Logics of Formal Software Specification Languages

Dines Bjørner

Technical University of Denmark

The 4th Panhellenic Logic Symposium

Thessaloniki, Greece, 7 July 2003, 11:00 – 11:50 am

An Essence

- In my mind the most outstanding problem of informatics,
  - ⋆ as a discipline separate from mathematics
  - ⋆ and as a discipline which is not based on the natural sciences,
- is that of providing, and having provided
  - ⋆ principles, techniques and tools

  to be, and being used in
  - ⋆ actual, and adequate

  descriptions of the world in which computing applications reside.
- For descriptions we focus on formal specification languages.

- There are, today, a number of reasonably widespread formal specification languages for use in the development, primarily, of software.
- Among these we select the following for our survey analysis:

	ASM	Abstract–State Machines	Y.Gurevich,E.Börger	1990
	B	after **B**ourbaki	J.–R.Abrial	1990
	CafeOBJ		K.Futatsuigi et al.	1995
	CASL	Common Algebraic Specification Language	CoFI	1995
♡	DC	Duration Calculi	Zhou ChaoChen	1990
♡♡	RSL	Raise Specification Language	C.George,S.Prehn, D.Bjørner et al.	1989
	TLA^{+}	Temporal Logic of Actions	L.Lamport	1985
♡♡♡	VDM	Vienna Development Method	H.Bekič, D.Bjørner et al.	1974
	Z	after **Z**ermelo	J.–R.Abrial	1980

- ♡: I was myself a co–originator (♡♡♡), an instigator (♡♡), resp. an “inspirator” (♡) of ...
- VDM (♡♡♡) and Z will, however, for space as well as time reasons, not be covered !

Structure of Talk

- (1) We overview the disciplines of the Computer & Computing Sciences, Software Engineering, Informatics and IT.
- (2) Then we present an example specification, expressed in RSL (the Raise Specification Language),
- (3) After that example, we overview, in a bit more detail, some of the above listed specification languages.
  - ⋆ The individual language cum logic overview will be very brief.

  There is no chance that “newcomers” will understand much of this part — it is meant as a teaser ! — there are too many languages.

  - ⋆ Therefore, please consult Volume 22, Nos. 3–4, 2003 double–issue of the Slovak Academy journal *Computing and Informatics* for details.
- (4) After this main part of the talk I emphasise, what I consider an important difference between the mathematical logicians' approach to mathematical logic, and that of the computing scientist cum software engineer.

The Disciplines of Our Field

- **Computer Science:**

  To us computer science is the study and knowledge of the **properties** of the "things" that can "exist" inside computers: Data and processes.

- **Computing Science:**

  To us computing science is the study and knowledge of **how to construct** those "things" that can "exist" inside computers.

- **Computer Science:**

  Computer science is characterised by such subjects as

  ⋆ computability, ie., computational models
  λ–Calculus, Turing Machines, foundations of Petri Nets, etc.,

  ⋆ denotational, axiomatic, operational and algebraic semantics foundations,

  ⋆ complexity theory,

  ⋆ type theory,

  ⋆ the mathematics of cryptography,

  ⋆ the logics of proof systems,

  ⋆ *&c.*

- **Computing Science:**

  Computing science is characterised by such subjects as

  ⋆ functional, imperative, logic, and parallel programming,

  ⋆ algorithms = data structures + control,

  ⋆ abstraction & modelling using formal specification languages and the related posit (ie., invent) & verify, respectively specification refinement, calculi — in other words: verification through the use of "proven" refinement rules, through theorem proving, and through model checking, and

  ⋆ special principles, techniques and tools for

    ⋄ compiler design,

    ⋄ database design,

    ⋄ database management systems design,

    ⋄ operating system design,

    ⋄ data communication systems (including protocol) design,

    ⋄ the design of real–time, embedded and safety critical systems, and

    ⋄ the design of application systems in general.

- Both **computer science** and **computing science** rely on mathematics: They both use mathematics, extensively. In works by **computer scientists** it is sometimes difficult to discern what is a possible contribution to **mathematics**, and what is a posible contribution to **computing**.
- To us the **software engineer** "walks the bridge" between computing science and technology: Constructs software (ie., technology) based on scientific insight, and/or analyses technology in order to ascertain its possible scientific content.

We mention the above so that we can see that **mathematical logic** is a very necessary educational background also for software engineers cum programmers.

Software Engineering

- To us **software engineering** combines computing science with practical, more–or–less socio–economic concerns such as software development project management and software product management.
  - ⋆ Software product management is on getting groups of people to work harmoniously, according to plans; allocating & scheduling resources (people, monies, equipment, office space, etc.), monitoring and controlling plans, and, first & foremost, ensuring that the developed software is correct with respect to requirements, and otherwise meets customer expectations. The latter includes, significantly, that the software is valid: Fits the domain in which it is (to be) inserted.

⋆ Software product management is concerned with such issues as devicing strategies and tactics for formulating, launching, marketing and servicing software products: The stepwise release of versions, incrementally adding optional features, pricing, etc.

We mention the above so that we can see that **mathematical logic** is a very necessary educational background also for software managers.

Informatics

- To us informatics is the combination of applied mathematics, computer and computing science, software engineering and applications. Informatics is thus not based on the natural sciences.

Information Technology

- To us information technology includes both hardware, ie., computer, data communications, human oriented input/out devices, and sensor/actuator technologies. Information technology is based on the natural sciences.

Information Technology vs. Informatics

- Information technology stands in relation to informatics, analogically as bio–technology stands to biology.That is: Only nominally !

A Trilogy of Development

- Before **software**, $\mathcal{S}$, can be **designed**
- we must understand its **requirements**, $\mathcal{R}$.
- Before requirements can be fully understood,
- we must first understand the application **domain**, $\mathcal{D}$.

Ideally speaking we should therefore, in any sound and professionally believable development project,

- start with developing a **domain theory** $\mathcal{D}$,
- from it (and other factors) "derive" the **requirements** $\mathcal{R}$,
- and then **design** the **software** $\mathcal{S}$.
- Proof of correctness of software with respect to requirements often need resort to assumptions about the domain, ie., the environment:

$$\mathcal{D}, \mathcal{S} \models \mathcal{R}$$

“What Can be Described ?”

- Physicists have spent almost half a millenium describing and analysing descriptions of Mother Nature.
- And civil (ie., building), mechanical, electrical, electronic and chemical engineers build on these theories.
- Nobody in their right mind would hire a technician without a deep insight into such theories as aerodynamics for the design of supersonic airplanes.
- Nobody in their right mind would hire an amateur chemist without a deep insight into biochemistry for the development of new medicines.
- But in software development that is what is going on !
- The software you use in your daily life is mostly developed by “programmer technicians” who are unaware of even basic facts of reasoning.
- They, and even rather professional software engineers have rarely paused to ponder: *“What Can be Described ?”*.

On Formal Specification Languages

- By a formal specification language we mean a set of sentences, ie., a language, with a formal syntax, with a formal semantics, and with a formal proof system —
- such that, ideally, the latter has been shown consistent wrt. the formal semantics, and otherwise relative complete.
- The formal semantics is usually given in some mathematical style:
  - ⋆ Denotational,
  - ⋆ structural operational,
  - ⋆ or other.

Aims & Objectives of Formal Specification

- The aims of formal specification languages are to be deployed in the development of both computing hardware and software.
- A main objective is to achieve as near correct developments as at any moment feasible: From domain descriptions via requirements prescriptions to software designs and program code.
- Other objectives include those of achieving elegant designs and conceptually clean and beautiful software.

An Example Specification

- We bring an example formal specification.
- It purports to model some intrinsic aspects of railway systems, namely rail net topologies:
  - ⋆ Their static layout and the possible dynamics of trains and traffic on these nets.
- The specification is expressed in RSL (the RAISE Specification Language).

Informal Narrative: The Statics of Rail Nets

We introduce the phenomena of railway nets, lines, stations, tracks, (rail) units, and connectors.

- Instead of “reading” the informal, but neccessary, informal text of slides 17–18 and 23–25, I will quickly skip to “reading” the formal, also neccessary, text of slides 19–22, respectively slides 26–29.

1. A railway net consists of one or more lines and two or more stations.
2. A railway net consists of rail units.
3. A line is a linear sequence of one or more linear rail units.
4. The rail units of a line must be rail units of the railway net of the line.
5. A station is a set of one or more rail units.
6. The rail units of a station must be rail units of the railway net of the station.
7. No two distinct lines and/or stations of a railway net share rail units.
8. A station consists of one or more tracks.

9. A track is a linear sequence of one or more linear rail units.
10. No two distinct tracks share rail units.
11. The rail units of a track must be rail units of the station (of that track).
12. A rail unit is either a linear, or is a switch, or a is simple crossover, or is a switchable crossover, etc., rail unit.
13. A rail unit has one or more connectors.
14. A linear rail unit has two distinct connectors, a switch rail unit has three distinct connectors, crossover rail units have four distinct connectors (whether simple or switchable), etc.
15. For every connector there are at most two rail units which have that connector in common.
16. Every line of a railway net is connected to exactly two, distinct stations of that railway net.
17. A linear sequence of (linear) rail units is a non-cyclic sequence of linear units such that neighbouring units share connectors.

Formalisation: The Statics of Rail Nets

- We define sorts (ie., abstract types) and observer functions, ie., postulated function which, from values of sorts observe values, usually sets of values of other sorts.
- The formula numbers refer to enumerated narrative entries.

type

1. N, L, S
2. U
8. Tr
13. C

value

1. obs_Ls: N → L-**set**,
1. obs_Ss: N → S-**set**
2. obs_Us: N → U-**set**,
3. obs_Us: L → U-**set**
5. obs_Us: S → U-**set**,

8. obs_Trs: S $\rightarrow$ Tr-**set**
12. is_Linear: U $\rightarrow$ **Bool**,
12. is_Switch: U $\rightarrow$ **Bool**
12. is_Simple_Crossover: U $\rightarrow$ **Bool**,
12. is_Switchable_Crossover: U $\rightarrow$ **Bool**
13. obs_Cs: U $\rightarrow$ C-**set**

17. lin_seq: U-**set** $\rightarrow$ **Bool**
    lin_seq(us) $\equiv$
      $\forall$ u:U $\cdot$ u $\in$ us $\Rightarrow$ is_Linear(u) $\wedge$
      $\exists$ q:U* $\cdot$ **len** q = **card** us $\wedge$ **elems** q = us $\wedge$
        $\forall$ i:**Nat** $\cdot$ {i,i+1} $\subseteq$ **inds** q $\Rightarrow$
          $\exists$ c:C $\cdot$ obs_Cs(q(i)) $\cap$ obs_Cs(q(i+1)) = {c} $\wedge$
        len q > 1 $\Rightarrow$ obs_Cs(q(i)) $\cap$ obs_Cs(q(**len** q)) = {}

Some formal axioms are now given, not all !

axiom

1. $\forall$ n:N $\cdot$ **card** obs_Ls(n) $\geq$ 1,

1. $\forall$ n:N $\cdot$ **card** obs_Ss(n) $\geq$ 2,

3. $\forall$ n:N, l:L $\cdot$ l $\in$ obs_Ls(n) $\Rightarrow$ lin_seq(l)

4. $\forall$ n:N, l:L $\cdot$ l $\in$ obs_Ls(n) $\Rightarrow$ obs_Us(l) $\subseteq$ obs_Us(n)

5. $\forall$ n:N, s:S $\cdot$ s $\in$ obs_Ss(n) $\Rightarrow$ **card** obs_Us(s) $\geq$ 1

6. $\forall$ n:N, s:S $\cdot$ s $\in$ obs_Ls(n) $\Rightarrow$ obs_Us(s) $\subseteq$ obs_Us(n)

7. $\forall$ n:N, l,l′:L $\cdot$ {l,l′} $\subseteq$ obs_Ls(n) $\wedge$ l$\neq$l′ $\Rightarrow$ obs_Us(l) $\cap$ obs_Us(l′) = {}

7. $\forall$ n:N, l:L, s:S $\cdot$ l $\in$ obs_Ls(n) $\wedge$ s $\in$ obs_Ss(n) $\Rightarrow$ obs_Us(l) $\cap$ obs_Us(s) = {}

7. $\forall$ n:N, s,s′:S $\cdot$ {s,s′} $\subseteq$ obs_Ss(n) $\wedge$ s$\neq$s′ $\Rightarrow$ obs_Us(s) $\cap$ obs_Us(s′) = {}

8. $\forall$ s:S $\cdot$ **card** obs_Trs(s) $\geq$ 1

9. $\forall$ n:N, s:S, t:T $\cdot$ s $\in$ obs_Ss(n) $\wedge$ t $\in$ obs_Trs(s) $\Rightarrow$ lin_seq(t)

10. $\forall$ n:N, s:S, t,t′;T $\cdot$
    s $\in$ obs_Ss(n) $\wedge$ {t,t′} $\subseteq$ obs_Trs(s) $\wedge$ t$\neq$t′ $\Rightarrow$ obs_Us(t) $\cap$ obs_Us(t′) = {}

15. $\forall$ n:N $\cdot$ $\forall$ c:C $\cdot$ c $\in$ $\cup$ { obs_Cs(u) | u:U $\cdot$ u $\in$ obs_Us(n) } $\Rightarrow$
    card{ u | u:U $\cdot$ u $\in$ obs_Us(n) $\wedge$ c $\in$ obs_Cs(u) } $\leq$ 2

16. $\forall$ n:N,l:L $\cdot$ l $\in$ obs_Ls(n) $\Rightarrow$
    $\exists$ s,s′:S $\cdot$ {s,s′} $\subseteq$ obs_Ss(n) $\wedge$ s$\neq$s′ $\Rightarrow$
    let sus = obs_Us(s), sus′ = obs_Us(s′), lus = obs_Us(l) **in**
    $\exists$ u:U $\cdot$ u $\in$ sus, u′:U $\cdot$ u′ $\in$ sus′, u″,u‴:U $\cdot$ {u″,u‴} $\subseteq$ lus $\cdot$
    let scs = obs_Cs(u), scs′ = obs_Cs(u′),
    lcs = obs_Cs(u″), lcs′ = obs_Cs(u‴) **in**
    $\exists$! c,c′:C $\cdot$ c $\neq$ c′ $\wedge$ scs $\cap$ lcs = {c} $\wedge$ scs′ $\cap$ lcs′ = {c′}
    end end

Informal Narrative: The Dynamics of Nets and Trains

We introduce defined concepts such as paths through rail units, state of rail units, rail unit state spaces, routes through a railway network, open and closed routes, trains on the railway net, and train movements on the railway net.

18. A path, $p : P$, is a pair of connectors, (c, c'),
19. which are distinct,
20. and of some unit.[1]
21. A state, $\sigma : \Sigma$, of a unit is the set of all open paths of that unit (at the time observed).[2]
22. A unit may, over its operational life, attain any of a (possibly small) number of different states ω, Ω.
23. A route is a sequence of pairs of units and paths —

[1] A path of a unit designate that a train may move across the unit in the direction from c to c'. We say that the unit is open in the direction of the path.
[2] The state may be empty: the unit is closed.

24. such that the path of a unit/path pair is a possible path of some state of the unit, and such that "neighbouring" connectors are identical.

25. An open route is a route such that all its paths are open.

26. A train is modelled as a route.

27. Train movement is modelled as a discrete function (ie., a map) from time to routes

28. such that for any two adjacent times the two corresponding routes differ by at most one of the following:

    (a) a unit path pair has been deleted (removed) from one end of the route;
    (b) a unit path pair has been deleted (removed) from the other end of the route;
    (c) a unit path pair has been added (joined) from one end of the route;
    (d) a unit path pair has been added (joined) from the other end of the route;
    (e) a unit path pair has been added (joined) from one end of the route, and another unit path par has been deleted (removed) from the other end of the route;

(f) a unit path pair has been added (joined) from the other of the route, and another unit path par has been deleted (removed) from the one end of the route;

(g) or there has been no changes with respect to the route (yet the train may have moved);

29. and such that the new route is a well–formed route.

Formalisation: The Dynamics of Nets and Trains

type

18. $P' = C \times C$
19. $P = \{| \ (c,c'):P' \cdot c \neq c' \ |\}$
21. $\Sigma = P\textbf{-set}$
22. $\Omega = \Sigma\textbf{-set}$
23. $R' = (U \times P)^*$
24. $R = \{| \ r:R' \cdot wf\_R(r) \ |\}$
26. $Trn = R$
27. $Mov' = T \xrightarrow[m]{} Trn$
28. $Mov = \{| \ m:Mov' \cdot wf\_Mov(m) \ |\}$

value

21. $obs\_\Sigma: U \rightarrow \Sigma$
22. $obs\_\Omega: U \rightarrow \Omega$

axiom
$\forall$ u:U $\cdot$
let ω = obs_Ω(u), σ = obs_Σ(u) **in** $\sigma \in \omega \wedge$
20. **let** cs = obs_Cs(u) **in** $\forall$ (c,c′):P $\cdot$ (c,c′) $\in \cup\, \omega \Rightarrow$ {c,c′} $\subseteq$ obs_Cs(u)
end end

24. wf_R: R′ $\rightarrow$ **Bool**
wf_R(r) $\equiv$
len r > 0 $\wedge$
$\forall$ i:**Nat** $\cdot$ i $\in$ **inds** r **let** (u,(c,c′)) = r(i) **in**
(c,c′) $\in \cup$ obs_Ω(u) $\wedge$ i+1 $\in$ **inds** r $\Rightarrow$ **let** (_,(c″,_)) = r(i+1) **in** c′ = c″ **end end**

25. open_R: R $\rightarrow$ **Bool**
open_R(r) $\equiv \forall$ (u,p):U$\times$P $\cdot$ (u,p) $\in$ **elems** r $\wedge$ p $\in$ obs_Σ(u)

27. wf_Mov: Mov $\rightarrow$ **Bool**
wf_Mov(m) $\equiv$ **card dom** m $\geq 2 \wedge$

$$
\begin{array}{l}
\quad \forall\ t,t'{:}T \cdot t,t' \in \mathbf{dom}\ m \wedge t < t' \\
\quad \wedge\ \mathrm{adjacent}(t,t') \Rightarrow \\
\qquad \mathbf{let}\ (r,r') = (m(t),m(t')) \\
\qquad\quad (u,p){:}U{\times}P \cdot p \in \cup\ \mathrm{obs\_\Omega}(u)\ \mathbf{in} \\
28a.\quad (\mathrm{l\_d}(r,r',(u,p)) \quad \vee \quad 28b.\quad \mathrm{r\_d}(r,r',(u,p)) \quad \vee \\
28c.\quad \mathrm{l\_a}(r,r',(u,p)) \quad \vee \quad 28d.\quad \mathrm{r\_a}(r,r',(u,p)) \quad \vee \\
28e.\quad \mathrm{l\_d\_r\_a}(r,r',(u,p)) \vee \quad 28f.\quad \mathrm{r\_d\_l\_a}(r,r',(u,p)) \vee \\
28g.\quad r{=}r') \wedge \mathrm{wf\_R}(r') \\
\qquad \mathbf{end}
\end{array}
$$

Last line's route well–formedness ensures that type of Move is maintained.

value

$$
\begin{array}{l}
\mathrm{adjacent}{:}\ T \times T \rightarrow \mathbf{Bool} \\
\mathrm{adjacent}(t,t') \equiv \sim\exists\ t''{:}T \cdot t'' \in \mathbf{dom}\ m \wedge t < t'' < t'
\end{array}
$$

$$
\mathrm{l\_d,r\_d,l\_a,r\_a,l\_d\_r\_a,r\_d\_l\_a}{:}\ R \times R \times P \rightarrow \mathbf{Bool}
$$

$$\text{l\_d}(\text{r},\text{r}',(\text{u},\text{p})) \equiv \text{r}' = \textbf{tl}\ \text{r} \qquad \textbf{pre len}\ \text{r}>1$$
$$\text{r\_d}(\text{r},\text{r}',(\text{u},\text{p})) \equiv \text{r}' = \text{fst}(\text{r}) \quad \textbf{pre len}\ \text{r}>1$$
$$\text{l\_a}(\text{r},\text{r}',(\text{u},\text{p})) \equiv \text{r}' = \langle(\text{u},\text{p})\rangle^\frown \text{r}$$
$$\text{r\_a}(\text{r},\text{r}',(\text{u},\text{p})) \equiv \text{r}' = \text{r}^\frown\langle(\text{u},\text{p})\rangle$$
$$\text{l\_d\_r\_a}(\text{r},\text{r}',(\text{u},\text{p})) \equiv \text{r}' = \textbf{tl}\ \text{r}^\frown\langle(\text{u},\text{p})\rangle$$
$$\text{r\_d\_l\_a}(\text{r},\text{r}',(\text{u},\text{p})) \equiv \text{r}' = \langle(\text{u},\text{p})\rangle^\frown\text{fst}(\text{r})$$

$$\text{fst: R} \stackrel{\sim}{\to} \text{R}'$$
$$\text{fst}(\text{r}) \equiv \langle\ \text{r}(\text{i}) \mid \text{i}\ \textbf{in}\ \langle 1..\textbf{len}\ \text{r}-1\rangle\ \rangle$$

Discussion

Question: *What is being expressed in the above definition ?*

- The specification given above presents a mathematical model of what is considered core properties of a railway system.
- As such it answers, in a mathematical manner, the questions:
  - ⋆ *What is a railway net ?*
  - ⋆ *What does it mean that a railway net has signals and switches, ie.,*
  - ⋆ *that certain routes are open and others close ?*
  - ⋆ *What is a train ?*
  - ⋆ *What is a train movement ?*
  - ⋆ Etcetera.

⋆ Since we may have skipped some earlier slides we repeat below:

- From a **domain specification** like the above one can then develop and study a **domain theory** of railway systems.
- And one can "derive", or relate, **requirements** to software for railway system planning, monitoring and control from, respectively to the above kind of **domain specification.**
- And from such RAISE specified **requirements** one can then develop **software designs**
- and **prove** these **correct** wrt. the **requirements** (and in the context of the **domain specification**).

The Logics of Specification Languages

ASM: Abstract–State Machines

- An ASM specification, roughly speaking, is an unordered set of clauses.

$$\{c_1, c_2, \ldots, c_m\}$$

- Each clause is of the form:

$$c_i : p_i(t_{pi_1}, t_{pi_2}, \ldots, t_{pi_{n_i}}) : f_i(t_{fi_1}, t_{fi_2}, \ldots, t_{fi_{n_i}}) := t_i$$

- p_i and f_i are predicate, respectively function symbols, t_{pi_j}, t_{fi_j} are terms.
- The symbol := designates assignment.
- If i_{fn_i} is 0, then the assignment is to a simple variable of the value of term t_i.
- Otherwise the assignment "changes" function f_i.
- Terms $p_j(t_{j_1}, t_{j_2}, \ldots, t_{j_{n_j}})$ or $f_j(t_{j_1}, t_{j_2}, \ldots, t_{j_{n_j}})$ for $j_{n_j} = 0$ are ground terms and denote constant values, like truth values, or integer, or sets, or other.
- If i_{pn_i} is 0 then the guard (p_i) is always true or always false.

- If we consider an algebra as a possibly infinite set of constant values and a possibly infinite set of functions, then an ASM specification denotes an evolving algebra.
- Evaluation of the clauses occur, repeatedly, in parallel, for all such clauses whose guard ($p_i(t_{pi_1}, t_{pi_2}, \ldots, t_{pi_{n_i}})$) is true.
- The evolving algebra terminates (ie., has a fix–point) if no $p_i(t_{pi_1}, t_{pi_2}, \ldots, t_{pi_{n_i}})$, at some point, is true.

- "Classical" ASM does not prescribe a formal syntax for terms.
- And "classical" ASM allows values to not necessarily be effective, ie., computable.
- Thus ASM is a kind of a specification mechanism which appears rather well suited to express abstract algorithms over arbitrary structures.

Some remarks on the Logic of ASM:

- Reasoning over ASM specifications occurs, not within a formal proof system, but basically in the classical style of mathematics.

Question: *How might ASM express the railway example ?*

- Well, only with some difficulty could an ASM specification be made to express answers to the questions posed at the very end of the railway example.
- ASM is "geared" to specify operational behaviours, in particular algorithms, not denotations, not the kind of properties expressed by the railway example.

B

- A B specification consists of one or more abstract machine definitions expressed in the abstract machine notation (AMN):

machine $\mathrm{m}(\mathrm{a}_1,\mathrm{a}_2,\ldots,\mathrm{a}_a)$
  constraints
    $\mathrm{c}(\mathrm{a}_1,\mathrm{a}_2,\ldots,\mathrm{a}_a)$
  sets
    $\mathrm{s}_1,\mathrm{s}_2,\ldots,\mathrm{s}_s$
  constants
    $\mathrm{k}_1,\mathrm{k}_2,\ldots,\mathrm{k}_k$
  properties
    $\mathrm{p}(\mathrm{s}_1,\mathrm{s}_2,\ldots,\mathrm{s}_s,\mathrm{k}_1,\mathrm{k}_2,\ldots,\mathrm{k}_k)$
  variables
    $\mathrm{v}_1,\mathrm{v}_2,\ldots,\mathrm{v}_v$
  invariant
    $\mathrm{inv}(\mathrm{v}_1,\mathrm{v}_2,\ldots,\mathrm{v}_v)$
  initialisation
    $\mathrm{v}_1{:=}\mathrm{val}_1,\mathrm{v}_2{:=}\mathrm{val}_2,\ldots,\mathrm{v}_v{:}\mathrm{val}_v$
  operations
    $\mathrm{f}_1(\mathrm{x}_{11},\mathrm{x}_{12},\ldots,\mathrm{x}_{11m}) \equiv \mathrm{clause}_1$
    $\mathrm{f}_2(\mathrm{x}_{21},\mathrm{x}_{22},\ldots,\mathrm{x}_{22m}) \equiv \mathrm{clause}_2$
    ...
    $\mathrm{f}_n(\mathrm{x}_{n1},\mathrm{x}_{n2},\ldots,\mathrm{x}_{nnm}) \equiv \mathrm{clause}_n$
  assertions
    $\mathrm{q}(\mathrm{v}_1,\mathrm{v}_2,\ldots,\mathrm{v}_v)$
end

- A schema calculus permits combination of abstract machines:
  - ⋆ Seeing,
  - ⋆ including,
  - ⋆ extending,
  - ⋆ promoting,
  - ⋆ using, and
  - ⋆ importing

  abstract machines.
- This schema calculus plus additional meta–linguistic (wrt. AMN) constructs
  - ⋆ aid the developer in structuring and
  - ⋆ proving properties of refinements.

Some remarks on the Logic of B:

- The logic is an untyped first order logic
- while the proof style is natural deduction.
- Although the logic is untyped, the set theory is typed
- — a set can be defined syntactically using a predicate,
- but this definition will only represent a valid set if it satisfies some type-checking rules.

Question: *How might B express the railway example ?*

- Well, quite easily one could rephrase the specific railwsy example into B.
- In fact, B probably represents one of the commercially most successfull approaches to railway system models.[3]
- — having been deployed in the development,
  - ⋆ from domain models,
  - ⋆ into requirements, and from such
  - ⋆ into actually correctness verified code for train control.

[3] [1] Babak Dehbonei, Fernando Meija: Verification of Proofs for the B Formal Development Process. SIGPLAN Notices 28(11): 16-21 (1993)
[2] G. Guiho, F. Mejia: Operational Safety Critical Software Methods in Railways. IFIP Congress (3) 1994: 262-269.
[3] Babak Dehbonei, Fernando Mejia: Formal Methods in the Railways Signalling Industry. FME Rail Workshop 1994: 26-34.

CafeOBJ

- Let us first illustrate a simple abstract data type example.
- A queue parameterised to (abitrary) elements:

```
mod* EQTRIV {
  [Elt]
  op _=_ : Elt Elt -> Bool {comm}
  }

mod! QUEUE (D :: EQTRIV) {
  [Queue]
  op empty  :              -> Queue
  op __     : Queue Queue -> Queue
  op put    : Queue Elt.D -> Queue
  op get    : Queue        -> Queue
  op top    : Queue        -> Elt.D
  op empty? : Queue        -> Bool
```

```
op _\in_  : Elt.D Queue -> Bool
var Q : Queue
vars X Y : Elt.D
eq put(empty,X)   = X empty .
eq put((Y Q),X)   = Y put(Q X) .
eq get(empty)     = empty .
eq get((X Q))     = Q .
eq top((X Q))     = X .
eq empty?(empty) = true .
eq empty?((X Q)) = false .
eq X \in empty   = false .
eq X \in (X Q)   = true .
ceq X \in (Y Q)  = X \in Q if not(X=Y) .
```

- With the above one can now prove:

```
For any q : Queue and any x,y : Elt.D,

[1]  (not empty?(q) and top(q) = x implies (top(put(q,y))=x),
[2]  not empty?(put(q,x)),
[3]  empty?(q) implies (not x \in q),
[4]  (x \in put(q,x)), and
[5]  (x \in q) implies (x \in put(q,y)).
```

- CafeOBJ is an algebraic specification (cum programming) language.
- The semantics of CafeOBJ is given in terms of category theoretical notions, in particular institutions.
- In this way it is possible to use CafeOBJ for diverse purposes:
  - ⋆ To (classically) model abstract data types with loose (*) and strict (or tight. !) semantics (inital algebra approach),
  - ⋆ to model behaviours (ie., process algebras and transition systems, over hidden states, based on coherent hidden algebras), etc.
- To achieve its unquestionable "power" CafeOBJ employs a technique of flattening, into a Grothendieck institution a number of such:
  - ⋆ Hidden order sorted algebras, many sorted algebras, rewrite logics, etc.
  - ⋆ A powerful rewrite logic allows behavioural specifications to be executed.

Some remarks on the Logic of CafeOBJ:

- The base logic of CafeOBJ is based on equational logic.
- Several equational theories are possible: Modulo associativity, commutativity, idenpotence, and combinations of these.
- The CafeOBJ semantics is based on the theory of institutions, a categorical abstract model theoretic meta–theory of logics.
- CafeOBJ thus allows many different kinds of logics to be deployed by suitable instantiation.

Question: *How might CafeOBJ express the railway example ?*

- In CafeOBJ one would first define the abstract data types of connectors, then units, then lines, tracks and stations.
- From the data types of units (ie., including connectors) one can then define the data types of paths, unit states and spaces of unit states.
- From these and the data type of nets one can then go on to define open and closed routes, train movements, etc.
- We did not, in the railway example, show any actions causing transitions on unit. But had we done so, we would, in CafeOBJ, have used its concept of *Abstract Machines* and *Observational Transition Systems.*

CASL: Common Algebraic Specification Language

- CASL, lile CafeOBJ, is an algebraic specification language.
- A CASL specification denotes a possibly infinite set of many–sorted algebras, that is: Collections of sets of data together with functions over those sets.
- As for CafeOBJ, the CASL semantics is expressed within an institutional framework.
- The institutions provide a linkage between the catehory, sum algebraic, semantics of CASL and model theory, and the linkage explains any suitable proof theory and the related logical consequence (or semantic entailment).
- Thus a logic is seen as an institution equipped with an entailment relation.

- An example specification is:

%list [__], nil, __::__
%precedence {__::__} < {__++__}

spec List [**sort** Elem] =
  free type List[Elem] ::= nil | __::__(head :? Elem,tail :? List[Elem]);
  sort NEList[Elem] = {L:List[Elem] · ∼L = nil};
  op __++__ : List[Elem] × List[Elem] → List[Elem]
  forall e:Elem, K,L:List[Elem]
    · nil ++ L = L
    · (e :: K) ++ L = e::K ++ L
end

Some remarks on the Logic of CASL:

- The CASL proof calculus lists fourteen deduction rules:

absurdity	tertium non datur	implication introduction
implication elimination	universal quantor introduction	universal quantor elimination
reflexivity	congruence	totality
substitution	function strictness	predicate strictness
induction	sort generator introduction	

- The logic is sound.
- Without the sort generation constraints the logic is complete.

Question: *How might CASL express the railwy example ?*

- An answer quite similar to those at the end of the CafeOBJ overview apply here as well.

DC: The Duration Calculi

Interval Logic

- First we briefly overview a so–called *Interval Logic*, IL:
  - $\star$ Global variables, **GVar**, are referred to as x (etc.), and contain the usual (assignable) values.
  - $\star$ Temporal variables, **TVar**, are referred to as v (etc.), and their meanings are that they are functions from (real valued) time to real valued intervals (of time).
  - $\star$ The special temporal variable ℓ stands for the interval function which gives the length of the interval (of the context in which it occurs) as value.
  - $\star$ There are the usual (arity n) function, F, and predicate function, G, symbols.
  - $\star$ Π stands for temporal proposition letters, each of which stands for a truth–valued interval function.

⋆ And there are terms, Θ, over these and the state and temporal variables. (Truth valued) Formulas over terms are:

$$\Phi ::= \Pi \mid G(\theta_1, ..., \theta_n) \mid \neg\Phi \mid \Phi_1 \vee \Phi_2 \mid \Phi_1 ^\frown \Phi_2 \mid \exists x.\Phi$$

⋆ $^\frown$ is a binary modality for "chopping" an interval into two consecutive sub–intervals.

◇ $\phi_1 ^\frown \phi_2$ asserts that first ϕ_1 holds, over some interval, i, then ϕ_2 holds, over some interval, that starts when i ends.

- The modal operators extend IL:

$$\Phi ::= ... \mid \Box\Phi \mid \bigcirc\Phi$$

- $\Box\phi$ asserts: For all sub–intervals.
- $\bigcirc\phi$ asserts: For some sub–interval.
- The IL then comes with a suitable proof system.

Duration Calculus

- The syntax (and informal semantics) of the Duration Calculus starts on the basis of a consideration of *state variables*, SVar, and *state expressions*, say S over state variables:

$$S ::= 0 \mid 1 \mid P \mid \neg S_1 \mid S_1 \vee S_2$$

- *Temporal variables*, TVar other than ℓ, have a structure:

$$\int S$$

that determines lengths of intervals.

- Abbreviations are:

$\lceil\rceil$	holds on the point interval
$\lceil P \rceil$	holds on the interval $[b,e]$ if P has truth value **tt** (true) almost everywhere in $[b,e]$ and $b < e$.
$\bigcirc D$	holds on the interval $[b,e]$ if D holds on some sub–interval of $[b,e]$
$\Box D$	holds on the interval $[b,e]$ if D holds on any sub–interval of $[b,e]$
$D_1 \rightarrow D_2$	holds on an interval with D_1 holding on an inital sub–interval if D_2 holds on some sub–interval from the point, if any, where D_1 ceases to hold
$D_1 \leadsto t \leadsto D_2$	holds on an interval with D_1 holding on an initial sub–interval if D_2 starts holding within time t

- It is high time for an example:

$$
\begin{aligned}
&\text{Req1} \equiv \ell \leq 30 \Rightarrow \textstyle\int(\text{Gas} \wedge \neg\text{Flame}) \leq 4 \\
&\text{Req2} \equiv \lceil\neg\text{Heatreq}\rceil \Rightarrow (\lceil\rceil \rightsquigarrow 60 \rightsquigarrow \lceil\neg\text{Flame}\rceil) \\
&\text{Req3} \equiv \lceil\text{Heatreq}\rceil \Rightarrow (\lceil\rceil \rightsquigarrow 60 \rightsquigarrow \lceil\neg\text{Flame}\rceil) \vee \bigcirc\text{IgniteFail} \vee \bigcirc\text{FlameFail}
\end{aligned}
$$

The example above expresses three requirements to a gas burner:

- For safety, gas must never leak more than 4 seconds in any period of operation of at most 30 seconds.
- Heat–request off shall result in the flame being off after at most 60 seconds.
- Heat request (on) shall, after 60 seconds, result in gas burning, unless an ignite or flame failure has occurred.

I leave it to the listener to infer the type of the variables and a possible structure of the gas burner !

Question: *Where does DC fit into the railway example ?*

- Well, in railways there are many time (as well as safety) critical issues.
- The temporal ones cannot be expressed in RAISE.
- Hence we suggest to express them in DC (or in TLA$^+$).
- And then to relate the two different kinds of specifications.
- RAISE — to be covered next — has been extended with Timing and with DC.

- Since our large examle was expressed in RSL, we may wish to skip this and the next two slides.

RSL: RAISE Specification Language

- An RSL specification, without schemes, classes and objects, consists of
  - ⋆ a non–empty set of sort and **type** definitions,
  - ⋆ a possibly empty set of **variable** declarations,
  - ⋆ a possibly empty set of of **channel** declarations,
  - ⋆ a non–empty set of function (including predicate function) and other **value** definitions,
  - ⋆ and a possibly empty set of **axioms.**

- Sort and type definitions introduce named sets of **type**d value**s.**
- Sorts are further unspecified ‘types’ of value.
- Type definitions imply some structure of the typed values:
  - ⋆ Integers, reals, Booleans, characters,texts,
  - ⋆ sets, cartesians, lists, maps [definite definition set (and hence range) functions], and general functions.
- Declaration of **variable**s imply an imperative, ie., a state–based model.
- Declaration of **channel**s assume that some function definitions express *CSP*–like *input/output communication* between concurrent *processes.*
- Function **value** definitions express functions from (Cartesians of) typed values into such values.
- Axioms may constrain, ie., impose certain properties on sorts (ie., **type**s) or **value**s.
- Our railway example primarily exhibited sorts and axioms.

- We now "bundle" the above kind of RSL specifications:
  - ⋆ Definitions of types and values and declaration of variables and channels, into **class**es.
  - ⋆ We can name these classes obtaining possibly parameterised **scheme**s.
  - ⋆ And we can instantiate these into **object**s.
- As such schemes, classes and objects derive much from Clear and OBJ, ie. resemble certain features of CafeOBJ and CASL.
- An RSL specification — roughly speaking — denotes a possibly infinite set of algebras.
- The algebras map defined names of the specifications (schemes, types, values) into their trace/behaviour oriented denotation.

TLA$^+$: Temporal Logic of Actions

- The '+' in TLA+ stands for the extension of TLA (Temporal Logic of Actions), a linear–time temporal logic with classical set theory according to Zermelo–Fränkel.
- There are two levels (tiers) to TLA formulas.
- Besides a very basic, ie., rudimentary use set membership, the low level language of TLA is basically that of well–formed formulas in a first–order equational logic:
  - ⋆ Predicate and function symbols of various arities are use to form terms.
  - ⋆ Variables, say v, are introduced and predicate expressions allow the mentioning of "before" (assignement) and "after" (assignment) references to values of variables: v respectively v'.
  - ⋆ Variables denote states and lower level formulas describe state transitions.
  - ⋆ Higher level formulas add temporal operators.
  - ⋆ Transition predicates and functions (ie., actions) are given a precise (valuation) semantics in the usual way.

- Adding temporal operators, in a special way, to low level TLA formulas gives us the high level formulas of TLA.
  - ⋆ Let the $\Box$ operator designate the (usual) 'always' (or henceforth).
  - ⋆ If F is a formula, then $\Box F$ is a formula.
  - ⋆ Variables are either "rigid" (constant value) or "flexible" (assignment varying values).
  - ⋆ Let s designate a state function, ie., syntactically be a listing of assignable, ie., declared) variables, and let A be an action, then $\Box[A]_s$ is a formula — and roughly speaking asserts:
    - ⋄ $\Box[A]_s$ holds iff every state transition that does not leave s unchanged satisfies the action A.

- ⋆ Actions are not formulas.
- ⋆ If x_r is a rigid variable then $\exists x_r : F$ is a formula.
- ⋆ If x_f is a flexible variable then $\exists\!\!\exists x_f : F$ is a formula.
- ⋆ The difference between $\exists x_r : F$ and $\exists\!\!\exists x_f : F$ lies in their valuation.
- ⋆ The temporal operator expression $\Diamond F$ is defined by $\neg\Box\neg F$.
- ⋆ The formula $F \leadsto G$ (the holding of F eventually leads to the holding of G) is defined by $\Box(F \rightarrow \Diamond G)$.
- ⋆ The formula $\Diamond\langle A\rangle_s$ (defined as $\neg\Box[\neg A]_s$) asserts:
  - ⋄ Some transition satisfies A and changes the state (ie., the value of) s.

- Let us show an example.

module Queue

EXTENDS Sequences
CONSTANT Message
VARIABLES i, q, o

$\mathrm{NoMsg} \equiv \mathbf{CHOOSE}\ \mathrm{c} : \mathrm{x} \notin \mathrm{Message}$

$\mathrm{Init} \equiv \mathrm{q} = \langle\rangle \wedge \mathrm{i} = \mathrm{NoMsg} \wedge \mathrm{o} = \mathrm{NoMsg}$

$\mathrm{Enq(m)} \equiv \ \bullet \wedge \mathrm{i} \neq \mathrm{m}$
$\quad\quad \bullet \wedge \mathrm{i}' = \mathrm{m} \wedge \mathrm{q}' = \mathrm{Append(q,m)}$
$\quad\quad \bullet \wedge \mathrm{o}' = \mathbf{out}$

$\mathrm{Deq} \equiv \ \bullet \wedge \mathrm{q} \neq \langle\rangle$
$\quad\quad \bullet \wedge \mathrm{o}' = \mathrm{Head(q)} \wedge \mathrm{q}' = \mathrm{Tail(q)}$
$\quad\quad \bullet \wedge \mathrm{i}' = \mathrm{i}$

$\mathrm{Next} \equiv (\exists\ \mathrm{m} \in \mathrm{Message} : \mathrm{Eng(m)}) \vee \mathrm{Deq}$

$\mathrm{vars} \equiv (\mathrm{i,q,o})$

$\mathrm{FifoI} \equiv \mathrm{Init} \wedge \Box[\,\mathrm{Next}\,]_{vars} \wedge \mathbf{WF}_{vars}(\mathrm{Deq})$

$\mathbf{THEOREM} \Rightarrow \ \bullet \wedge \Box\ (\mathrm{q} \in \mathrm{Seq(Message)})$
$\quad\quad \bullet \wedge \Box\ [\,\mathrm{Deq} \Rightarrow \mathrm{o}' \neq \mathrm{o}\,]_{vars}$
$\quad\quad \bullet \wedge \forall\ \mathrm{m} \in \mathrm{Message} : \mathrm{i{=}m} \leadsto \mathrm{o{=}m}$

From TLA to TLA+

- TLA is extended with classical set theory into TLA$^+$. The set theory is Zermelo–Fränkel with choice.
- The term **Choose** $x : P$ interprets as:

$$[\![\mathbf{Choose}\ x : P]\!]^{\xi}_{s,t} \equiv \ \epsilon(\{d \mid \ [\![P]\!]_{\alpha_{s,t,\xi[d/x]}} = t)$$

- where s, t are state and ξ is a mapping of rigid variable names to their (constant) values, and where ϵ is Russels choice function.

$$\begin{array}{rcll}
\text{distributed union:} & \textsc{union}\ S & \equiv & \mathbf{Choose}\ M : \forall\, x : (x \in M \equiv\ \exists\, T \in S : x \in T) \\
\text{dyadic union:} & S \cup T & \equiv & \textsc{union}\ \{S, T\} \\
\text{subset:} & S \subseteq T & \equiv & \forall\, x : (x \in\ S \rightarrow\ x \in\ T) \\
\text{powerset:} & \textsc{subset}\ S & \equiv & \mathbf{Choose}\ M : \forall\, x : (x \in M \equiv\ x \subseteq\ S) \\
\text{comprehension}_1\text{:} & \{x \in\ S : P\} & \equiv & \mathbf{Choose}\ M : \forall\, x : (x \in\ M \equiv\ x \in S \wedge\ P) \\
\text{comprehension}_2\text{:} & \{t : x \in S\} & \equiv & \mathbf{Choose}\ M : \forall\, y : (y \in M \equiv\ \exists x \in S : y = t)
\end{array}$$

- From the above, and with TLA, one can build quite a collection of data types.

Some remarks on the Logic of TLA^+:

- Besides what has been mentioned above, the logic of TLA^+ comes "equipped" with inference rules over the full TLA^+.

Question: *Where does TLA^+ fit into the railway example ?*

- An answer quite similar to those at the end of our treatment of the Duration Calculus apply here as well.

VDM and Z

- No space in the proceedings for treating another of "my own babies": VDM (in addition to RAISE/RSL).
- Hence also no space in the proceedings for treating Z.
- Sorry.

Logicians contra Software Engineers

- With the rather definite risk of being ostrasized — having been invited to present this paper of my own choice to a group of mathematical logicians —
- I venture the following simplifying (quoted, Rushby 93) statements:[4]
- *Mathematical logicians were originally, and many are still, interested in providing a minimal and self–evident foundation for mathematics in general.*
- *Later, mathematical logicians were primarily interested in the technicalities of mathematical logic.*
- *Much of mathematical logic is set up for* **meta**–*mathematical reasons:*
  - ⋆ *to show that certain elementary concepts allow some parts of mathematics to be formalised,* ***in principle,***
  - ⋆ *and to support relatively simple proofs of properties such as soundness and completeness.*

[4] In this my confidence had been greatly strengthened by reading — from which we quote the above *slanted text parts.*

- *Formal methods in computer science, on the other hand, is/are concerned with formalising*
  - ⋆ *domains,*
  - ⋆ *requirements,*
  - ⋆ *software designs, and*
  - ⋆ *algorithms,*
- *and with developing formal proofs* ***in practice.***
- I continue quoting Rushby:

Thus, formal systems and approaches, that are very suitable for mathematicians, may not be so suitable as a foundation for formal methods in computing science.

- In the years to come we need to understand better this division of labour:
- Between mathematical logicians and computing scientists.

Conclusion

- Time has gone.
- See the paper for conclusions.

Acknowledgements

- I thank the Scientific and the Organising Committee Chairs:
  - ⋆ Professor Thanassis Tzouvaras
  - ⋆ Professor Lefteris Kirousis

  for — boldly — inviting me, and for enduring my many e–mails, etc.