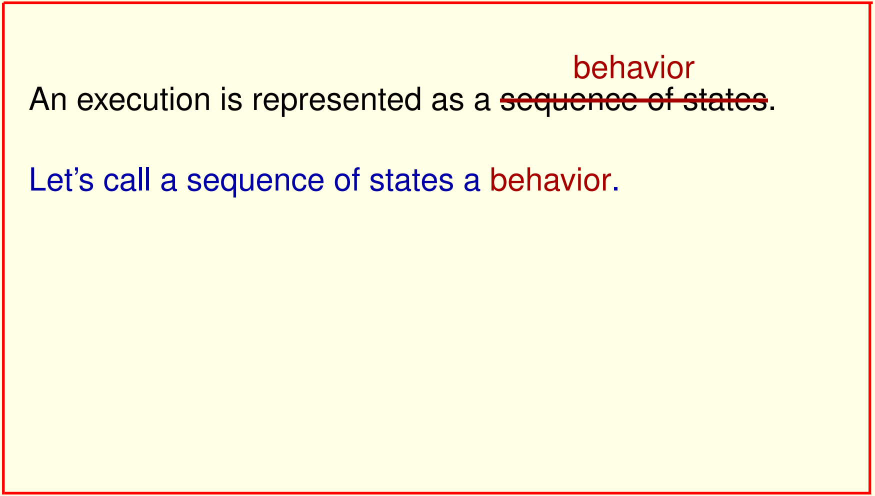
But we model digital systems by a state that changes in discrete steps.

As in most sciences, TLA+ describes a state as an assignment of values to variables.

# STATE MACHINES

An execution is represented as a sequence of states.

An execution of a digital system is represented as a sequence of states.



behavior

An execution is represented as a ~~sequence of states~~.

Let’s call a sequence of states a behavior.

An execution of a digital system is represented as a sequence of states.

Let’s call a sequence of states a behavior,

[ slide 156 ]

An execution is represented as a behavior.

An execution of a digital system is represented as a sequence of states.

Let’s call a sequence of states a behavior,
so an execution of a digital system is represented as a behavior.

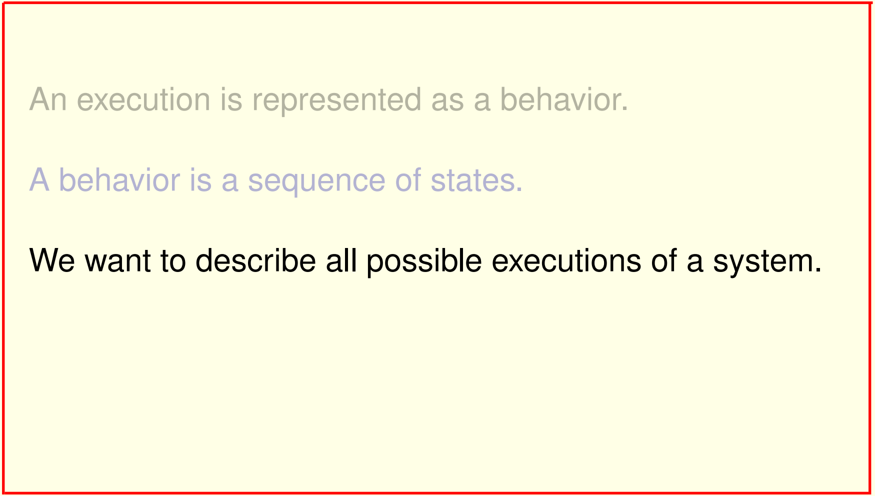
An execution is represented as a behavior.

A behavior is a sequence of states.

An execution of a digital system is represented as a sequence of states.

Let’s call a sequence of states a behavior,
so an execution of a digital system is represented as a behavior.

Where a behavior is a sequence of states.



An execution is represented as a behavior.

A behavior is a sequence of states.

We want to describe all possible executions of a system.

What we want to describe are all the possible executions of a system

An execution is represented as a behavior.

A behavior is a sequence of states.

We want to ~~describe~~ specify all possible ~~executions~~ behaviors of a system.

What we want to describe are all the possible executions of a system

which means specifying all its possible behaviors.

An execution is represented as a behavior.

A behavior is a sequence of states.

We want to specify all possible behaviors of a system.

What we want to describe are all the possible executions of a system

which means specifying all its possible behaviors.

An execution is represented as a behavior.

A behavior is a sequence of states.

We want to specify all possible of a system.

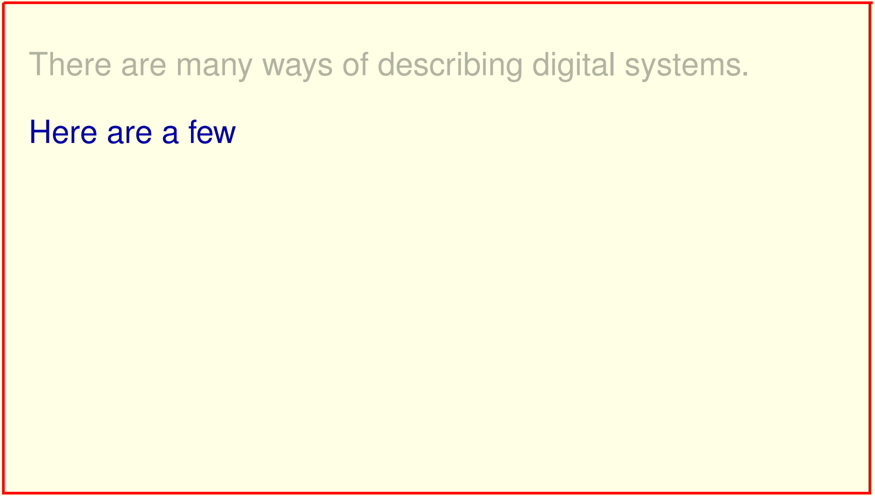
How do we do that?

What we want to describe are all the possible executions of a system which means specifying all its possible behaviors.

How do we do that?

There are many ways of describing digital systems.

People have used many ways of describing digital systems.



There are many ways of describing digital systems.

Here are a few

People have used many ways of describing digital systems.

Here are a few

There are many ways of describing digital systems.

Here are a few

- Programming languages

People have used many ways of describing digital systems.

Here are a few

- Programming languages

There are many ways of describing digital systems.

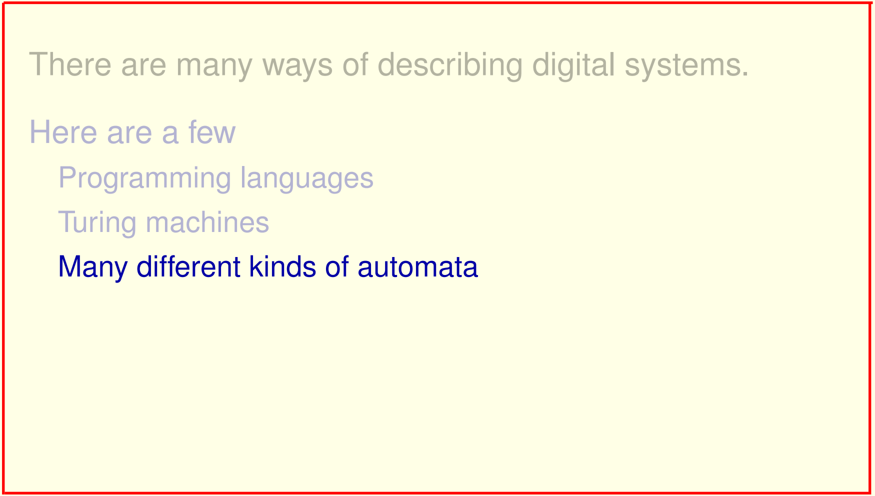
Here are a few

- Programming languages
- Turing machines

People have used many ways of describing digital systems.

Here are a few

- Programming languages
- Turing machines



There are many ways of describing digital systems.

Here are a few

- Programming languages
- Turing machines
- Many different kinds of automata

People have used many ways of describing digital systems.

Here are a few

- Programming languages
- Turing machines
- Lots of different kinds of automata

There are many ways of describing digital systems.

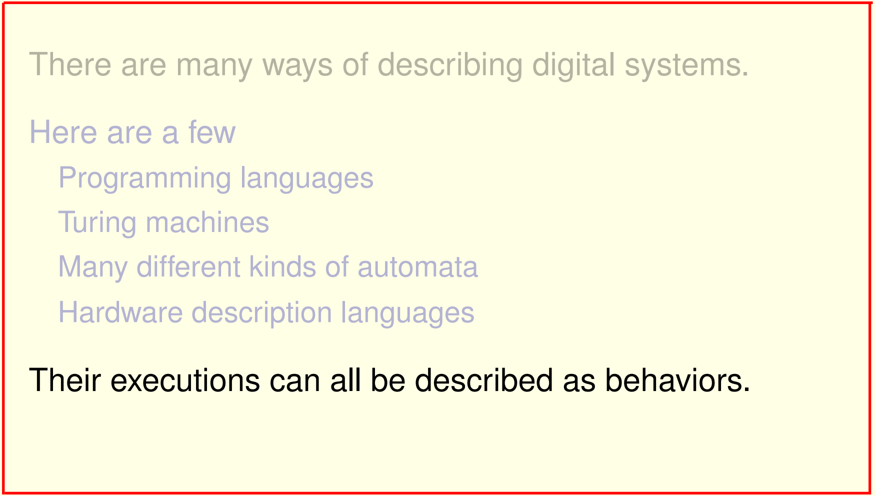
Here are a few

- Programming languages
- Turing machines
- Many different kinds of automata
- Hardware description languages

People have used many ways of describing digital systems.

Here are a few

- Programming languages
- Turing machines
- Lots of different kinds of automata
- Hardware description languages



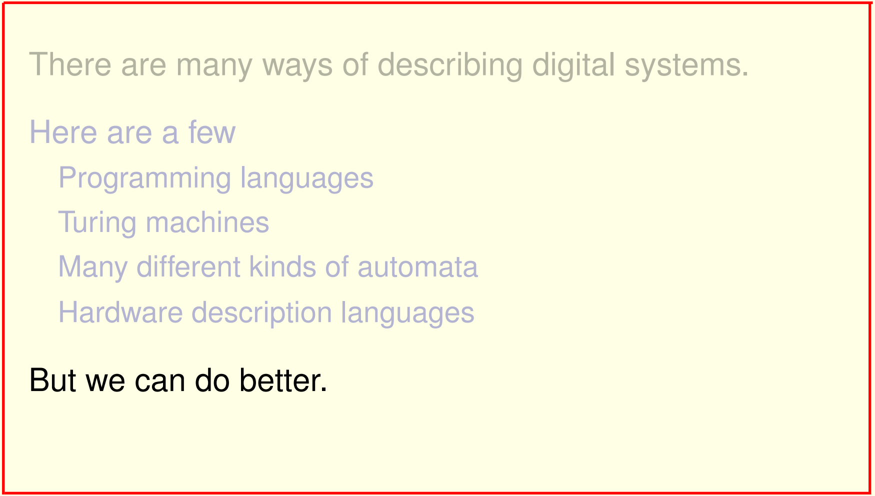
There are many ways of describing digital systems.

Here are a few

- Programming languages
- Turing machines
- Many different kinds of automata
- Hardware description languages

Their executions can all be described as behaviors.

Their executions can all be described as behaviors.



There are many ways of describing digital systems.

Here are a few

- Programming languages
- Turing machines
- Many different kinds of automata
- Hardware description languages

But we can do better.

Their executions can all be described as behaviors.

But we can do better.

[ slide 170 ]

There are many ways of describing digital systems.

Here are a few

- Programming languages
- Turing machines
- Many different kinds of automata
- Hardware description languages

But we can do better.

We can abstract them all as **state machines**.

Their executions can all be described as behaviors.

But we can do better.

We can abstract them all as what I call **state machines**.

A state machine is described by:

A state machine is described by two things:

A state machine is described by:

1. All possible initial states.

A state machine is described by two things:

All its possible initial states.

A state machine is described by:

1. All possible initial states.
2. What next states can follow any given state.

A state machine is described by two things:

All its possible initial states.

And what the possible next states are that can follow any given state.

A state machine is described by:

1. All possible initial states.
2. What next states can follow any given state.

It halts if there is no possible next state.

The state machine halts if it reaches a state for which there is no possible next state.

1. All possible initial states.
2. What next states can follow any given state.

A state is an assignment of values to variables

The state machine halts if it reaches a state for which there is no possible next state.

A state is an assignment of values to variables,

1. All possible initial states.
2. What next states can follow any given state.

A state is an assignment of values to variables,
so a state machine is described by:

The state machine halts if it reaches a state for which there is no possible next state.

A state is an assignment of values to variables,
so to describe a state machine we must describe three things:

1. All possible initial states.
2. What next states can follow any given state.

A state is an assignment of values to variables,
so a state machine is described by:

0. What the variables are.

What the variables are.

1. All possible initial states.

2. What next states can follow any given state.

A state is an assignment of values to variables,
so a state machine is described by:

0. What the variables are.

1. Possible initial values of variables.

What the variables are.

What the possible initial values of the variables are.

1. All possible initial states.
2. What next states can follow any given state.

A state is an assignment of values to variables,
so a state machine is described by:

0. What the variables are.
1. Possible initial values of variables.
2. A relation between their values in the current state

What the variables are.

What the possible initial values of the variables are.

And what the relation is between the values of the variables in the current state

1. All possible initial states.
2. What next states can follow any given state.

A state is an assignment of values to variables,
so a state machine is described by:

0. What the variables are.
1. Possible initial values of variables.
2. A relation between their values in the current state and their possible values in the next state.

What the variables are.

What the possible initial values of the variables are.

And what the relation is between the values of the variables in the current state and the their possible values in the next state.

# A TINY EXAMPLE

# A C program

The example is a simple C program.

```
int i ;
void main()
    { i = someNumber() ;
      i = i + 1 ;
    }
```

The example is a simple C program. It’s not quite legal C because

```
int i ;
void main()
    { i = someNumber() ;
      i = i + 1 ;
    }
```

The example is a simple C program. It’s not quite legal C because ANSI C requires *main* to return an int.

```
int i ;
void main()
    { i = someNumber() ;
      i = i + 1 ;
    }
```

The example is a simple C program. It's not quite legal C because ANSI C requires *main* to return an int. But it's obvious what this program should do

```
int i ;
void main()
    { i = someNumber() ;
      i = i + 1 ;
    }
```

The example is a simple C program. It's not quite legal C because ANSI C requires *main* to return an int. But it's obvious what this program should do except for the function $someNumber$.

```
int i ;
void main()
    { i = someNumber() ;
      i = i + 1 ;
    }
```

$someNumber()$ returns a number from 0 to 1000.

The example is a simple C program. It's not quite legal C because ANSI C requires *main* to return an int. But it's obvious what this program should do except for the function $someNumber$.

Let's assume that a call to $someNumber$ returns an arbitrarily chosen number from 0 to 1000.

```
int i ;
void main()
   { i = someNumber() ;
     i = i + 1 ;
   }
```

*someNumber*() returns a number from 0 to 1000.

Possibly different numbers in different executions.

The example is a simple C program. It's not quite legal C because ANSI C requires *main* to return an int. But it's obvious what this program should do except for the function *someNumber*.

Let's assume that a call to *someNumber* returns an arbitrarily chosen number from 0 to 1000.

And that it can return different numbers in different executions. Let's not worry about how *someNumber* is implemented in C.

```
int i ;
void main()
    { i = someNumber() ;
      i = i + 1 ;
    }
```

Obvious representation: a single variable $i$.

An obvious way to represent this program as a state machine is with a single variable $i$.

```
int i ;
void main()
    { i = someNumber() ;
      i = i + 1 ;
    }
```

A possible execution:

An obvious way to represent this program as a state machine is with a single variable $i$.

Let's look at one possible execution

```
int i ;
void main()
    { i = someNumber() ;
      i = i + 1 ;
    }
```

A possible execution:

[$i$ : 0]

An obvious way to represent this program as a state machine is with a single variable $i$.

Let's look at one possible execution

Here's the initial state.

```
int i ;
void main()
    { i = someNumber() ;
      i = i + 1 ;
    }
```

A possible execution:

$[i : 0]$   A state in which $i$ has value 0.

An obvious way to represent this program as a state machine is with a single variable $i$.

Let’s look at one possible execution

Here’s the initial state.   It’s a state that assigns the value 0 to $i$.

```
int i ;            C initializes i to 0.
void main()
   { i = someNumber() ;
     i = i + 1 ;
   }
```

A possible execution:

$[i : 0]$

An obvious way to represent this program as a state machine is with a single variable $i$.

Let's look at one possible execution

Here's the initial state.   It's a state that assigns the value 0 to $i$.

It's the only possible initial state because in C, this declaration initializes $i$ to 0.

```
int $i$ ;
void $main()$
    { $i = someNumber()$ ;    Returns 42.
      $i = i + 1$ ;
    }
```

A possible execution:

$$[i : 0] \;\rightarrow\; [i : 42]$$

In the second state $i$ equals 42 because the call of $someNumber$ happened to return 42

```
int i ;
void main()
    { i = someNumber() ;
      i = i + 1 ;
    }
```

A possible execution:

$$[i : 0] \;\rightarrow\; [i : 42] \;\rightarrow\; [i : 43]$$

In the second state $i$ equals 42 because the call of $someNumber$ happened to return 42

Execution of the third statement increments $i$, producing the third state, in which $i$ equals 43.

```
int i ;
void main()
   { i = someNumber() ;
     i = i + 1 ;
   }
```

A possible execution:

$$[i : 0] \rightarrow [i : 42] \rightarrow [i : 43]$$

OK

```
int i ;
void main()
    { i = someNumber() ;
      i = i + 1 ;
    }
```

A possible execution:

$$[i : 0] \rightarrow [i : 42] \rightarrow [i : 43]$$

How do we describe this as a state machine?

OK How do we describe this tiny program as a state machine?

```
int i ;
void main()
  { i = someNumber() ;
    i = i + 1 ;
  }
```

A possible execution:

$[i : 0] \rightarrow [i : 42] \rightarrow [i : 43]$

How do we describe this as a state machine?

0. What the variables are.
1. Possible initial values of variables.
2. The relation between their values in the current state and their possible values in the next state.

OK How do we describe this tiny program as a state machine?

Remember, we must describe these three things: The variables, their initial values, and the relation between their values in the current state and their possible values in the next state.

```
int i ;
void main()
    { i = someNumber() ;
      i = i + 1 ;
    }
```

A possible execution:

$[i : 0] \rightarrow [i : 42] \rightarrow [i : 43]$

0. The variables: $i$

OK How do we describe this tiny program as a state machine?

Remember, we must describe these three things: The variables, their initial values, and the relation between their values in the current state and their possible values in the next state.

We're representing this program with the single variable $i$.

```
int i ;
void main()
   { i = someNumber() ;
     i = i + 1 ;
   }
```

A possible execution:

$[i : 0] \rightarrow [i : 42] \rightarrow [i : 43]$

0. The variables: $i$

1. Initial values: $i = 0$

OK  How do we describe this tiny program as a state machine?

Remember, we must describe these three things: The variables, their initial values, and the relation between their values in the current state and their possible values in the next state.

We're representing this program with the single variable $i$.

There's only one possible initial value of $i$: namely, 0.

```
int i ;
void main()
    { i = someNumber() ;
      i = i + 1 ;
    }
```

A possible execution:

$$[i : 0] \rightarrow [i : 42] \rightarrow [i : 43]$$

0. The variables: $i$

1. Initial values: $i = 0$

2. The relation between the value of $i$ in the current state and its possible values in the next state.

What about 2, the relation between the current value of $i$ and its possible values in the next state?

```
int i ;
void main()
    { i = someNumber() ;
      i = i + 1 ;
    }
```

A possible execution:

$[i : 0] \rightarrow [i : 42] \rightarrow [i : 43]$

Current: i = 43, no next value

0. The variables: $i$

1. Initial values: $i = 0$

2. The relation between the value of $i$ in the current state and its possible values in the next state.

What about 2, the relation between the current value of $i$ and its possible values in the next state?

Let’s look at $i$’s possible next value when its current value is 43. In this behavior, there is no possible next value because the program has terminated.

[ slide 203 ]

```
int i ;
void main()
    { i = someNumber() ;
      i = i + 1 ;
    }
```

Another possible execution:

$[i : 0] \rightarrow [i : 43] \rightarrow [i : 44]$

0. The variables: $i$
1. Initial values: $i = 0$
2. The relation between the value of $i$ in the current state and its possible values in the next state.

Let's look at another possible execution, in which the call of $someNumber$ returned the number 43.

```
int i ;
void main()
   { i = someNumber() ;
     i = i + 1 ;
   }
```

Another possible execution:

$[i : 0] \rightarrow [i : 43] \rightarrow [i : 44]$

Current: i = 43,  next: i = 44

0. The variables: $i$

1. Initial values: $i = 0$

2. The relation between the value of $i$ in the current state and its possible values in the next state.

Let's look at another possible execution, in which the call of $someNumber$ returned the number 43.

In this execution, when the current value of $i$ is 43 its next value must be 44.

```
int i ;
void main()
    { i = someNumber() ;
      i = i + 1 ;
    }
```

A possible execution:

$$[i : 0] \rightarrow [i : 42] \rightarrow [i : 43]$$

Current: i = 43, no next value

0. The variables: $i$

1. Initial values: $i = 0$

2. The relation between the value of $i$ in the current state and its possible values in the next state.

Let's look at another possible execution, in which the call of $someNumber$ returned the number 43.

In this execution, when the current value of $i$ is 43 its next value must be 44.

This execution

```
int i ;
void main()
    { i = someNumber() ;
      i = i + 1 ;
    }
```

Another possible execution:

$$[i : 0] \rightarrow [i : 43] \rightarrow [i : 44]$$

Current: i = 43,  next: i = 44

0. The variables: $i$

1. Initial values: $i = 0$

2. The relation between the value of $i$ in the current state and its possible values in the next state.

Let's look at another possible execution, in which the call of $someNumber$ returned the number 43.

In this execution, when the current value of $i$ is 43 its next value must be 44.

This execution  and this execution require different next values of $i$ for the same current value 43.

```
int i ;
void main()
    { i = someNumber() ;
      i = i + 1 ;
    }
```

Another possible execution:

$$[i : 0] \rightarrow [i : 43] \rightarrow [i : 44]$$

Current: i = 43,  next: i = 44

0. The variables: $i$

1. Initial values: $i = 0$

2. The relation between the value of $i$ in the current state and its possible values in the next state.

**IMPOSSIBLE**

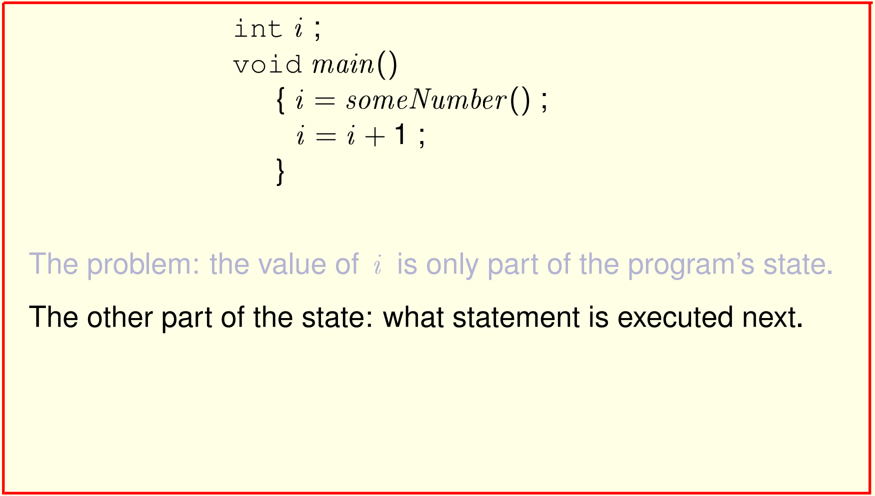
So it’s impossible to represent the program as a state machine in this way.

```
int i ;
void main()
  { i = someNumber() ;
    i = i + 1 ;
  }
```

```
int i ;
void main()
   { i = someNumber() ;
     i = i + 1 ;
   }
```

The problem: the value of $i$ is only part of the program's state.

The problem is that the value of $i$ is only part of the program's state.



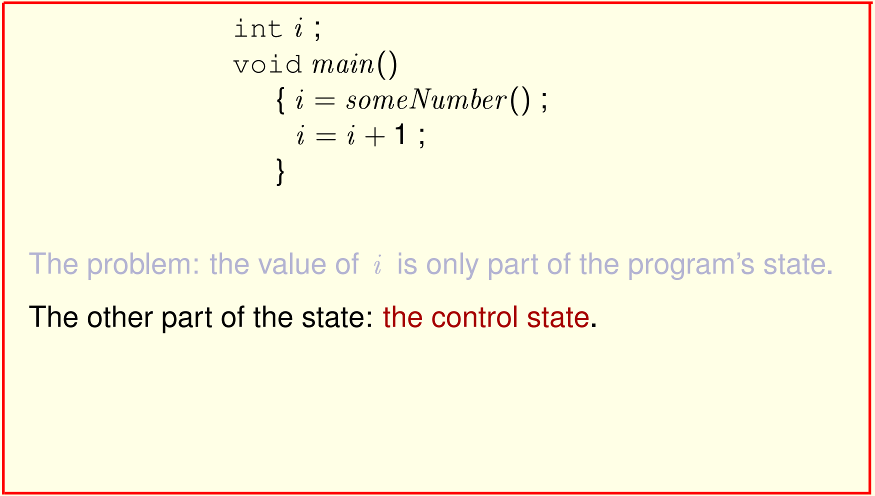
```
int i ;
void main()
   { i = someNumber() ;
     i = i + 1 ;
   }
```

The problem: the value of $i$ is only part of the program’s state.

The other part of the state: what statement is executed next.

The problem is that the value of $i$ is only part of the program’s state.

There’s another part of the state that specifies what statement is to be executed next.



```
int i ;
void main()
   { i = someNumber() ;
     i = i + 1 ;
   }
```

The problem: the value of $i$ is only part of the program's state.

The other part of the state: the control state.

The problem is that the value of $i$ is only part of the program's state.

There's another part of the state that specifies what statement is to be executed next.

That part of the state is called the control state.

[ slide 212 ]

```
int i ;
void main()
   { i = someNumber() ;
     i = i + 1 ;
   }
```

The problem: the value of $i$ is only part of the program's state.

The other part of the state: the control state.

0. The variables: $i$ , $pc$

The problem is that the value of $i$ is only part of the program's state.

There's another part of the state that specifies what statement is to be executed next.

That part of the state is called the control state.

So let's introduce another variable called $pc$ (for program control) to represent the control state.

```
int i ;
void main()
   { i = someNumber() ;
     i = i + 1 ;
   }
```

The problem: the value of $i$ is only part of the program's state.

The other part of the state: the control state.

0. The variables: $i$ , $pc$

The problem is that the value of $i$ is only part of the program's state.

There's another part of the state that specifies what statement is to be executed next.

That part of the state is called the control state.

So let's introduce another variable called $pc$ (for program control) to represent the control state.

```
                 int $i$ ;
                 void $main()$
$pc = $ “$start$”    { $\boxed{i = someNumber()\ ;}$
                    $i = i + 1$ ;
                    }
```

The problem: the value of $i$ is only part of the program’s state.

The other part of the state: the control state.

0. The variables: $i$ , $pc$

When $pc$ equals the string “$start$”, it means that the first assignment statement is to be executed next.

```
               int i ;
               void main()
                  { i = someNumber() ;
pc = "middle"       i = i + 1 ;
                  }
```

The problem: the value of $i$ is only part of the program’s state.

The other part of the state: the control state.

0. The variables: $i$ , $pc$

When $pc$ equals the string “$start$”, it means that the first assignment statement is to be executed next.

$pc$ equal to the string “$middle$” means that the second assignment statement is to be executed next.

```
int i ;
void main()
    { i = someNumber() ;
      i = i + 1 ;
    }  pc = "done"
```

The problem: the value of $i$ is only part of the program's state.

The other part of the state: the control state.

0. The variables: $i$ , $pc$

When $pc$ equals the string "$start$", it means that the first assignment statement is to be executed next.

$pc$ equal to the string "$middle$" means that the second assignment statement is to be executed next.

And $pc$ equals "$done$" means that the program has finished, and there is nothing left to execute.

$$\begin{array}{rl}
 & \texttt{int}\ i\ ; \\
 & \texttt{void}\ main() \\
pc = \text{“}start\text{”} & \quad \{\ \boxed{i = someNumber()\ ;} \\
 & \qquad i = i + 1\ ; \\
 & \quad \}
\end{array}$$

The problem: the value of $i$ is only part of the program’s state.

The other part of the state: the control state.

0. The variables: $i$ , $pc$

1. Initial values: $i = 0$ and $pc = \text{“}start\text{”}$

The initial value of $i$ is 0, and the initial value of $pc$ is the string “$start$”.

```
int i ;
void main()
   { i = someNumber() ;
     i = i + 1 ;
   }
```

2. The relation between the current values of $i$ and $pc$ and their possible next values.

The initial value of $i$ is 0, and the initial value of $pc$ is the string "$start$".

And now we can describe the relation between the current values of $i$ and $pc$ and their possible next values.

```
int i ;
void main()
   { i = someNumber() ;
     i = i + 1 ;
   }
```

**if** current value of $pc$ equals "$start$"
  **then**

The initial value of $i$ is 0, and the initial value of $pc$ is the string "$start$".

And now we can describe the relation between the current values of $i$ and $pc$ and their possible next values.

If the current value of $pc$ equals "$start$", then

```
int i ;
void main()
   { i = someNumber() ;
     i = i + 1 ;
   }
```

```
if current value of $pc$ equals "$start$"
   then next value of $i$ in $\{0, 1, \ldots, 1000\}$
```

The initial value of $i$ is 0, and the initial value of $pc$ is the string "$start$".

And now we can describe the relation between the current values of $i$ and $pc$ and their possible next values.

If the current value of $pc$ equals "$start$", then

the next value of $i$ can be any number in the set of integers from 0 through 1000

[ slide 221 ]

```
int i ;
void main()
    { i = someNumber() ;
      i = i + 1 ;
    }
```

```
if current value of $pc$ equals “$start$”
    then next value of $i$ in $\{0, 1, \ldots, 1000\}$
         next value of $pc$ equals “$middle$”
```

and the next value of $pc$ is the string “$middle$”.

```
int i ;
void main()
   { i = someNumber() ;
     i = i + 1 ;
   }
```

```
if current value of $pc$ equals "$start$"
    then next value of $i$ in $\{0, 1, \ldots, 1000\}$
         next value of $pc$ equals "$middle$"
    else if current value of $pc$ equals "$middle$"
           then
```

and the next value of $pc$ is the string "$middle$".

If not, if the current value of $pc$ equals "$middle$", then

```
int i ;
void main()
   { i = someNumber() ;
     i = i + 1 ;
   }
```

```
if current value of $pc$ equals “$start$”
   then next value of $i$ in $\{0, 1, \ldots, 1000\}$
        next value of $pc$ equals “$middle$”
   else if current value of $pc$ equals “$middle$”
           then next value of $i$ equals current value of $i + 1$
```

and the next value of $pc$ is the string “$middle$”.

If not, if the current value of $pc$ equals “$middle$”, then

the next value of $i$ equals the current value of $i$ plus one

```
int i ;
void main()
   { i = someNumber() ;
     i = i + 1 ;
   }
```

```
if current value of $pc$ equals “$start$”
   then next value of $i$ in $\{0, 1, \ldots, 1000\}$
        next value of $pc$ equals “$middle$”
   else if current value of $pc$ equals “$middle$”
           then next value of $i$ equals current value of $i + 1$
                next value of $pc$ equals “$done$”
```

and the next value of $pc$ is the string “$middle$”.

If not, if the current value of $pc$ equals “$middle$”, then

the next value of $i$ equals the current value of $i$ plus one

and the next value of $pc$ is the string “$done$”.
Otherwise, the only remaining possibility is that $pc$ equals “$done$”

```
int i ;
void main()
   { i = someNumber() ;
     i = i + 1 ;
   }
```

```
if current value of $pc$ equals “$start$”
    then next value of $i$ in $\{0, 1, \ldots, 1000\}$
         next value of $pc$ equals “$middle$”
    else if current value of $pc$ equals “$middle$”
           then next value of $i$ equals current value of $i + 1$
                next value of $pc$ equals “$done$”
           else no next values
```

and the program has finished, so there are no next values of $i$ and $pc$.

```
if current value of $pc$ equals “$start$”
    then next value of $i$ in $\{0, 1, \ldots, 1000\}$
         next value of $pc$ equals “$middle$”
    else if current value of $pc$ equals “$middle$”
            then next value of $i$ equals current value of $i + 1$
                 next value of $pc$ equals “$done$”
            else no next values
```

OK.

```
if current value of $pc$ equals "$start$"
  then next value of $i$ in $\{0, 1, \ldots, 1000\}$
       next value of $pc$ equals "$middle$"
  else if current value of $pc$ equals "$middle$"
          then next value of $i$ equals current value of $i + 1$
               next value of $pc$ equals "$done$"
          else no next values
```

Not very easy to read.

OK. This isn't very easy to read.

```
**if** current value of $pc$ equals "$start$"
    **then** next value of $i$ in $\{0, 1, \ldots, 1000\}$
             next value of $pc$ equals "$middle$"
    **else** **if** current value of $pc$ equals "$middle$"
                 **then** next value of $i$ equals current value of $i + 1$
                          next value of $pc$ equals "$done$"
                 **else** no next values
```

It's simpler and more elegant in TLA$^+$

OK. This isn't very easy to read.

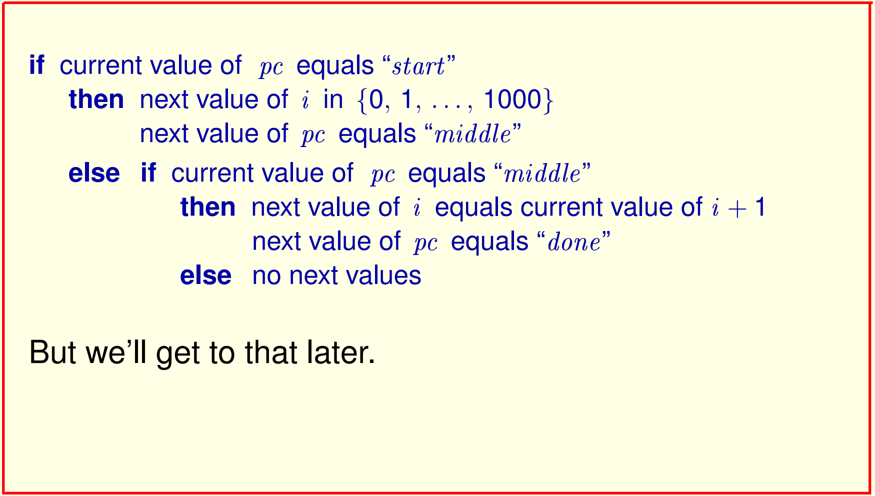
It's simpler and more elegant in TLA+,

```
if current value of $pc$ equals "$start$"
    then next value of $i$ in $\{0, 1, \ldots, 1000\}$
         next value of $pc$ equals "$middle$"
    else if current value of $pc$ equals "$middle$"
            then next value of $i$ equals current value of $i + 1$
                 next value of $pc$ equals "$done$"
            else no next values
```

It’s simpler and more elegant in TLA$^+$, because it’s written as a mathematical formula.

OK. This isn’t very easy to read.

It’s simpler and more elegant in TLA+, because it’s written as a mathematical formula.



```
if current value of $pc$ equals “$start$”
    then next value of $i$ in $\{0, 1, \ldots, 1000\}$
         next value of $pc$ equals “$middle$”
    else if current value of $pc$ equals “$middle$”
            then next value of $i$ equals current value of $i + 1$
                 next value of $pc$ equals “$done$”
            else no next values
```

But we’ll get to that later.

OK. This isn’t very easy to read.

It’s simpler and more elegant in TLA+, because it’s written as a mathematical formula.

But we’ll get to that later.

## State machines are simpler than programs.

State machines are much simpler than programs.

State machines are simpler than programs.

In a state machine, all parts of the state are represented as values of variables.

State machines are much simpler than programs.

In a state machine, all parts of the state are represented the same way: as the values of variables.

In programs, different parts of the state are represented differently:

In programs, different parts of the state are represented differently.

In programs, different parts of the state are represented differently:

The values of variables.

There are the values of the variables.

In programs, different parts of the state are represented differently:

The values of variables.

The control state.

There are the values of the variables. There's the control state.

In programs, different parts of the state are represented differently:

- The values of variables.
- The control state.
- The call stack.

There are the values of the variables. There's the control state. There's the call stack.

In programs, different parts of the state are represented differently:

- The values of variables.
- The control state.
- The call stack.
- The heap.

There are the values of the variables. There's the control state. There's the call stack. There's the heap. Etcetera. Etcetera.

In programs, different parts of the state are represented differently.

They’re represented differently because they’re implemented differently.

Those different parts of the state are represented differently in programs because they’re implemented differently.

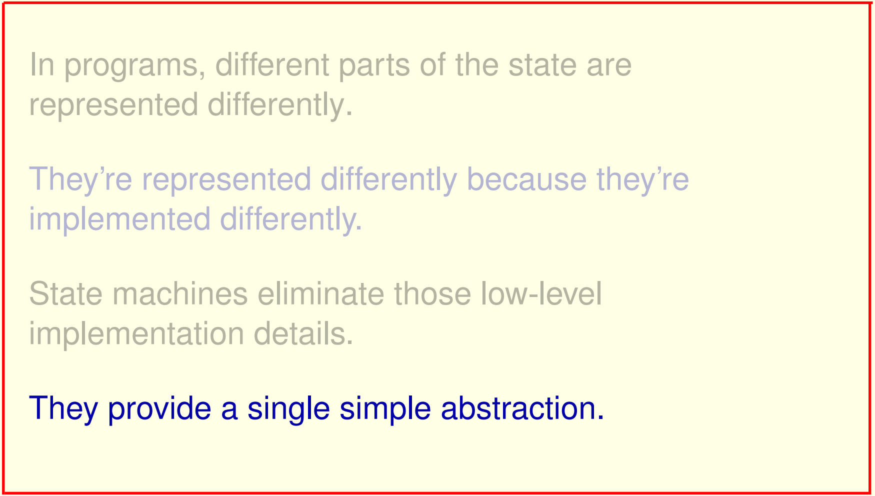
In programs, different parts of the state are represented differently.

They’re represented differently because they’re implemented differently.

State machines eliminate those low-level implementation details.

Those different parts of the state are represented differently in programs because they’re implemented differently.

State machines eliminate those low-level implementation details.



In programs, different parts of the state are represented differently.

They’re represented differently because they’re implemented differently.

State machines eliminate those low-level implementation details.

They provide a single simple abstraction.

Those different parts of the state are represented differently in programs because they’re implemented differently.

State machines eliminate those low-level implementation details.

State machines provide a single simple abstraction.

TLA$^+$ is an elegant, expressive language for describing state machines.

TLA$^+$ is an elegant, extremely expressive language for describing state machines.

**TLA$^+$ Video Course**

# End of Lecture 1

# INTRODUCTION TO TLA$^+$

This is the end of Lecture 1 of the TLA$^+$ Video Course

—

Introduction to TLA+.