Embedded Systems Specification and Design
Model-based Design and Verification

David Kendall

Introduction

- Why do we need better approaches to specification, design and verification?
- Why do embedded systems provide a challenging problem area for testing our ideas?
- What do we mean by specification, design and verification?
- A recap of some simple maths
- A first formal specification
- What are the topics covered in this module?
- How is the module assessed?
- What learning resources are available?

Examples of software and hardware failures

Some failures

- Intel Pentium Floating Point Division
- Therac-25 Radiotherapy Machine
- London Ambulance Service Computer-Aided Dispatch System
- Ariane V
- Toyota Unintended Acceleration
- Nissan Airbag Deployment Failure
- Spectre and Meltdown
- TSB account migration problems

Consequences

- Loss of life
- Loss of billions of pounds
- Loss of reputation

Lessons of failure

Causes of failure are various and complex

- System failures - not just software
- Poor project management
- Poor specification, design and verification
- Poor implementation practices
- Inadequate testing

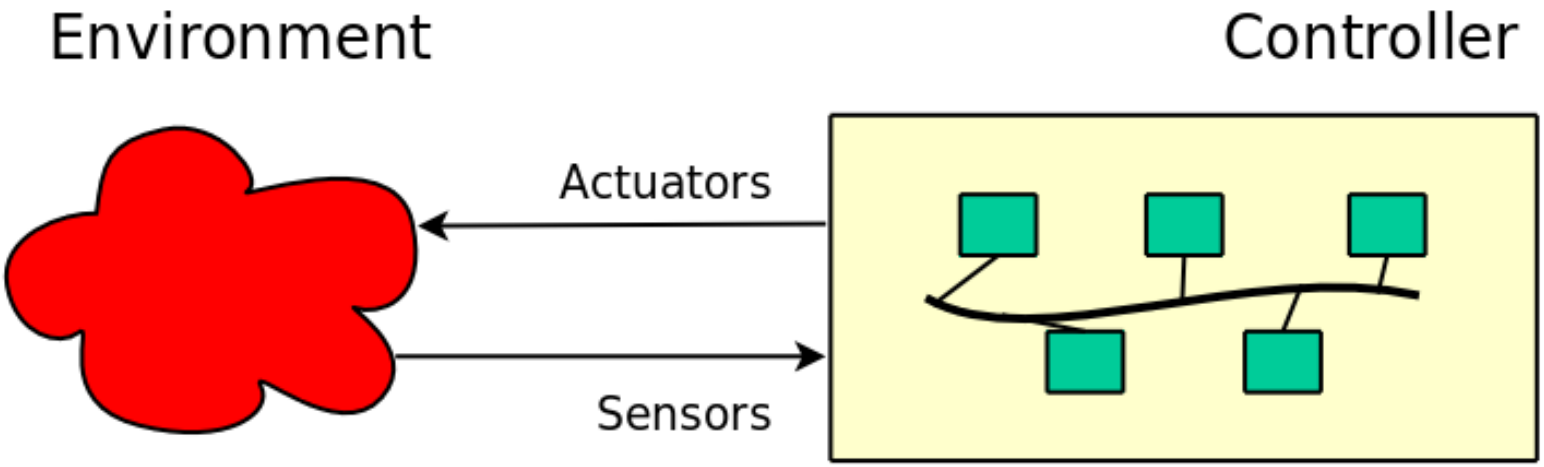
An aim of this module is to introduce you to better approaches to specification, design and verification.

We hope that a consequence of pursuing this aim will be to improve the way that you *think* about system development

Why study embedded systems?

Computing system *embedded* in physical environment for purpose of monitoring and/or control

Other titles: *Cyber-physical systems*, *Internet of Things*



- *concurrent* – composed of *multi-tasking* and/or *distributed* processes
- *communicating* processes cooperate to perform some specific task
- *real-time* – timing requirements imposed by the environment
- *resource-constrained* – limited resources: processing, memory, peripherals, power, . . .

Software and Systems Development

A typical development lifecycle for an embedded system includes everything needed for a data processing system. . . and more besides

- Requirements specification
  - Physical system modelling
  - Control law design
  - Digital system requirements
- Digital System Design and Verification
  - Model-Driven Design (MDD)
  - Verification
- Implementation and testing

Model-Based Design

All other engineering disciplines have model-building at the core of their design processes

- Bridge
- Aeroplane wing
- Jet engine
- Building
- …

Apply this approach to the engineering of digital systems too

Design by creating models of (important aspects of) the system to be developed

- system architecture
- synchronization
- communication protocols
- real-time behaviour

Model-Based Design (ctd)

Analyse the models to check if they have desired properties

- *bug hunting*
- *verification*

Iterate

- Refine models
- Further analysis

Agile development of system models

Modelling digital systems

A digital system is a system whose behaviour consists of *discrete events*

Abstraction is the most important tool that we have for thinking about systems

- What features of the system can we “take away” (abstract) from its representation?
- What features should its representation include?
- The more we “take away”, the simpler but less accurate the representation becomes
- We want the simplest representation that is accurate enough

A model is a representation of an abstraction of a system

Modelling digital systems

There are several ways of modelling systems.

In this module, we focus on an approach in which systems are modelled as *state transition* systems.

State transition system

An abstract system is is described as a collection of behaviours, each representing a possible execution of the system, where a behaviour is a sequence of states, and a state is an assignment of values to variables.

Event, step

An event, also called a *step*, is the transition from one state to the next state in a behaviour.

Representing and reasoning about models

Formal methods

- Applied discrete mathematics: set theory, logic
- Languages, analysis techniques, tools
- Applicable to development of computer hardware and software
- Useful for
  - Description: e.g., specifying requirements
    precise, concise, unambiguous
  - Analysis: e.g., demonstrating program function implements design
- Rigour of mathematics helps to develop convincing argument using calculation
- Reduces reliance on human intuition and judgement

The essence of a formal method

Informally

The system satisfies its specification

Formally

$Sys \models Spec$

We use *formal languages* for the system model *Sys* and the specification *Spec*

A formal language has well-defined:

- syntax,
- semantics, and
- rules for reasoning about relationship between expressions

Some current formal methods and tools

Model-based

- Abstract State Machines
- Alloy
- B and Event-B
- PRISM
- Promela / SPIN
- SMV
- TLA$^+$
- UPPAAL
- VDM
- Z

Logics

- First order predicate logic
- Modal logic
- Temporal Logic - LTL, CTL, PCTL
- Higher-order logic

Algebras and calculi

- Maude,
- CCS, π-calculus, Bigraphs
- CSP
- LOTOS

How to choose where to start?

Considerations for choice of first formal method

Based on simple discrete mathematics

- First order predicate logic
- Set theory

Computer-based tool support

- Syntax checking
- Checking semantic relationships
- Bug finding
- Simulation

Applicable to industrial-scale systems

- Evidence of industrial use

Good “return on investment”

- Easy to learn
- Able to model and reason about a wide variety of systems and properties
- Knowledge and understanding gained should be easily transferable to a variety of other formal methods

Model Checking

Computer-based tool support

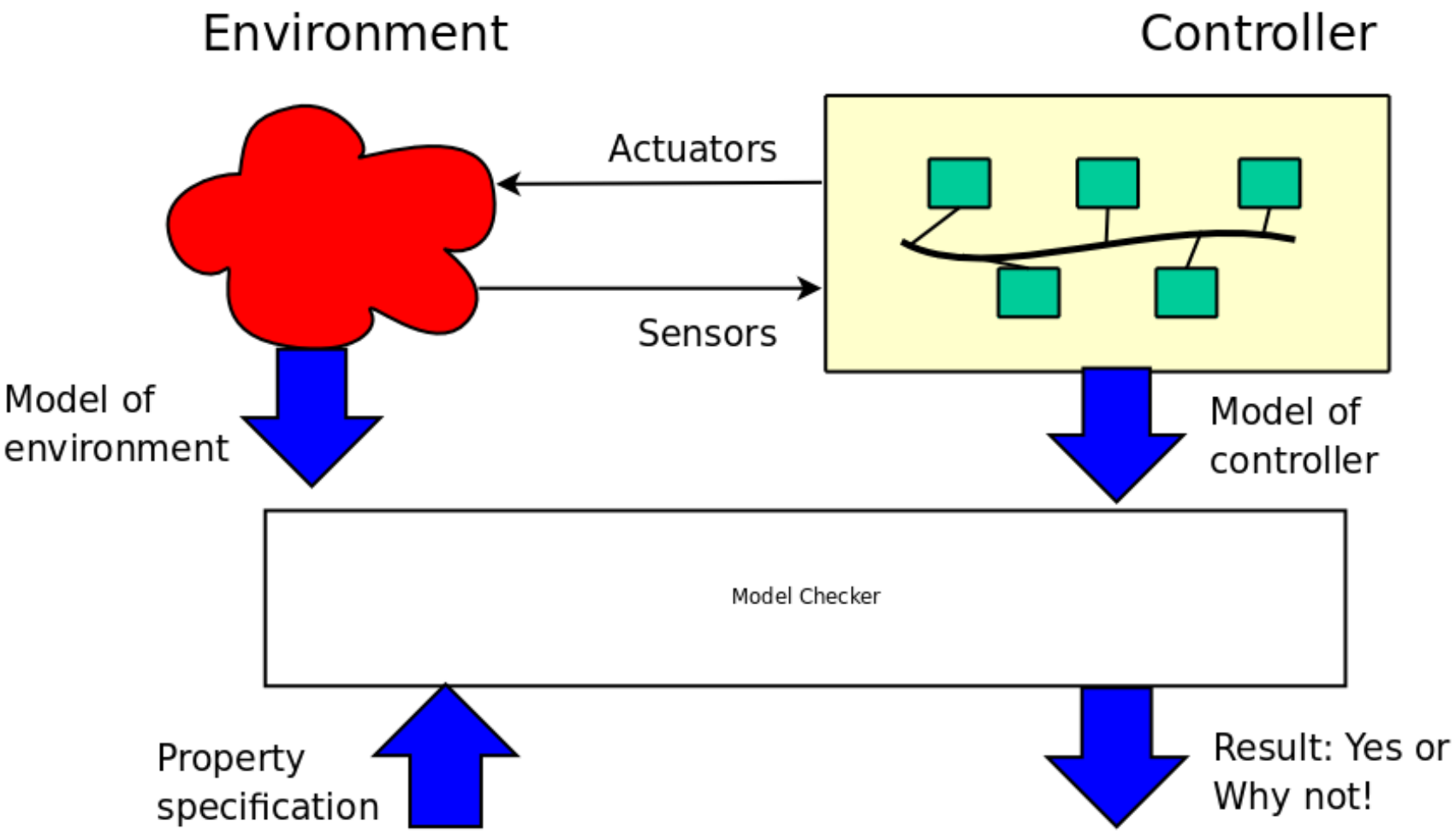
A key element in making formal methods accessible to digital systems engineers

Pragmatic engineering approach to use of formal methods in systems design

Supports the essential features of a formal method

- Sys – state transition model
- Spec – propositions of first-order and temporal logic
- $\models$ – established by model checking — 'press the button'

The model-checking approach



Formal Methods in Industry

Largely thanks to advances in model-checking, major organisations are tackling the specification, design and verification problem using *formal methods*

- Amazon,
- Microsoft,
- Intel,
- Google,
- ARM,
- NASA

See David Langworthy's talk for some examples of the use of TLA$^+$ at Microsoft.

Short interlude...

Name 3 winners of the Nobel Prize in Computer Science

There is no Nobel Prize for Computer Science!

The equivalent award is the *Turing Award*

- Awarded annually since 1966
- Currently comes with a prize of $1,000,000

Let's look at some Turing Award winners...

Temporal Logic

Amir Pnueli
Turing Award Winner 1996

A. Pnueli, *The Temporal Logic of Programs*, 18th Annual Symposium on Foundations of Computer Science, pp. 46–57, 1977

Model-checking

Ed Clarke

E. Allen Emerson

Joseph Sifakis

Turing Award Winners 2007

Edmund M. Clarke, E. Allen Emerson, *Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic*, Proceedings of Workshop on Logic of Programs, pp. 52-71, 1981.

J.P. Queille and J. Sifakis, *Specification and verification of concurrent systems in CESAR*, Proceedings of International Symposium on Programming, LNCS 137, pp. 337–351, 1982

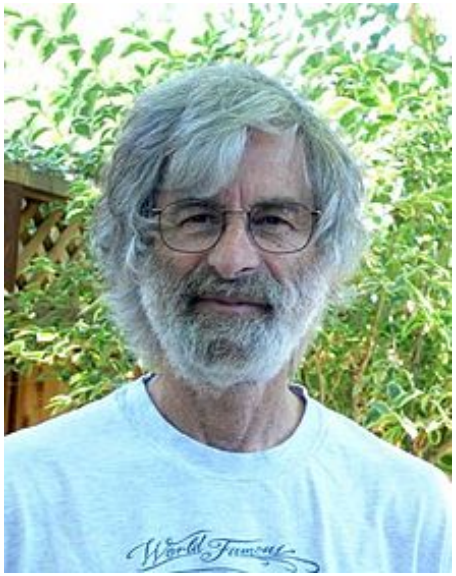
A language-theoretic formulation of model-checking

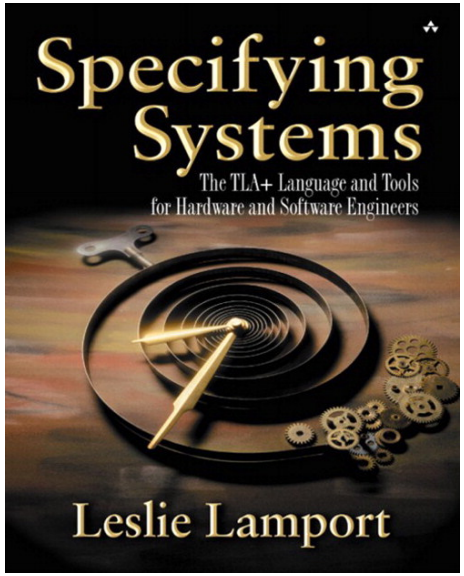
Moshe Vardi
Gödel Prize Winners 2000
Paris Kannelakis Prize Winners 2005

Pierre Wolper

M. Vardi, P. Wolper, *An automata-theoretic approach to automatic program verification*, Proceedings of the First Symposium on Logic in Computer Science, pp. 322–331, 1986

Temporal Logic of Actions (TLA$^+$)





Leslie Lamport
Turing Award Winner 2013

> *For fundamental contributions to the theory and practice of distributed and concurrent systems, notably the invention of concepts such as causality and logical clocks, safety and liveness, replicated state machines, and sequential consistency.*

Recap of some basic maths

Propositional Logic

Definition (Proposition, Propositional Variable)

A proposition is a statement that is either TRUE or FALSE.

A propositional variable is a variable that has one of two possible values: TRUE and FALSE. Propositional variables are used to represent propositions.

- Propositions
  - "It rained in Newcastle on 2 October 2018".
  - "Mike Ashley is an excellent football club chairman"
  - `n <= 1`, `len(buffer) == MAX_BUF_SIZE`
- Propositional variables
  - rain, ashley, p, q, ...
- Not propositions
  - "Pass the salt"
  - "Will Newcastle United win a premier league match in the 2018-19 season?"

Propositional Logic

Definition (Logical Operators)

Compound propositions can be formed from simpler propositions using logical operators, whose meaning can be given by a truth table.

p	q	$\neg p$	$p \vee q$	$p \wedge q$	$p \Rightarrow q$	$p \Leftrightarrow q$
p	q	not ~	or \/	and /\	implies =>	iff <=>
FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	TRUE
FALSE	TRUE	TRUE	TRUE	FALSE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE
TRUE	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE

- *Tautology* — a proposition that is TRUE for all possible valuations of its variables
- *Contradiction* — a proposition that is FALSE for all possible valuations of its variables

Sets

Set theory is the foundation of ordinary mathematics.

We treat the concept of *set* and the relation $\in$ as primitive notions and don't try to define them further.

For a set S, if $x \in S$, we say x is *an element of* S, or more briefly, x *is in* S.

A set can have a finite or infinite number of elements. The set of natural numbers, $0, 1, 2, \ldots$ is infinite. The set of natural numbers less than 3 is finite and can be written as $\{0, 1, 2\}$.

A set is completely determined by its elements. Two sets are equal iff they have the same elements, e.g.

$$\{0, 1, 2\} = \{2, 1, 0\} = \{0, 0, 1, 2, 2\}$$

Let $S \mathrel{\widehat{=}} \{0, 1, 2\}$ then

$$1 \in S \text{ and } 3 \notin S \text{ are TRUE propositions}$$

Set operators

Intersection $\cap$

$S \cap T$ The set of elements in both S and T

$\{1, -1/2, 3\} \cap \{1, 2, 3, 5, 7\} = \{1, 3\}$

Union $\cup$

$S \cup T$ The set of elements in S or T (or both)

$\{1, -1/2\} \cup \{1, 5, 7\} = \{1, -1/2, 5, 7\}$

Set difference $\setminus$

$S \setminus T$ The set of elements in S that are not in T

$\{1, -1/2, 3\} \setminus \{1, 5, 7\} = \{-1/2, 3\}$

Subset $\subseteq$

$S \subseteq T$ TRUE iff every element of S is an element of T

$\{1, 3\} \subseteq \{3, 2, 1\}$ is TRUE

SUBSET and UNION

Given any set S, the *set of all subsets of* S is is written in TLA$^+$ as SUBSET S.

Example Let $S = \{1, 2, 3\}$, then SUBSET S is

$$\{\{\}, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$$

Notice that every element in SUBSET S is a subset of S. There are no other subsets of S.

Mathematicians usually call SUBSET S the *power set* of S and write it as $\mathcal{P}(S)$ or 2^S

If S is a set of sets, then UNION S is the union of all of the elements of S, e.g.

$$\text{UNION}\,\{\{1, 2\}, \{2, 3\}, \{3, 4\}\} = \{1, 2, 3, 4\}$$

Mathematicians usually write UNION S as $\bigcup S$

Cardinality of finite set

Cardinality For any finite set, S, the total number of elements in S is known as the *cardinality* of S.

Mathematicians usually write the cardinality of S as $|S|$.

Notice For any finite set S, $|\mathcal{P}(S)| = 2^{|S|}$

In TLA$^+$, the cardinality operator is written as *Cardinality*, e.g. $\mathit{Cardinality}(\{1, 2, 3\}) = 3$

Set constructors

If S is a set then we can construct new sets from S.

Assume $S = \{1, 2, 3, 4, 5, 6, 7, 8\}$ in the following.

$\{x \in S : p\}$ A set that contains just those elements of S that satisfy some predicate p, e.g. $\{n \in S : n\%2 = 0\}$ is just the set of even numbers in S, i.e. $\{2, 4, 6, 8\}$

$\{e : x \in S\}$ A set of elements of the form e, for all elements of the set S, e.g. $\{2 * n : n \in S\}$ is the set $\{2, 4, 6, 8, 10, 12, 14, 16\}$

TLA$^+$ defines an operator, $..$, on the set *Int* of integers, using a set constructor:

$$m..n \mathrel{\widehat{=}} \{v \in Int : m \leq v \wedge v \leq n\}$$

Predicate Logic

Predicate logic extends propositional logic with two operators called *quantifiers*

- *Universal quantifier* ($\forall$)
  - $\forall\, x \in S : P(x)$ is the conjunction of the formulas $P(x)$, for all x in S
  - $\forall\, n \in \{1, 2, 3\} : n^2 > n$ equals the formula $1^2 > 1 \wedge 2^2 > 2 \wedge 3^2 > 3$
  - typed `\A`
- *Existential quantifier* ($\exists$)
  - $\exists\, x \in S : P(x)$ is the disjunction of the formulas $P(x)$, for all x in S
  - $\exists\, n \in \{1, 2, 3\} : n^2 > n$ equals the formula $1^2 > 1 \vee 2^2 > 2 \vee 3^2 > 3$
  - typed `\E`

Predicate Logic

Two important tautologies:

$$\neg\,(\forall\, x \in S : P(x)) \quad \Leftrightarrow \quad \exists\, x \in S : \neg\, P(x)$$
$$\neg\,(\exists\, x \in S : P(x)) \quad \Leftrightarrow \quad \forall\, x \in S : \neg\, P(x)$$

Some abbreviations:

$\forall\, x \in S, y \in T : P(x, y)$ means
$\forall\, x \in S : \forall\, y \in T : P(x, y)$

$\exists\, x, y, z \in S : P(x, y, z)$ means
$\exists\, x \in S : \exists\, y \in S : \exists\, z \in S : P(x, y, z)$

Predicate Logic

TLA$^+$ syntax problem:

$$\begin{array}{lcll} Foo & \widehat{=} & \forall\, x \in S : P \wedge \forall\, x \in T : Q & \text{is parsed as} \\ Foo & \widehat{=} & \forall\, x \in S : (P \wedge \forall\, x \in T : Q) & \text{error: x multiply defined} \end{array}$$

Can be solved by writing as

$$Foo \;\widehat{=}\; (\forall\, x \in S : P) \wedge (\forall\, x \in T : Q)$$

or, even better

$$\begin{array}{lcll} Foo & \widehat{=} & \wedge & \forall\, x \in S : P \\ & & \wedge & \forall\, x \in T : Q \end{array}$$

A first TLA$^+$ specification

Requirements for a simple building control system

Informal statement of requirements:

1. The system should monitor persons as they enter and leave a building.
2. A person can only enter the building if they are a registered user.
3. The system should be aware of whether a registered user is currently inside or outside the building.

We will attempt to produce a formal specification of this system using TLA$^+$.

What is needed in the TLA$^+$ specification?

A specification of the initial states

- Usually given by an *operation* called *Init*
- *Init* is defined by a *state predicate*, i.e. a predicate on the values of the variables in a single state

A specification of what the next state can be, given any current state

- Usually given by an *operation* called *Next*
- A *state* is completely defined by the value of its variables . . .
- So the *Next* operation needs to specify the relationship between the values of the variables in the current state and the values of the variables in the next state
- This is done using a predicate, relating these two sets of values, that must be true in any behaviour of the system — this is known as an *action* predicate

Action predicates

An *action predicate* specifies a *step* in the behaviour of a system.

It relates the values of the variables in the *first* state of a step to the values of the variables in the *next* state.

We use a *prime* symbol ($'$) to differentiate the value of a variable in the next state of a step from its value in the first state.

For example, for a variable x, x denotes its value in the first state of a step, and x' denotes its value in the next state.

So an action predicate that says that a step increments the value of the variable x by 1 can be written

$$x' = x + 1$$

Notice that this is *not* an assignment statement. It is a predicate — a statement that may be TRUE or FALSE. Any behaviour satisfying this specification requires that the predicate is TRUE.

Representing the state of the system

The informal specification mentions

- *persons* who should be monitored
- a *register* of users
- registered users *inside* the building
- registered users *outside* the building

CONSTANT

PERSON	set of persons

VARIABLE

register,	set of registered users
in,	set of users inside the building
out	set of users outside the building

The Init Operation

The *Init* operation identifies the possible initial states, e.g.

$$
\begin{aligned}
&\mathit{Init} \mathrel{\widehat{=}} \\
&\quad \land\ \mathit{register} = \{\} \\
&\quad \land\ \mathit{in} = \{\} \\
&\quad \land\ \mathit{out} = \{\}
\end{aligned}
$$

The *Init* operation is defined by a state predicate, i.e. a predicate that must be TRUE for any initial state in a behaviour of this system

This predicate is *deterministic*, i.e. there is only *one possible state* that makes it TRUE

The Register Operation

An operation is specified using a definition, which may have *zero or more parameters*

- What kind of thing can we register?
- Should it be possible to register a person who is already registered?
- What should be TRUE of the value of the `register` variable in the *next state*?
- Is that all, e.g. should this person be noted also as inside or outside the building?
- Are we done now?

$$
\begin{aligned}
&Register(p) \mathrel{\widehat{=}} \\
&\quad \land\ p \in PERSON \setminus register \\
&\quad \land\ register' = register \cup \{p\} \\
&\quad \land\ out' = out \cup \{p\} \\
&\quad \land\ in' = in
\end{aligned}
$$

The Enter Operation

- What should be TRUE of `p` in the first state of any `Enter` step?
- What should be TRUE of the value of the `in` variable in the next state?
- What should be TRUE of the value of the `out` variable in the next state?
- What should be TRUE of the value `register` variable in the next state?

$$
\begin{array}{l}
Enter(p) \mathrel{\widehat{=}} \\
\quad \land\ p \in PERSON \quad \land\ p \in register \quad \land\ p \in out \\
\quad \land\ in' = in \cup \{p\} \\
\quad \land\ out' = out \setminus \{p\} \\
\quad \land\ register' = register
\end{array}
$$

The Leave Operation

Exercise: Take 2 or 3 minutes to write down your own specification of the `Leave` operation

$$
\begin{aligned}
&\mathit{Leave}(p) \mathrel{\widehat{=}} \\
&\quad \land\ p \in \mathit{in} \\
&\quad \land\ \mathit{out}' = \mathit{out} \cup \{p\} \\
&\quad \land\ \mathit{in}' = \mathit{in} \setminus \{p\} \\
&\quad \land\ \mathit{register}' = \mathit{register}
\end{aligned}
$$

The Next Operation

Now we need to combine the `Register`, `Enter` and `Leave` operations into a definition of the `Next` operation

A `Next` step can be

- *either* a `Register` step
- *or* an `Enter` step
- *or* a `Leave` step

Each step might involve any member of the `PERSON` set

$$
\begin{array}{l}
\mathit{Next} \mathrel{\widehat{=}} \\
\quad \exists\, p \in \mathit{PERSON} : \\
\qquad \lor\ \mathit{Register}(p) \\
\qquad \lor\ \mathit{Enter}(p) \\
\qquad \lor\ \mathit{Leave}(p)
\end{array}
$$

This definition of `Next` is *non-deterministic*, i.e. there may be many different possible next states that satisfy this predicate

A final thought

What about types?

What are the types of our variables `register`, `in`, and `out`?

TLA$^+$ is *untyped*!!

Actually, every value in TLA$^+$ is a *set*

TLA$^+$ doesn't impose restrictions through a type system on what we can say in a specification but it is often helpful for us as specifiers to impose our own restrictions

We normally do this by defining our own *type invariant*, i.e. a predicate that should be TRUE for the values of all our variables throughout any behaviour of the system

A Type Invariant

What would be a sensible type invariant for the simple building control system?

$$
\begin{array}{l}
\mathit{TypeOk} \mathrel{\widehat{=}} \\
\quad \mathit{register} \subseteq \mathit{PERSON} \\
\quad \mathit{register} = \mathit{in} \cup \mathit{out} \\
\quad \mathit{in} \cap \mathit{out} = \{\}
\end{array}
$$

Everyone on the register is a PERSON

The location of every registered PERSON is known

Nobody can at the same time be both inside and outside the building

We use our model-checker, TLC, to check that this type predicate is TRUE in every state of every behaviour of our system model, i.e. that it really is *invariant*

View the complete specification

Summary

- Model-driven design approach
- Employ formal methods to model and analyse systems
- Detect bugs early in design lifecycle
- Model-checking can
  - Demonstrate that a model has required properties
  - Be used in practice by engineers with only limited mathematical knowledge
- We'll begin by writing specifications using TLA$^+$
- . . . and use TLC, a model checker that is freely available, used in industry, e.g. at Amazon, Microsoft, Intel, . . . , and widely respected.
- Module pages at http://hesabu.net/cm0604/ and http://hesabu.net/kf6009/