"What is an Infrastructure ?" Towards an Informatics Answer

Dines Bjørner
Department of Computing Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Bldg.322, Richard Petersens Place
DK–2800 Kgs.Lyngby, Denmark

db@imm.dtu.dk; http://www.imm.dtu.dk/~db

Thursday June 19, 2003

Abstract

We briefly discuss the dogmas of a domain engineering oriented and a formal techniques based approach to software engineering. Then we try delineate the concepts of infrastructure and infrastructure components. Finally we hint at an abstract example work flow domain model: transaction script work flows. It is claimed that such are one of the core informatics characteristics of infrastructure components. The paper ends with some reflections on 10 years of UNU/IIST.

1 Some Software Engineering Dogmas

1.1 From Science via Engineering to Technology

The "ST" in UNU/IIST stands for software technology. But UNU/IIST seems, in the last 10 years, to have stood for an engineering basis for construction of software. The engineering basis was scientific, and was based on computer science. The approach to the construction of software was based on computing science — programming methodology. We saw it, and I believe UNU/IIST still sees it, this way: The engineer as "walking the bridge" between science and technology: Creating technology based on scientific insight; and, vice–versa, analysing technological artifacts with a view towards understanding their possible scientific contents. Both science and technology; both synthesis and analysis.

1.2 CS $\oplus$ CS $\oplus$ SE

Computer science, to me, is the study and knowledge of the artifacts that can "exist" inside computers: Their mathematical properties: Models of computation, and the underlying mathematics itself. Computing science, to me, is the study and knowledge of how to construct those artifacts: programming languages, their pragmatics, their semantics, including proof systems, their syntax; and the principles and techniques of use. The difference is, somehow, dramatic. Software engineering is the art, discipline, craft, science and logic of conceiving,

constructing, and maintaining software. The sciences are those of applied mathematics and computing. I consider myself both a computing scientist and a software engineer.

1.3 Informatics

Informatics, such as I saw it in the early 1990s, at UNU/IIST, was a combination of mathematics, computer & computing science, software engineering, and applications. Perhaps this is a way still to see it ? Some "sobering" observation: Informatics relates to information technology (IT) as biology does to bio—technology; $\mathcal{E}$tcetera ! The political (UN, Macau, PRC, *&c.*) world is, forever (?) caught by the syntax of "gadgets". UNU/IIST was steadfast in its focus on pragmatics and semantics.

1.4 A Triptych Software Engineering

1.4.1 The Dogma

The *Triptych Dogma:* Before software can be designed, we must understand the requirements. Before requirements can be expressed we must understand the domain. This then was a dogma — is it still ? Software engineering consists of the engineering of domains, engineering of requirements, and the design of software. In summary, and ideally speaking: We first *describe* the domain: $\mathcal{D}$, from which we *define* the domain requirements; from these and interface and machine requirements, ie. from $\mathcal{R}$, we *specify* the software design: $\mathcal{S}$. In a suitable reality we secure that all these are properly documented and related: $\mathcal{D}, \mathcal{S} \models \mathcal{R}$, when all is done !

In proofs of correctness of software ($\mathcal{S}$) wrt. requirements ($\mathcal{R}$) assumptions are often stated about the domain ($\mathcal{D}$). But, by domain descriptions, $\mathcal{D}$, we mean "much more" than just expressing such assumptions.

1.4.2 Some Issues of Domain Engineering

The Facets: To understand the application domain we must describe it. We must, I believe, describe it, informally (ie. narrate), and formally, as it is, the very *basics,* ie. the *intrinsics;* the *technologies* that *support* the domain; the *management & organisation* structures of the domain; the *rules & regulations* that should guide human behaviour in the domain; those *human behaviours:* the correct, diligent, loyal and competent work; the absent–minded, "casual", sloppy routines; and the near, or outright criminal, neglect. *&c.*

In [2] we go into more details on domain facets while our lecture notes (cum planned book [3]) brings the "full story".

The Evidence: How are we describing the domain ? We are *rough sketching* it, and *analysing* these sketches to arrive at *concepts.* We establish a *terminology* for the domain. We *narrate* the domain: A concise professional language description of the domain using only (otherwise precisely defined) terms of the domain. And we *formalise* the *narrative.* We then *analyse* the *narrative* and the *formalisation* with the aims of *validating* the *domain description* "against" domain stake–holders, and of *verifying* properties of the *domain description.*

On Documentation in General: In general there will be many documents for each phase[1], stage[2] and step[3] of development: Informative documents: Needs and concepts, development briefs, contracts, *&c.* Descriptive/prescriptive documents: Informal (rough sketches, terminologies, and narratives) and (formal models) analytic documents: Concept formation, validation, and verification. These sets of documents are related, and occur and re–occur for all phases. See for example our distinction, below, between what is elsewhere called: User requirements, vs. system requirements.

1.4.3 Some Issues of Requirements Engineering

How are we otherwise to formulate requirements ? We see requirements definitions as composed from three viewpoints: Domain, interface and machine requirements. We survey these.

Requirements are about the machine: The hardware and software to be designed.

Domain Requirements: Requirements that can be expressed solely with reference to, ie. using terms of, the domain, are called *domain requirements*. They are, in a sense, "derived" from the *domain understanding*. Thus whatever vagueness, non–determinism and undesired behaviour in the domain, as expressed by the respective parts of the domain *intrinsics, support technologies, management & organisation, rules & regulations,* and *human behaviour,* can now be constrained, if need be, by becoming requirements to a desirably performing computing system.

The development of domain requirements can be supported by a number of principles and techniques. *Projection:* Not all of the domain need be supported by computing — hence we project only part of the domain description onto potential requirements; *Determination:* Usually the domain description is described abstractly, loosely as well as non–deterministically — and we may wish to remove some of this loosenes and non–determinism. *Instantiation:* Typically domain rules & regulations are different from instance to instance of a domain. In domain descriptions they are abstracted as functions. In requirements prescriptions we typically design a script language to enable stake–holders to "program", ie., to express, in an easily computerisable form, the specific rules & regulations. *Extension:* Entities, operations over these, events possible in connection with these, and behaviours on some kinds of such entities may now be feasibly "realisable" — where before they were not, hence some forms of domain requirements extend the domain. *Initialisation:* Phenomena in the world need be represented inside the computer — and initialising computing systems, notably the software "state", is often a main computing task in itself, as is the ongoing monitoring of the "state" of the 'outside' world for the purpose of possible internal state (ie. database) updates. There are other specialised principles and techniques that support the development of requirements.

Interface Requirements: Requirements that deal with the phenomena shared between external users (human or other machines) and the machine (hardware and software) to be designed, such requirements are called *interface requirements*. Examples of areas of concern

[1]Domain, requirements and software design are three main phases of software development.

[2]Phases may be composed of stages, such as for example the domain requirements, the interface requirements and the machine requirements stages of the requirements phase, or, as another example, the software architecture and the program organisation stages of the software design phase.

[3]Stages may then consist of one or more steps of development, typically data type reification and operation transformation — also known as refinements.

for interface requirements are: Human computer interfaces (HCI, CHI), including graphical user interfaces (GUIs), dialogues, etc., and general input and output (examples are: Process control data sampling (input sensors) and controller activation (output actuator)). Some interface requirements can be formalised, others not so easily, and yet others are such for which we today do not know how to formalise them.

Machine Requirements: Requirements that deal with the phenomena which reside in the machine are referred to as *machine requirements.* Examples of machine requirements are: performance (resource [storage, time, etc.] utilisation), maintainability (adaptive, perfective, preventive, corrective and legacy–oriented), platform constraints (hardware and base software system platform: development, operational and maintenance), business process re–engineering, training and use manuals, and documentation (development, installation, and maintenance manuals, etc.).

1.4.4 Some Issues of Software Design

Once the requirements are reasonably well established software design can start. We see software design as a potentially multiple stage, and, within stages, multiple step process. Concerning stages one can identify two "abstract" stages: The software architecture design stage in which the domain requirements find an computable form, albeit still abstract. Some interface requirements are normally also, abstract design–wise "absolved", and the programme organisation design stage in which the machine requirements find a computable form. Since machine requirements are usually rather operational in nature, the programme organisation design is less abstract than the software architecture design. Any remaining interface requirements are also, abstract design–wise "absolved".

This finishes our overview of the triptych phases of software development.

1.5 Formal Techniques

A significant characteristics in our approach is that of the use of formal techniques: formal specification, and verification by proofs and by model checking. The area as such is usually — colloquially — referred to as "formal methods". By a method we understand a set of principles of analysis and for selecting techniques and tools in order efficiently to achieve the construction of an efficient artifact. By formal specification we mean description by means of a formal language: One having a formal semantics, a formal proof system and a formal syntax. In this paper we shall rather one–sidedly be illustrating just the specification side and not at all show any verification issues. And in this paper we shall rather one–sidedly also be using only one tool: The Raise Specification Language: RSL [11, 10].

2 On Infrastructures and their Components

UNU/IIST was placed in a UN + World Bank environment[4]. In that environment such terms as: infrastructure, self–reliance, and sustainable development, were part of the daily parlance. How was UNU/IIST to respond to this. It had to !

[4] Also known as the Bretton Woods Institutions.

2.1 The World Bank Concept of Infrastructure — A 1st Answer

One may speak of a country's or a region's infrastructure.[5] But what does one mean by that ?

2.1.1 A Socio–Economic Characterisation

According to the World Bank,[6] 'infrastructure' is an umbrella term for many activities referred to as 'social overhead capital' by some development economists, and encompasses activities that share technical and economic features (such as economies of scale and spill-overs from users to non-users).

Our interpretation of the 'infrastructure' concept, see below, albeit different, is, however, commensurate.

2.1.2 Concretisations

Examples of infrastructure components are typically: The transportation infrastructure sub–components (road, rail, air and water [shipping]); the financial services industry (banks, insurance companies, securities trading, etc.); health–care; utilities (electricity, natural gas, telecommunications, water supply, sewage disposal, etc.), etc. ?

2.1.3 Discussion

There are thus areas of human enterprises which are definitely included, and others areas that seem definitely excluded from being categorised as being infrastructure components. The production (ie. the manufacturing) — of for example consumer goods — is not included. Fisheries, agriculture, mining, and the like likewise are excluded. Such industries rely on the infrastructure to be in place — and functioning. What about the media: TV, radio and newspapers ? It seems they also are not part of the infrastructure. But what about advertising and marketing. There seems to be some grey zones between the service and the manufacturing industries.

2.2 The mid 1990's UNU/IIST Concept of Infrastructure — A 2nd Answer

UNU/IIST took[7] a more technical, and, perhaps more general, view, and saw infrastructures as concerned with supporting other systems or activities.

Software for infrastructures is likely to be distributed and concerned in particular with supporting communication of information, people and/or materials. Hence issues of (for example) openness, timeliness, security, lack of corruption, and resilience are often important.[8]

2.3 "What is an Infrastructure ?" — A 3rd Answer

We shall try answer this question in stages: First before we bring somewhat substantial examples; then, also partially, while bringing those examples; and, finally, in a concluding section, Section 4.2 of this paper. The answer parts will not sum up to a definitive answer !

[5] Winston Churchill is quoted to have said, during a debate in the House of Commons, in 1946: *... The young Labourite speaker that we have just listened to, clearly wishes to impress upon his constituency the fact that he has gone to Eton and Oxford since he now uses such fashionable terms a 'infra–structure' ...*

[6] Dr. Jan Goossenarts, an early UNU/IIST Fellow, is to be credited with having found this characterisation.

[7] I write "mid 1990's" since that is what I can vouch for.

[8] The above wording is due, I believe, to Chris George, UNU/IIST.

2.3.1 An Analysis of the Characterisations

The World Bank characterisation, naturally, is "steeped" in socio–economics. It implies, I claim, that what is characterised is well–functioning. It could, possibly, be criticised for not giving a characterisation that allowed one to speak of well-functioning, and of not so well–functioning infrastructures. It cannot be used as a test: Is something presented an infrastructure, or is it not ? And it begs the question: Can one decompose an infrastructure into parts, or as we shall call them, components ?

The UNU/IIST characterisation, naturally, is "steeped" in systems engineering. It seems we were more defining requirements to the business process engineering of an infrastructure (component), than the domain — which, as for the World Bank characterisation, assumes a concept of "good functionality."

We shall, despite these caveats, accept the two characterisations in the following spirit: For a socio–economically well–functioning infrastructure (component) to be so, the characterisations of the intrinsics, the support technologies, the management & organisation, the rules & regulations, and the human behaviour, must, already in the domain, meet certain "good functionality" conditions.

That is: We bring the two characterisations together, letting the latter "feed" the former. Doing so expresses a conjecture: One answer, to the question" *"What is an infrastructure"*, is, seen from the viewpoint of systems engineering, that it is a system that can be characterised using the technical terms typical of computing systems.

2.3.2 The Question and its Background

The question and its first, partial answer, only makes sense, from the point of view of the computer & computing sciences if we pose that question on the background of some of the achievements of those sciences. We mention a few analysis approaches. They are the denotational, the concurrency, the modal (incl. temporal) logic, the type/value, and the knowledge engineering approaches.

An important aspect of my answer, in addition to be flavoured by the above, derives from the *semiotics* distinctions between: *pragmatics, semantics,* and *syntax.* So we will also discuss this aspect below.

2.3.3 A Third Attempt at an Answer

A first concern of the socio–economics of infrastructures seems to be one of pragmatics: For society, through state or local government intervention, either by means of publicly owned, or by means of licensed semi–private enterprises, to provide infrastructure component means for "the rest of society": Private people and private (or other public) enterprises, to function properly. Depending on "the politics of the day" provision of such means may, or may not be state subsidised. So efficiency and profitability of such infrastructure components were sometimes not a main concern. The above observations certainly seems to have applied in the past.

With the advent of *informatics*, the confluence of computing science, mathematics (incl. mathematical modelling), and applications, the business process re–engineering of infrastructure components forces as well as enables a new way of looking at infrastructure components. We therefore recapitulate the UNU/IIST view of infrastructures.

Computing systems for infrastructures are distributed and concurrent, and are concerned with the flow of information, people, materials,and control, and the manipulation of the "flowed items".

Concepts like denotations, concurrency, types, logics (including modal logics), agents and speech acts, computational models, and semiotics (pragmatics, semantics and syntax) seems to offer a mind set associated with a vocabulary that "lifts" daily, short-range, and hence often short–sighted reasoning, and thus a framework for long–range thinking about necessary infrastructure process re–engineering.

So our "third try" at an answer to the question: *"What is an Infrastructure ?"*, is a rather unconventional one: An infrastructure, as seen from the point of view of informatics (mathematics $\oplus$ computing science $\oplus$ applications), is a challenge: A class of systems that we need characterise both from the point of view of socio–economics, and from the point of view of computing science, and to relate the two answers.

3 Work Flow Domains

We first motivate these seeming digressions: From, in Section 1, overviewing a software enineering paradigm, via, in Section 2, discussing the socio–economic as well as other meanings of the term 'infrastructure', to now, in Section 3, bringing an example of a domain description. It may puzzle some readers.

At UNU/IIST we had to be, and were glad to be involved with providing research and methods for the development of software for the support of infrastructure components.

Firstly it was natural for us, then, to ask such questions as: "What is a Railway ?", mathematically, that is, formally speaking. What is a "Financial Service System ?", etc. And we found ways of answering these questions, well, to tolerably well ! So it was obvious that we had to ask, and try answer the more general question: "What is an, or 'the' Infrastructure ?". Since answers to such questions as: "What is a computer progam" can be given in denotational, or other computer science terminology, we should, naturally think of railways and infrastructures also having such computer science attributes.

Secondly, to provide such answers we had to delve deep into the modelling, such as we knew how to do it, of example such domains. It seems that work flow systems, of various kinds, are at the core of infrastructure component systems. And it seems, when abstracting several rather different kinds of work flow systems, that transaction processing systems are at the core of either of these infrastructure component systems.

So the first and the second section now finds its first "reunion" — in this section — in trying to apply just a tiny fragment of software engineering (and then again it is only a tiny fragment of domain engineering) to provide the basis for subsequent answers. The larger setting of software engineering: With domains, requirements and software design, is necessary, we believe, in order not to loose sight of the larger picture, namely that, eventually, after answering esoteric, abstract questions, of providing software !

3.1 Work Flows and Transactions

We would have liked to exemplify three kinds of concrete work flow systems: (1) Electronic business: Buyers and sellers in the form of consumers, retailers, wholesalers, and producers. Agents and brokers acting on behalf of one side (buyer or seller), respectively both sides of these. Transactions like inquiry, quotation, order placement, order confirmation, delivery,

acceptance, invoicing, payment, etc. *&c.* (See [1].) (2) Health–care system: Citizens visiting medical doctors, pharmacies, clinical test laboratories, hospitals, etc. Medicine flowing from doctor to patient, from pharmacy to patient, etc. Patient medical journals flowing between the above "layers". *&c.* (3) Freight transport logistics. People sending and receiving fraight. Logistics firms arranging for the transport of freight. Transport companies: Railways, trucking firms, shipping companies, air cargo lines. Their vehicles: Trains, trucks, boats and air crafts. Transport networks: rail lines, road nets, shipping lanes and air corridors. Transport hubs: Train stations, truck depots, harbours and airports. The traffics: Trains, trucks, ships, air crafts. *&c.* All exemplify the movement of information, materials and control.

But we refrain — for lack of space !

Instead we illustrate some of the facets of a transaction script work flow example. You can interpret the transaction script work flow as an abstract E–business, an abstract health–care, or an abstract logistics system !

3.2 Transaction Scripts

3.2.1 The Problem

In domains 1–2–3 (see the previous section) tasks were carried out by a distributed set of activities either on potential or real trade (as for the electronic business example), or on patients (as for the health–care system), or on freight (as for the logistics example), The distributed set of operations were somehow effected by there being an actual or a virtual (a tacitly understood) protocol. We will now examine this notion of "protocol" further.

There are two issues at stake: To find a common abstraction, a general concept, by means of which we can (perhaps better) understand an essence of what goes on in each of the previously illustrated examples; and thus to provide a "common denominator" for a concept of work flow systems, a concept claimed to be a necessary (but not sufficient) component of "being an infrastructure".[9] We could now proceed to a slightly extended *discussion & analysis* of various issues that are exemplified by the previous three examples; but we omit such a *discussion & analysis* here — leaving it to a more vivid "class–room" interaction to do so. Instead we delve right into one outcome of, ie. one solution to, this *discussion & analysis*, respectively search for a *common abstraction, a general concept.*

3.2.2 Clients, Work Stations (Servers), Scripts and Directives

There are *client*s and there are *work station*s (servers). Clients initialise and *interpret scripts.* A script is a set of *time–interval* stamped collection of *directive*s. Interpretation of a script may lead a client to *visit* (ie. to *go to*) a work station. A client can at most visit one work station at a time. Thus clients are either *idle,* or *on their way* to or from a work station: Between being idle or visiting a previous work station. At a work station a client is being *handled* by the work station. Thus work stations handle clients, one at a time. That is, a client and a work station enter into a *"rendez vous"*, ie. some form of *co–operation.* Client/work station co–operation exhibits the following possible *behaviours:* A directive is *fetched* (thus *removed*) from the script. It is then being *interpreted* by the client and work station in unison.

[9] Railway systems, as are indeed all forms of transportation systems, are thought of as being infrastructure components, yet, in our past models of railway systems the work flow nature was somewhat hidden, somewhat less obvious.

A directive may either be one which prescribes one, or another, of a small set of *operation*s to take place — with the possible effect that, at operation completion, one or more directives have been added to the client script; or a directive prescribes that the client *goes on to* visit another work station; or a directive prescribes that the client be *released*. Release of a client sets the client free to leave the work station. Having left a work station as the result of a release directive "puts" the client in the idle state. In the idle state a client is free either to fetch only *go to* work station directives, or to add a *go to work station w* directive to its script, or to remain idle.

3.2.3 A Simple Model of Scripts

Formalisation of Syntax:

type
  T, Δ
axiom
  ∀ t,t′:T, ∃ δ:Δ • t′>t ⇒ δ = t′−t
type
  C, Cn, W, Wn
  S′ = (T × T) $\xrightarrow[m]{}$ D-**set**
  S = {| s:S′ • wf_S(s) |}
  D == g(w:Wn) | p(w:W,f:F) | release
  F′ = (C × W) → (W × C)
  F = {| f:F • wf_F(f) |}
value
  obs_Cn: C → Cn
  obs_S: C → S
  obs_Wn: W → Wn
  wf_S: S → **Bool**
  wf_S(s) ≡ ∀ (t,t′):(T×T) • (t,t′)∈ **dom** s • t≤t′
  wf_F: F → **Bool**
  wf_F(c,w) **as** (c′,w′)
  post obs_Cn(c)=obs_Cn(c′) ∧ obs_Wn(w)=obs_Wn(w′)

Annotations I: There are notions of (absolute) time (T) and time intervals (Δ). And there are notions of named (Cn, Wn) clients (C) and work stations (W). Clients possess scripts, one each. A script associates to (positively directed) intervals over (absolute) times zero, one or more directives. A directive is either a *go to*, or a *perform*, or a *release* directive. *Perform* directives specify a function to be performed on a pair of clients and work stations, leaving these in a new state, however constrained by not changing their names.

3.2.4 A Simple Model of Work Flow

Formalisation of Semantics — The Work Flow System:

type
  Cn, Wn

```
  CΣ, WΣ
  CΩ = Cn -m-> CΣ
  WΩ = Wn -m-> WΣ
value
  obs_S: CΣ → S
  remove: (T×T) × D → S → S
  add: (T×T) × D → S → S
  merge: S × CΣ → CΣ
  obs_CΣ: C → CΣ
  obs_WΣ: W → WΣ

  cω:CΩ, wω:WΩ, t0:T, δ:Δ

  sys: Unit → Unit
  sys() ≡ ‖{client(cn)(t0)(cω(cn))|cn:Cn}‖(‖{ work_station(wn)(t0)(wω(wn))|wn:Wn})
```

Annotations II: Clients and work stations have (ie. possess) states. From a client state one can observe its script. From a script one can remove or add a time interval stamped directive. From the previous notions of clients and work stations one can observe their states.[10] $\mathsf{c}\omega$, $\mathsf{w}\omega$ t_0, and δ represent initial values of respective types — needed when intialising the system of behaviours. A work flow system is now the parallel combination of a number (# Cn) of clients and a number (# Wn) of work stations, the latter all occurring concurrently.

Formalisation of Semantics — Clients:

```
channel
  { cw[cn,wn] | cn:Cn, wn:Wn } M

value
  client: cn:Cn → T → CΣ → in,out { cw[cn,wn] | wn:Wn } Unit
  client(cn)(t)(cσ) ≡ c_idle(cn)(t)(cσ) ⌈⌉ c_step(cn)(t)(cσ)

  c_idle: Cn → T → CΣ → Unit
  c_idle(cn)(t)(cσ) ≡ let t′:T•t′>t in client(cn)(t′)(cσ) end

  c_step: cn:Cn → T → CΣ → in,out { cw[cn,wn] | wn:Wn } Unit
```

Annotations III: Any client can, in principle, visit any work station. Channels model this ability. A client is either idle or potentially visiting a work station (making one or more transaction steps). The client makes the (ie. a non–deterministic internal) choice, whether idle or potential action steps. To "perform" an idle "action" is to non–deterministically advance the clock.

[10] The two notions may eventually, in requirements be the same. In the domain it may be useful to make a distinction.

Formalisation of Semantics — Clients Continued:

```
c_step(cn)(t)(cσ) ≡
  let s = obs_S(cσ) in
  if ∃ (t′,t″):(T×T),g(wn):D • (t′,t″) ∈ dom s ∧ t′≤t≤t″ ∧ g(wn) ∈ s(t′,t″)
    then
      let (t′,t″):(T×T),g(wn):D • (t′,t″) ∈ dom s ∧ t′≤t≤t″ ∧ g(w) ∈ s(t′,t″) in
      let cσ′ = remove((t′,t″),g(wn))(cσ) in
      let (t‴,cσ″) = c2ws_visit(t′,t″)(cn,wn)(t)(cσ′) in
      client(cn)(t‴)(cσ″) end end end
    else
      let t‴:T • t‴ = t + δ in
      client(cn)(t‴)(cσ) end
  end end

c2ws_visit: (T×T×cn:Cn×wn:Wn)→T→CΣ→in,out {cw[cn,wn′]|wn′:Wn} (T×CΣ)
c2ws_visit(t′,t″)(cn,wn)(t)(cσ) ≡ cw[cn,wn]!((t′,t″),cn,t,cσ);[]{cw[cn,wn′]?|wn′:Wn}
```

Annotations IV: From a client state we observe the script. If there is a time interval recorded in the script for which there is a goto directive then such a time interval and goto directive is chosen: removed from the script, and then a visit is made, by the client to the designated work station, with this visit resulting in a new client state — at some "later" time. Otherwise no such visit can be made, but the clock is advanced. A work station visit starts with a rendez–vous initiated by the client, and ends with a rendez–vous initiated by the work station.

Formalisation of Semantics — Work Stations:

```
work_station: wn:Wn → WΣ → in,out { cw[cn,wn] | cn:Cn } Unit
work_station(wn)(wσ) ≡
  let ((t′,t″),cn,t‴,cσ) = []{cw[cn,wn]?|cn:Cn} in
  let (t⁗,(sσ′,wσ′)) = w_step((t′,t″),cn,t‴,(cσ,wσ)) in
  cw[cn,wn]!(t⁗,sσ′) ;
  work_station(wn)(wσ′) end end

w_step: (T×T) → wn:Wn → (CΣ×WΣ) → in,out { cw[cn,wn] | cn:Cn } Unit
w_step((t′,t″),(cn,wn),t‴,(cσ,wσ)) ≡
  let s = obs_S(cσ) in
  if s={} then (t‴,(cσ,wσ))
    else assert: (t′,t″) ∈ dom s
      let d:D • d ∈ s(t′,t″) in
      case d of
        p(wn,f) →
          let (t⁗,(sσ′,wσ′)) = act(f,t⁗,(sσ′,wσ′)) in
          let sσ″ = remove((t′,t″),p(wn,f))(sσ′) in
          w_step((t′,t″),(cn,wn),t⁗⁗,(sσ″,wσ′)) end end
```

```
        release →
            let sσ′ = remove((t′,t″),p(wn,f))(sσ) in
            (t‴,(cσ′,wσ)) end,
        _ → (t‴,(cσ,wσ))
end end end end
```

Annotations V: Each work station is willing to engage in co–operation with any client. Once such a client has been identified (cn, cσ), a work station step can be made. If the client script is empty no step action can be performed. A work station step action is either a function performing action, or a release action. Both lead to the removal of the causing directive. Script *go to* directives are ignored (by work station steps). They can be dispensed by client steps. Function performing actions may lead to further work station steps.

3.2.5 Discussion

We have sketched a semi–abstract notion of transaction flow. A syntactic notion of directives and scripts have been defined. And the behavioural semantics of scripts as interpreted by clients and work stations. We emphasize that the model given so far is one of the domain. This is reflected in the many non–deterministic choices expressed in the model, and hence in the seemingly "erratic", unsystematic and not necessarily "exhaustive" behaviours made possible by the model. We shall comment on a number of these. See the client behaviour: Whether or not a client is step is possible, the client may choose to remain idle. See the client idle behaviour: The client may choose to remain idle for any time interval, that is "across" time points at which the script may contain directives "timed" for action. Now we turn to the client step behaviour. The purpose of the client step behaviour is to lead up to a client to (2) work station visit: Several 'goto work station' directives may be prescribed to occur sometime during a time interval "surrounding" the "current" time t of the client: $t' \leq t \leq t''$. Which one is chosen is not specified. In fact, one could argue that we are over–specifying the domain. A client may choose to go to a work station ahead of time: $t < t' \leq t''$. or late: $t' \leq t'' < t$. We leave such a domain "relaxation" as an exercises to the reader. If there are no selectable 'goto work station' directive, time (t) is stepped up by a fixed amount, but, again, one could choose any positive increment, but that would make no difference as it would just "reduce" (correspond) to the client idle behaviour. The client to (2) work station visit (c2ws_visit) behaviour models the interface between *client*s and *work station*s as seen from the *client* side. That "same" interface as seen from the side of *work station*s is modelled by the two formula lines surrounding the formula line in which the 'work station step' behaviour is invocated. We now turn to work station step behaviour. This is the behaviour "where things get done !". The behaviours described above effected the flow. Now we describe the work. And the work is done by performing functions. Here it should be recalled that when a client interacts with a work station both their states are "present". This is amply illustrated in the work station step behaviour. The functions to be performed apply to both client and work station states, and may affect both.

If the script is empty nothing more can be done — so we are finished. If the script is not empty then we can assert that the work station step time interval argument is one for which an entry can be, and is, selected from the script — non–deterministically. That entry can (thus) be either of several: It can be a perform directive aimed at the present work station —

in which case the designated function is acted upon, the directive is removed from the script, and another step is encouraged. It can be a release directive — in which case the client is released, becoming an unengaged client again after the release directive has been removed. Or it can be any other directive (other perform directives, aimed at other work stations, or go to directives) — in which case the client is likewise "released", but the directive is not removed. Observe the looseness of description. Besides including all the possibly desirable behaviours, the full model above also allows for such behaviours as could be described as being sloppy, delinquent, or even outright criminal. This concludes our sketch model of transaction scripts and their intended work flow.

4 Conclusion

4.1 Summary and Discussion

We have tried to conjure an image of a notion of infrastructure components. We have brought forward both a question and a number of fragments of concurrency and type/value models of such infrastructure components. And we have tried encircle the problem: Namely trying to answer the question *"What is an infrastructure ?"* by mentioning claimed engineering disciplines of software development: Denotational semantics, process algebras (concurrency), type/value systems, logics, including modal logics, agents and linguistics.

Our attempt at "decomposing" development of software into "featuring" denotational, concurrency, type/value, knowledge and other engineering considerations is, somehow, orthogonal (read: Complementary to) to Michael Jackson's work on *Problem Frames* [12].

An Apology: It is lamentable that my examples did not illustrate uses of other than `RSL` [10]. It ought also have contained examples of uses of one or another `Duration Calculus` [5, 14, 8, 4, 9, 7, 6]. Especially since I brought an example which excudes temporalities. I really apologise.

4.1.1 Infrastructure Models and Abstract Specification

UNU/IIST, had to address issues of developing countries, and newly industrialised countries, and thus had to address issues of (i) infrastructures, (ii) self-sufficiency and (iii) self–reliance. We found that we could do so, believably, with respect to all three facets mentioned above (i–ii–iii), by applying the dogmas of: domain engineering, and abstract (hence: Formal) specification. And we found that our Fellows had little problem in learning and practising this ! Myths about so–called "Formal Methods" were — I should say —- decisively dispelled.

4.2 "What is an Infrastructure ?"

Perhaps the question is an ill–posed question ? One that does not make sense ! Are the infrastructure components so complex, anyway, as to escape simple characterisations ? Perhaps ! Do they not, these components, encompass the whole spectrum of all the applications to which we put computers ? Perhaps ! *&c.* We shall, persist, however, and try a fourth attempt at an answer. One that seems "a cop out", an "escape from under the rug !"

4.2.1 A Fourth Answer

An infrastructure is a collection of infrastructure components. There is synchronisation and communication between and within the components. The transaction script example is claimed to illustrate some facets of this.

An infrastructure component is a language: The professional, specialised jargon language spoken by professionals and users of the infrastructure component. We have, in Section 3.1, mentioned several such languages: The language of "the market"; the language of logistics; the language of health–care, and the language of transaction scripts and directives; *&c.*

Through the transaction script abstraction of work flow systems we have modelled verbs of these languages in terms of behaviours over states and events. So infrastructure components are seen as "computing systems" although they are not necessarily computable !

4.2.2 A Possible Impact of Computing Science upon Infrastructures

If, what we are saying above, has any relevance, then it is perhaps this: That in future business process re–engineering (BPR) of infrastructure components the BPR engineer may be well served in being fluent in — and in using — the kind of informatics and computing science concepts exemplified by this paper.

It is all a matter of language !

4.3 10 Years of UNU/IIST

Having "founded" UNU/IIST, of course, makes my next statements rather biased. I believe that UNU/IIST — exactly in spanning computer and computing science with software engineering (along the lines of denotational, concurrency, temporal and type/value engineering) was able to contribute (i) socio–economically, helping its "client" countries in easening their way towards software reliance and self–sufficiency, (ii) "informatically" to better understanding problems of infrastructure component computing systems development, (iii) programming methodologically by researching and developing principles and techniques for development of software for infrastructure components, and (iv) scientifically in providing firmer theoretical bases for the development of real–time, embedded, safety critical systems. Macau, in the 1990s, was not like Florence of the Renaissance. For that to have been the case we needed even more generous support from our sponsors — and Mr. Stanley Ho and others are not the Medicis of our day — and Macau was not exactly at the science cross–roads. But I think we did rather well in comparison.

When I started in Macau, my dear friends, at the Academy and at universities in China, asked: *"How big will the Institute be; how many staff ?"*. When I answered, casually, and truthfully: *"Oh, I guess, some 6–8 scientists, some 6 administrative staff and some 12–24 Fellows !"*, they rather immediately lost interest. Big was important. When I left after five years we had been far more productive in science and, to some extent in advanced engineering, than most of their departments. I believe there is no secret here: We were, and they still are, two well–fitted, harmonious groups, both understanding the didactics of one anothers' fields and disciplines; both supporting each other; and in a well-defined area. We did not, as do usual departments, have to cover "the world". Small, if not 'important' in the eyes of politicians, can be beautiful.

Let us wish that UNU/IIST can continue along its course: improving here and there, adjusting here and there, diversifying just a bit, not too much. *"If it works, don't fix it,"* the

saying goes. It works.

4.4 Acknowledgements

UNU/IIST, in my days, owed its successes to several groups of people.

To my colleagues at the IBM Vienna Laboratory of the early 1970s, when VDM was first conceived: To the late Hans Bekič, to Peter Lucas, Kurt Walk, Cliff Jones and others, and to the visitors at IBM: Dana Scott, John Reynolds, and many others.

To the Board Members of UNU/IIST with whose much appreciated support we were able to "fight" some myths. Perhaps that Board did not know it, but they were really of immense help. A delight to work for.

More generally, to the computing scientists who have inspired us in what we had propagated: The members of IFIP WG 2.1, WG 2.2 and WG 2.3 — Manfred Broy, Sir Tony Hoare, Michael Jackson, Cliff Jones, Jayadev Misra, Carroll Morgan, David Parnas, Amir Pnueli, Natarajan Shankar, Douglas R. Smith, and several of whom came to visit us at Macau (Egidio Astesiano, Hans Langmaack, J S. Moore, Wlad Turski, etc.). As well as to many others.

To the devoted and loyal staff and Fellows of UNU/IIST:

To the lovely ladies of the administrative staff who now for many, many years have endured these strange scientists and their fellows, who have ensured our daily, smooth operations, and who have stood by, in physical as well as mental typhoons; to the 77 Fellows who had visited UNU/IIST by June 1997; and to the professional staff: To Mrs. Margaret Stewart, my Financial & Administrative officer, and to such wonderful scientists as: Zhou Chaochen, Søren Prehn, Chris George, Dang Van Hung, Xu Qi Wen, Tomasz Janowski, Richard Moore, and Kees Middelburg — while I was in charge. What more can one want ?

My most emotional thanks, perhaps, goes to Zhou Chao Chen: Thanks for your readiness to take charge, thanks for all the loyal support during the "mental typhoons", and thanks for your wise and wonderful way of continuing UNU/IIST. Like me, You will be very proud of what has been achieved here. Thanks.

I also think it appropriate here to commemorate the memory of the late Dr. António Rodrigues Juniór, the Macau Foundation President, who, on behalf of the Macau Government, helped UNU/IIST through many, many years, but whose untimely passing away saddened us all deeply. *God Bless his Soul.*

4.5 Bibliographical Notes

A book has just been published: *Specification Studies in RAISE.* It is edited by Chris George, Tomasz Janowski, Richard Moore, and Dan Van Hung. It is published, early 2002, in the Springer–Verlag UK FACT series [13]. It contains so many relevant papers and references that the below should suffice.

References

[1] Dines Bjørner. Domain Models of "The Market" — in Preparation for E–Transaction Systems. In *Practical Foundations of Business and System Specifications (Eds.: Haim Kilov and Ken Baclawski)*, page 34 pages, The Netherlands, December 2002. Kluwer Academic Press.

[2] Dines Bjørner. *"What is a Method ?" — An Essay of Some Aspects of Software Engineering*, chapter 9, pages 175–203. Monographs in Computer Science. IFIP: International Federation for

Information Processing. Springer Verlag, New York, N.Y., USA, 2003. Programming Methodology: Recent Work by Members of IFIP Working Group 2.3. Eds.: Annabelle McIver and Carrol Morgan. .

[3] Dines Bjørner. *The SE Book: Principles and Techniques of Software Engineering*, volume I: Abstraction & Modelling (750 pages), II: Descriptions and Domains (est.: 500 pages), III: Requirements, Software Design and Management (est. 450 pages). [Publisher currently (June 2003) being negotiated], 2003–2004.

[4] Zhou Chaochen. Duration Calculi: An Overview. In *Proceedings of Formal Methods in Programming and Their Applications, D. Bjørner, M Broy, and I.V. Pottosin (Eds.)*, pages 256–266. LNCS 735, Springer-Verlag, 1993.

[5] Zhou Chaochen, C.A.R. Hoare, and A.P. Ravn. A Calculus of Durations. *Information Processing Letters*, 40(5):269–276, 1991.

[6] Zhou Chaochen, Dang Van Hung, and Li Xiaoshan. A duration calculus with infinite intervals. In *Fundamentals of Computation Theory, Horst Reichel (Ed.)*, pages 16–41. LNCS 965, Springer-Verlag, 1995.

[7] Zhou Chaochen, Zhang Jingzhong, Yang Lu, and Li Xiaoshan. Linear duration invariants. In *Formal Techniques in Real-Time and Fault-Tolerant Systems, H. Langmack, W.-P. de Roever, and J. Vytopil (Eds.)*, pages 86–109. LNCS 863, Springer-Verlag, 1994.

[8] Zhou Chaochen, A.P. Ravn, and M.R. Hansen. An extended duration calculus for hybrid systems. In R.L. Grossman, A. Nerode, A.P. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 36–59. Springer-Verlag, 1993.

[9] Zhou Chaochen and Li Xiaoshan. A mean value calculus of durations. In A.W. Roscoe, editor, *A Classical Mind: Essays in Honour of C.A.R. Hoare*, pages 431–451. Prentice Hall International, 1994.

[10] Chris George, Peter Haff, Klaus Havelund, Anne Haxthausen, Robert Milne, Claus Bendix Nielsen, Søren Prehn, and Kim Ritter Wagner. *The RAISE Specification Language*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1992.

[11] Chris George, Anne Haxthausen, Steven Hughes, Robert Milne, Søren Prehn, and Jan Storbank Pedersen. *The RAISE Method*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1995.

[12] Michael A. Jackson. *Problem Frames — Analysing and structuring software development problems*. ACM Press, Pearson Education. Addison–Wesley, Edinburgh Gate, Harlow CM20 2JE, England, 2001.

[13] Hung Dang Van, Chris George, Tomasz Janowski, and Richard Moore, editors. *Specification Case Studies in RAISE*. FACIT: Formal Approaches to Computing and Information Technology. Springer–Verlag, April 2002. ISBN 1-85233-359-6.

[14] Liu Zhiming, A.P. Ravn, E.V. Sørensen, and Zhou Chaochen. A probabilistic duration calculus. In H. Kopetz and Y. Kakuda, editors, *Responsive Computer Systems*, volume 7 of *Dependable Computing and Fault-Tolerant Systems*, pages 30–52. Springer Verlag Wien New York, 1993.

