SEAMAIL

Newsletter from Software Engineers Association

Vol. 13, Number 5 May, 2002

目　次

ソフトウェア技術者協会

Software Engineers Asociation

ソフトウェア技術者協会 (SEA) は，ソフトウェアハウス，コンピュータメーカ，計算センタ，エンドユーザ，大学，研究所など，それぞれ異なった環境に置かれているソフトウェア技術者または研究者が，そうした社会組織の壁を越えて，各自の経験や技術を自由に交流しあうための「場」として，1985年 12月に設立されました．

その主な活動は，機関誌 SEAMAIL の発行，支部および研究分科会の運営，セミナー／ワークショップ／シンポジウムなどのイベントの開催，および内外の関係諸団体との交流です．発足当初約 200人にすぎなかった会員数もその後増加し，現在，北は北海道から南は沖縄まで，500 余名を越えるメンバーを擁するにいたりました．法人賛助会員も 25社を数えます．支部は，東京以外に，関西，横浜，名古屋，九州，広島，東北の各地区で設立されており，その他の地域でも設立準備をしています．分科会は，東京，関西，名古屋で，それぞれいくつかが活動しており，その他の支部でも，月例会やフォーラムが定期的に開催されています．

「現在のソフトウェア界における最大の課題は，技術移転の促進である」といわれています．これまでわが国には，そのための適切な社会的メカニズムが欠けていたように思われます． SEA は，そうした欠落を補うべく，これからますます活発な活動を展開して行きたいと考えています．いままで日本にはなかったこの新しいプロフェッショナル・ソサイエティの発展のために，ぜひとも，あなたのお力を貸してください．

ＳＥＡＭＡＩＬ Ｖｏｌ．１３，Ｎｏ．５ ２００２年５月３１日発行 編集人 岸田孝一
発行人 ソフトウェア技術者協会（ＳＥＡ）
〒１６０-０００４ 東京都新宿区四谷３－１２ 丸正ビル５Ｆ
T: 03-3356-1077 F: 03-3356-1072 sea@sea.or.jp
印刷所 有限会社 錦正社 〒１３０-００１３ 東京都墨田区錦糸町４－３－１４
定価 ５００円 （禁無断転載）

編集部から

☆

この号は当初，２月に開催された UML Workshop のレポートを特集する予定でしたが，６月の総会に併設して行われる特別 Forum の招待講師 Dines Bjorner 先生から，参考資料として力のこもった長論文が送られてきましたので，Forum に参加できない方々の便宜も考え，全ページを割いて載せることにしました．

☆☆

Bjorner 先生は，形式仕様言語 VDM の開発者であり，ヨーロッパにおける Formal Approach の普及・実践活動の中心人物としての活躍は，すでに SEA 会員のみなさんも御存知の通りです．

☆☆☆

また，数年前まで，マカオに設立された国連大学・国際ソフトウェア技術研究所 (UNU/IIST) の初代所長として，開発途上国に対する先進ソフトウェア技術の導入を目指して，精力的名活動を展開されました．

☆☆☆☆

SEA との関わりでは，1995年以降，中国での国際会議 ISFST やその他 ICSE 併設ワークショップの共同開催など，いろいろと御協力いただきました．

☆☆☆☆☆

今回寄稿いただいた論文は，さきごろの金融システム・トラブルでも明かになったような IT 社会におけるインフラストラクチャの構築および保全にかかわる諸問題に関して，計算機科学やソフトウェア工学がどのような形で貢献すべきかを具体的に論じた力作です．じっくりとお読みください．

☆☆☆☆☆☆

Informatics of Infrastructures*
A Future for Computing Science

Dines Bjørner
Computer Science and Engineering
Informatics and Mathematical Modelling
Technical University of Denmark
DK–2800 Kgs. Lyngby, Denmark
E–Mail: db@imm.dtu.dk

14th of May 2002

Abstract

After some introductory characterisations of *computer science, computing science,* and (computing systems cum) *software engineering,* 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.* We illustrate the concept of infrastructure by hinting at some concrete and abstract domain models of infrastructure components: Railway systems, electronic commerce, logistics, health–care, and transaction script work flows; and at two models of varieties of views of 'information' (type/value) entities: Graphical user interface (GUI) and relational database systems, and document systems.

The informal and formal examples of the paper are mere sketches. They serve to indicate somewhat uncommon aspects of what has to be dealt with in domain models of infrastructure component systems. And they serve to indicate that such models are composed from diverse, yet logically related concepts. Some such examples are teasers, some contain, perhaps, an eye– or mind–opener.

The paper ends with some reflections on rôles of *semiotics: Pragmatics, semantics,* and *syntax;* of *method* and *methodology principles, techniques* and *tools,* in particular such which can be grouped under general conceptualisations such as *property–* and *model–orientedness, denotational* versus *computational* models, *hierarchical* and *compostional model–building* and *–presentation,* models of *time, space* and *time/space, configurations* as compositions in a spectrum between *contexts* and *states,* etc., *domain abstraction* and *modelling* of *domain attributes, stake–holder perspectives,* and *domain facets, requirements projection, instantiation, extension,* and *initialisation,* and *software design* issues such as *architecture, component design, modularisation* (*object–orientedness*), etc.

The aims of the paper are to present an overview of programming methodological issues supported by a number of illustrative example hints, so as to better achieve the objectives of the paper which are to suggest that proper professional education and training in informatics is based on courses with a fair selection of computer science topics, and with a heavy emphasis on topics in computing science cum programming methodology — cum software engineering; to suggest that "formal methods" is not a course one gives in separation from all other informatics courses, but that 'formal techniques' are an integral

*Extended text of a (shorter) talk presented at SEA Seminar, Tokyo, Monday June 17th, 2002

part of all computing science and software engineering courses; and to suggest that maybe the borders between AI and software engineering constitute un–natural divisions.

Contents

1 Some Software Engineering Dogmas

τ:9
lisboa/intro

1.1 CS ⊕ CS ⊕ SE

τ:15

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; computing systems: Operating systems, database management systems, data communication systems, *&c.*, and applications — such as we shall illustrate some today. The difference, between computer and computing science, is, somehow, dramatic.

τ:10

Software engineering, to me, spans domain engineering, as we shall soon characterise it, requirements engineering, and software design. And: 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.

1.2 Informatics

τ:11

Informatics, such as I see it us a combination of: Mathematics, computer & computing science, software engineering, and applications. Some "sobering" observation: Informatics relates to information technology (IT) as biology does to bio—technology; *&c*tcetera !

1.3 A Triptych Software Engineering

τ:12

1.3.1 The Dogma

τ:20

The *Triptych Dogma:* Before software can be designed, we must understand the requirements. Before requirements can be expressed we must understand the (application) domain.

Software engineering thus consists of the engineering of domains, engineering of requirements, and the design of software. Software development, to us, encompasses all three.

τ:13

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.

1.3.2 Some Issues of Domain Engineering

τ:14

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 works; the absent–minded, "casual", sloppy routines; and the near, or outright criminal, neglect. *&c.*

In [1] we go into more details on domain facets while in the planned [2] we present a more comprehensive view of domain engineering. Finally our lecture notes (cum planned book [3]) brings the "full story".

τ:15

The Evidence: We *inform about* the domain: Present a not necessarily descriptive synopsis of it, emphasising, typically, the pragmatics, and the needs and ideas of the domain that might lead to computing support.

We *describe* the domain: We *rough sketch* it, and *analyse* the sketches to arrive at *domain 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
τ:16 of the domain. And we *formalise* the *narrative.*

We *analyse* the *narrative* and the *formalisation* with the aims of: *validating,* "against" domain stake–holders, and *verifying* properties of, the *domain description.*

τ:17

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, ter-
τ:18 minologies, 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.

1.3.3 Some Issues of Requirements Engineering

τ:19

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

We see requirements prescriptions as composed from three viewpoints: Domain, interface and machine requirements.

We now survey these.

τ:20

[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 component design 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.

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 principles and techniques of *projection:* Not all of the domain need be supported by computing — hence we project only part of the domain description onto potential requirements; *instantiation:* Usually the domain description is described abstractly, loosely as well as non–deterministically — and we may wish to remove some of these properties; *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; and *initialisation:* Phenomena in the world need be represented inside the computer — and initialising computers 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 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. We shall later, in Section 4.3 give an example of a GUI prescription.

Machine Requirements: Requirements that deal with the phenomena which reside in the machine are referred to as *machine requirements.* Examples of concerns 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.).

1.3.4 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 component design stage in which the machine requirements find a computable form.

Since machine requirements are usually rather operational in nature, the programme component 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.

1.4 Formal Techniques

A significant characteristics in our approach is that of the use of formal techniques: formal specification, verification & model checking.

1.4.1 Method

τ:28

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 a specification 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
τ:29 rather one–sidedly also be using only one tool: The Raise Specification Language: RSL [4, 5].

A method can never be formal: The principles for selecting techniques and tools, and the principles of concept analysis cannot be formalised, let alone mechanised. Humans perform these tasks.

Quite a considerable set of techniques and tools can be formalised. So we prefer the term 'formal techniques: Specification and verification'.

1.4.2 Methodology

τ:30

By methodology we understand the study and knowledge of (varieties of) methods, in particular techniques and tools: Petri nets, VDM (VDM–SL), Z, RAISE (RSL), B, etc., CSP, State–charts (Statemate), etc., CASL, Cafe–OBJ, Maude, etc. even "UML" !

2 On Infrastructures and their Components

τ:31
lisboa/infra

In the period 1991–1997 I was founding and first director of UNU/IIST: The UN University's International Institute for Software Technology, located, then as now, in Macau, near Hong
τ:32 Kong and Canton. My view of the infrastructure concept arose then. 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 ! And, I claim, we "informaticians" are perhaps best at trying to understand the infrastructure concept.

2.1 The World Bank Concept of Infrastructure

τ:33

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

[4] Also known as the Bretton Woods Institutions.

[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'* ...

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.); and perhaps also education, 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 UNU/IIST Concept of Infrastructure

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 ?"

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 5.2 of this paper. The answer parts will not sum up to a definitive answer !

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

[6] Dr. Jan Goossenarts, an early UNU/IIST Fellow, is to be credited with having found this characterisation.
[7] In the "mid 1990's" since that is what I can vouch for.
[8] The above wording is due, I believe, to Chris George, UNU/IIST.

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
7:39 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
7:40 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
7:41 achievements of those sciences. We select a few analysis approaches. These are aspects of denotational, concurrency, type/value, and knowledge engineering approaches, as well as a computer science approach.

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.

7:42

Denotational Engineering: In the *denotational engineering* view we associate to syntactic entities a meaning expressed, usually as a mathematical function. Normally we would expect the homomorphism lemma to apply: The meaning, $\mathcal{M}$, of a simple, "atomic" entity, a, is a simple function, $\mathcal{M}(a) \equiv \mathcal{F}(a)$. The meaning, $\mathcal{M}$, of a composite entity, $(c_1, c_2, \ldots, c_n)$, is a function, $\mathcal{H}$, of the meaning of the parts $\mathcal{M}(c_1, c_2, \ldots, c_n) \equiv \mathcal{H}(\mathcal{M}(c_1), \mathcal{M}(c_2), \ldots, \mathcal{M}(c_n))$.
7:43 One may say that the *denotational engineering* view entails a model oriented view. For mundane application domains such as railways, electronic commerce, health–care, logistics, etc., how do we apply the principle of *denotational engineering* ? Well, we first look out for suitable semantic types for the phenomena that we observe (viz., Train traffic, traces of behaviours, etc.) (as associated with syntactic phenomena, ie. types (viz., time tables, bill–of–ladings (way bills), etc.)). Then we device of simple, primitive operations on semantic values. Then we device of suitable syntactic abstractions. And finally we express the meaning of the syntactic entities in terms of compositions of primitive operations.

7:44

Concurrency Engineering: In a *concurrency engineering* view, not necessarily 'the' view, certain phenomena are seen, are abstracted as concurrently progressing and communicating processes, very much in the sense of CSP. [6, 7, 8, 9] Processes — simple ones — are sequential: Effecting, one–by–one changes to a process local state, while, now–and–then synchronising with other processes, performing, as–it–were, "rendez–vous" with these and (usually one way)

communicating information. "Rendez–vous" model events. Similar characterisations can be given for concurrency modelling using `Petri Nets`. Processes $\mathcal{P}$ — not so simple ones — may "split" (||) into parallel ones: $\mathcal{P} = \mathcal{P}_1 \parallel \mathcal{P}_2 \parallel \ldots \parallel \mathcal{P}_n$, or may decide (non–deterministic external choice, []), depending on external events, between either of n alternative processes: $\mathcal{P} = \mathcal{P}_1 \,[]\, \mathcal{P}_2 \,[]\, \ldots \,[]\, \mathcal{P}_n$, or may decide (non–deterministic internal choice, $\sqcap$), depending on internal "events", between either of n alternative processes: $\mathcal{P} = \mathcal{P}_1 \sqcap \mathcal{P}_2 \sqcap \ldots \sqcap \mathcal{P}_n$. It is these latter possibilities of non–deterministic choices, with their elegant algebraic laws, that makes `CSP` so well suited to model human and technology behaviours of an actual domain. `CSP` and `Petri Nets` provide a tool, and come with modelling techniques, with which to tackle description of domains of infrastructure components.

Type/Value Engineering: By *type/value engineering* we mean the engineering of typed information structures. In a view, not necessarily 'the' view, of *type/value engineering* — one that is significant in the usually "paper laden" bureaucracies of the man–made, and oftentimes public government operated or, at least, regulated infrastructure enterprises — information, in the form of documents of various kinds, "float" around the infrastructure enterprises, where this information may, or may not be structured, where it is subject to various kinds of operations: "Readings", edits (augmented updates), deposits in repositories (folders, files), copyings, creations, destructions, etc., and where many other kinds of operations may be performable.

More formally: There are types and values, and they relate:

```
type
  TYP, VAL
value
  typ: VAL → TYP
```

Values adhere to well–formedness, may be structured, and may be presented in many ways: copied, time–stamped, located in space, or otherwise made unique. Their types may reflect this, and operations creating values and operating upon values adhere to implied constraints:

```
value
  wf_TYP: TYP → Bool
  wf_VAL: VAL → Bool
  create_VAL: TYP × VAL ~→ VAL
  create_VAL(t,v) as v'
    pre wf_TYP(t) ∧ wf_VAL(v) ∧ ...
    post wf_VAL(t,v,v')
  op: VAL × ... × VAL ~→ VAL
  op(v1,...,vn) as v
    pre wf_VAL(v1) ∧ ... ∧ pre wf_VAL(vn) ∧ ...
    post wf_VAL(v1,...,vn,v)
```

Type/value engineering typically applies *denotational engineering* principles and techniques.

We shall take a look at *type/value engineering* phenomena in Section 4.

τ:49

Logics, Agents and Language–based Knowledge Engineering: There is the *knowledge engineering* view. In one, of several variants, of this view — and we shall only cover that variant, albeit ever so briefly — one focuses on *logics, agent behaviours* and *speech acts.*

The *logics* area has two facets to it: The classical logics which are part also of the denotational, concurrency, type system, and formal techniques facets described and assumed earlier, and the less classical logics of modal logics. Thus, by *logics* we here mean those of the *epistemic logics* of *knowledge & belief,* the *deontic logics* of *permission & obligation,* the
τ:50 *modal logics* of *possibility & necessity, &c.* Other logics are relevant — also when describing domains: *dynamic logics* of action, *defeasible, uncertainty* and *possibilistic logics, logics* of *belief revision, &c.* These are not just logics of AI and logic programming but also logics of general domain engineering. Thus, to express understanding of phenomena in domains and
τ:51 "derived" requirements entail, we find, extensive use of modal logics.

We present what may be termed the AI approach to *"agency"*, but intend to "lift" the AI *"agency"* notions to apply, inter alia, to domain engineering as well as to software (requirements and design). Agents *interact* through *communication.* Agents come in groups: *Multi agent "systems".* Agents perform both *competitive* and *co–operative* tasks. *Open multi agent "systems"* have agents serve different interests, *autonomously* and *heterogeneously.*
τ:52 Just like humans ! Agent *interaction* (alphabetically listed)[9] involves *arguments:* Formation of reasons, drawing of conclusions, and applying these actively; *commitments, conversations, co–ordination, dialogue, negotiation, obligation, planning, &c.* In doing so agents deploy
τ:53 various modal logics, and, as we shall next see: Speech acts.

Speech acts are characterised by: *Locutions* — The physical utterances of speakers; *illocutions* — The intended meaning of speaker utterances; and *perlocutions* — The actions that result from locutions. Wrt. illocutions, speech acts are often classified in the following five *performatives: Assertive,* ie. statements of fact; *directive,* ie. commands, requests or advice; *commissive,* eg. promises; *expressive,* eg. feelings and attitudes; and *declarative* which entail the occurrence of an action in themselves. Obviously speech acts and agents relate strongly. Speech act theories offer, I think, important concepts with which to model infrastructure domains.

τ:54

Computer Science: And then there is the *computer science* view: Computer science, as claimed earlier, is concerned with the mathematical properties of the things that can exist inside computers, and in particular with such things as: Computational models, complexity, and types. Although phenomena of the infrastructure component domains are rarely computable, one can still speak of functions, and of properties, although these might not be computable or decidable. Hence the *computer science* view shall flavour the ways in which we shall attempt to understand infrastructure component domains.

τ:55

Semiotics: Pragmatics, Semantics & Syntax: The *semiotics engineering* view ascribes primary importance to that which can basically not be formalised: The *pragmatics,* why we do what we do, why we use certain linguistic constructs, why we ascribe certain meanings and forms. The semiotics engineering view, on the basis of a clarified stand on pragmatics, then proceeds to first think and act *semantically:* Searching for, finding, narrating and formalising

[9] The listing is extracted from my MSc student, Hans Madsen Petersen's MSc pre–project report: *Agent Communication Languages and Speech Acts — and their Semantics,* October 2001.

"deepest" meanings, most abstract, elegant and beautiful. Finally the "semiotics engineer" selects suitable *syntactic* forms to designate the meanings (and "cover" the pragmatics). The *semiotics engineering* view is itself one of pragmatics, "meta–pragmatically", it is, for example, applied in *denotational* as well as in *type/value engineering*.

Discussion: There are other views. But the above suffice. Why we have *"gone to the trouble"* of enumerating and briefly explain the above computing & computer science views will now be revealed in the section which immediately follows.

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 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, with computerisation made possible by affordable information technology, the business process re–engineering of infrastructure components — as made possible by domain modelling methods — 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 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 ⊕ computing science ⊕ 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

3.1 Work Flows and Transactions

We exemplify four kinds of concrete work flow systems: railways, electronic business (Section 3.3), freight transport logistics (Section 3.4), and health–care. All exemplify the movement of information, materials and control. We now "flip" through several examples of domain descriptions. They are all reasonably substantial. But time does not permit us to

dwell on any one of them. I could have decided to show mere excerpts of formulas. Instead I "flip" quickly — while leaving paper versions of the foils for your perusal.

3.2 Railways

τ:63

3.2.1 On Railway Systems

Colloquially speaking, a railway system consists of the rail net: Lines between stations, and
τ:64 stations; lines and stations as consisting of rail units: Linear, switches, crossovers, etc.; rail
τ:65 units being in states, states implying the possibility of train movement along zero, one or
τ:66 more paths of a unit; and the rail net thus definining open and closed routes around the
τ:67 net, etc.; time tables and traffic: time tables prescribing train traffic: Movement of trains
τ:68 along routes and over time, etc.; the rolling stock: carriages (passenger or freight), locomo-
τ:69 tives; trains being composable from these, etc.; planning and operation: Planning and actually carrying out the insertion or removal of rail units, lines and stations; time table construction based on available and/or obtainable rolling stock, statistics and expectations of traffic; planning and handling the composition and decomposition of carriages into assembliews and trains; planning and executing the scheduled maintenance of carriages and assemblies; planning and carrying out station, line, service facilities and train crew rostering further operations: Passenger and freight inquieries, ticketing and reservation; etc. *&c.*

The railway domain thus is a multi–dimensional domain consisting of many, but highly interrelated, iner–woven "sub–systems".

τ:70 Section 7 on page 51 illustrates, by figures, some of the above.

3.2.2 A Hierarchical Narrative: Nets, Lines, Stations and Units

τ:71
ch9rail1

1. A railway net consists of one or more lines and two or more stations.
2. A railway net consists of units.
3. A line is a linear sequence of one or more linear units.
4. The units of a line must be units of the net of the line.
5. A station is a set of units.
6. The units of a station must be units of the net of the station.
7. No two distinct lines and/or stations of a net share units.
8. τ:72 A station consists of one or more tracks.
9. A track is a linear sequence of one or more linear units.
10. No two distinct tracks share units.
11. The units of a track must be units of the station (of that track).
12. A unit is either linear, or is a switch, or a is simple crossover, or is a switchable crossover, etc.

13. A unit has one or more connectors.A linear unit has two distinct connectors, a switch has three distinct connectors, crossovers have four distinct connectors, etc.
14. For every connector there are at most two units which have that connector in common.
15. Every line of a net is connected to exactly two, distinct stations of the net.
16. A linear sequence of units is a non-cyclic sequence of linear units such that neighbouring units share connectors.

3.2.3 A Formalisation: Nets, Lines, Stations and Units

τ:73
ch9/9rail2

type
N, L, S, Tr, U, 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 → **Tr-set**
12. is_Linear: U → **Bool**,
12. is_Switch: U → **Bool**
12. is_Simple_Crossover: U → **Bool**,
12. is_Switchable_Crossover: U → **Bool**
13. obs_Cs: U → **C-set**

τ:76

τ:74

value
16. lin_seq: **U-set** → **Bool**
    lin_seq(q) ≡
      let us = obs_Us(us) **in**
      ∀ i:U • u ∈ us ⇒ is_Linear(u) ∧
      ∃ q:U* • **len** q = **card** us ∧ **elems** q = us ∧
        ∀ i:**Nat** • {i,i+1} ⊆ **inds** q ⇒ ∃ c:C •
          obs_Cs(q(i)) ∩ obs_Cs(q(i+1)) = {c} ∧
        len q > 1 ⇒
          obs_Cs(q(i)) ∩ obs_Cs(q(**len** q)) = {}
      end

τ:77
ch9/9

τ:75

Some formal axioms are now given, not all !

axiom
1. ∀ n:N • **card** obs_Ls(n) ≥ 1,

1. ∀ n:N • **card** obs_Ss(n) ≥ 2,

3. ∀ n:N, l:L • l ∈ obs_Ls(n) ⇒ lin_seq(l)

7. $\forall$ n:N, l,l′:L •
   $\{l,l'\} \subseteq obs\_Ls(n) \wedge l \neq l'$
   $\Rightarrow obs\_Us(l) \cap obs\_Us(l') = \{\}$

7. $\forall$ n:N, l:L, s:S •
   $l \in obs\_Ls(n) \wedge s \in obs\_Ss(n)$
   $\Rightarrow obs\_Us(l) \cap obs\_Us(s) = \{\}$

7. $\forall$ n:N, s,s′:S •
   $\{s,s'\} \subseteq obs\_Ss(n) \wedge s \neq s'$
   $\Rightarrow obs\_Us(s) \cap obs\_Us(s') = \{\}$

8. $\forall$ s:S • **card** $obs\_Trs(s) \geq 1$

9. $\forall$ n:N, s:S, t:T •
   $s \in obs\_Ss(n) \wedge t \in obs\_Trs(s) \Rightarrow lin\_seq(t)$

τ:76

10. $\forall$ n:N, s:S, t,t′;T •
    $s \in obs\_Ss(n) \wedge \{t,t'\} \subseteq obs\_Trs(s) \wedge t \neq t'$
    $\Rightarrow obs\_Us(t) \cap obs\_Us(t') = \{\}$

14. $\forall$ n:N • $\forall$ c:C •
    $c \in \cup \{ obs\_Cs(u) \mid u:U \bullet u \in obs\_Us(n) \}$
    $\Rightarrow$ **card**$\{ u \mid u:U \bullet u \in obs\_Us(n) \wedge c \in obs\_Cs(u) \} \leq 2$

15. $\forall$ n:N,l:L • $l \in obs\_Ls(n) \Rightarrow$
    $\exists$ s,s′:S • $\{s,s'\} \subseteq obs\_Ss(n) \wedge s \neq s' \Rightarrow$
    let $sus = obs\_Us(s)$, $sus' = obs\_Us(s')$, $lus = obs\_Us(l)$ **in**
    $\exists$ u:U • $u \in sus$, u′:U • $u' \in sus'$, u″,u‴:U • $\{u'',u'''\} \subseteq lus$ •
    let $scs = obs\_Cs(u)$, $scs' = obs\_Cs(u')$,
    $lcs = obs\_Cs(u'')$, $lcs' = obs\_Cs(u''')$ **in**
    $\exists$! c,c′:C • $c \neq c' \wedge scs \cap lcs = \{c\} \wedge scs' \cap lcs' = \{c'\}$
    end end

3.2.4 A Compositional Narrative: Unit States, Routes and Train Movement

τ:77
ch9/9rail3

1. A path, $p : P$, is a pair of connectors, (c, c'), of some unit.[10]
2. A state, $\sigma : \Sigma$, of a unit is the set of all open paths of that unit (at the time observed).[11]
3. A unit may, over its operational life, attain any of a (possibly small) number of different states ω, Ω.
4. A route is a sequence of pairs of units and paths —

[10] 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.

[11] The state may be empty: the unit is closed.

5. 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.
6. An open route is a route such that all its paths are open. τ:78
7. A train is modelled as a route.
8. Train movement is modelled as a discrete function (map) from time to routes such that for any two adjacent times the two corresponding routes differ by at most one of the following: a unit path pair has been deleted from (one or another end) of the open routes, or (similarly) added, or both, or no changes — a total of seven possibilities (i–vii).

3.2.5 A Formalisation: Unit States, Routes and Train Movement

τ:79
ch9/9rail4

```
type
1 P = C × C
2 Σ = P-set
3 Ω = Σ-set
4 R′ = (U × P)*
5 R ={| r:R′ • wf_R(r) |}
7 Trn = R
8 Mov = T ~m~> Trn
value
2 obs_Σ: U → Σ
3 obs_Ω: U → Ω
```

τ:80

τ:82

```
5 wf_R: R′ → Bool
  wf_R(r) ≡
    ∀ i:Nat • i ∈ inds r let (u,(c,c′)) = r(i) in
      (c,c′) ∈ ⋃ obs_Ω(u) ∧ i+1 ∈ inds r ⇒
        let (_,(c″,_)) = r(i+1) in c′ = c″ end end

6 open_R: R → Bool
  open_R(r) ≡
    ∀ (u,p):U×P • (u,p) ∈ elems r ∧ p ∈ obs_Σ(u)

8 wf_Mov: Mov → Bool
  wf_Mov(m) ≡ card dom m ≥ 2 ∧
    ∀ t,t′:T • t,t′ ∈ dom m ∧ t < t′
    ∧ ~∃ t″:T • t″ ∈ dom m ∧ t < t″ < t′ ⇒
      let (r,r′) = (m(t),m(t′)) in clauses (i) – (vii) end
```

3.2.6 Discussion

τ:81

On the basis of domain models like the above we have worked out requirements, and, in cases, the design of software for:

τ:83

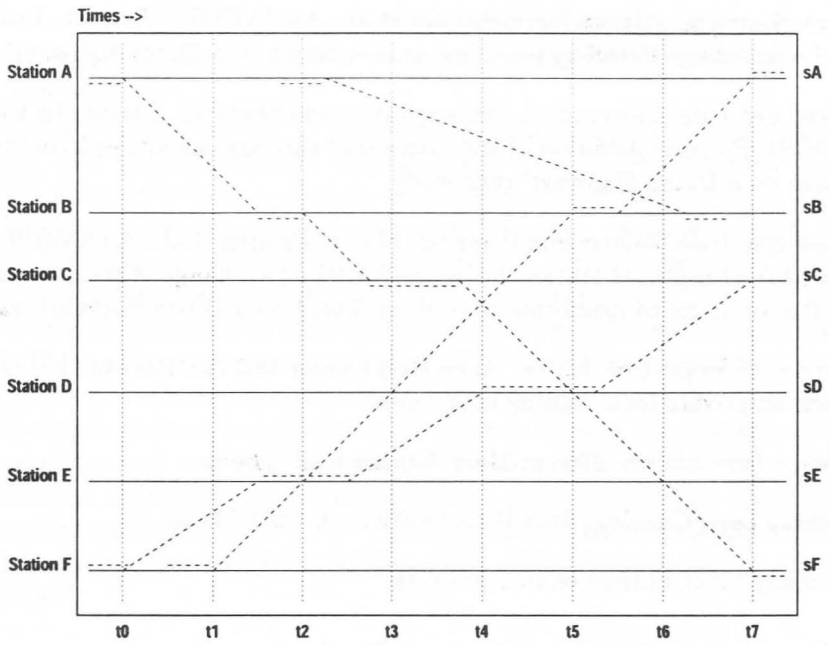

Figure 1: A Running Map

1. Running Maps: A UNU/IIST project for the Chinese Railways. Rescheduling trains when delayed.

2. Marshalling: The planning of freight train decomposition and composition. An ongoing project.

A Railway Marshalling Yard

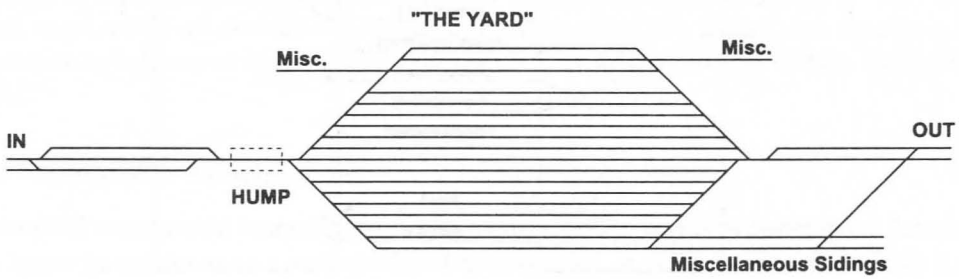

Figure 2: A Freight Train Marshalling Yard

Figure 2 shows a marshalling yeard. The "hump" is to be thought of as located in the highest point in the terrain. "The Yard" tracks are all to be thought of as sloping down, away from the hump, in direction left–to–right.

3. Crew Rostering: Albena Strupchanska et al.: An AMORE[12] Project. Periodic allocation and scheduling of staff to passenger trains, based on a Dutch Railways' case study.

4. Passenger Train Composition, Decomposition and Shunting: Panagiotis Karras et al.: An AMORE Project. Addition of and removal of carriage assemblies from "running" trains, based on a Dutch Railways' case study.

5. Passenger Train Maintenance Planning: Martin Penicka et al.: An AMORE Project. Planning day or night, at station, or between station, exchange of train assemblies in preparation for visits to maintenance stations, based on a Dutch Railways' case study.

6. Control of Single Line Traffic: Anne Haxthausen and Jan Peleska: FM99. A problem of German private local railway companies.

7. Station Interlocking: Kirsten Mark Hansen PhD Thesis.

8. Railway Level Crossing: Jens Ultik Skakkebæk PhD Thesis.

And many other railway related projects.

3.3 Electronic Administration and Business

We generalise a concept of electronic trading to apply across a full spectrum of Government institutions, $\mathcal{G}$, businesses (enterprises), $\mathcal{B}$, and citizens, $\mathcal{C}$. and thus to include as full a variety of $\mathcal{G}2\mathcal{G}$, $\mathcal{G}2\mathcal{B}$, $\mathcal{G}2\mathcal{C}$, $\mathcal{B}2\mathcal{G}$, $\mathcal{B}2\mathcal{B}$, $\mathcal{B}2\mathcal{C}$, $\mathcal{C}2\mathcal{G}$, $\mathcal{C}2\mathcal{B}$, $\mathcal{C}2\mathcal{C}$, transations.

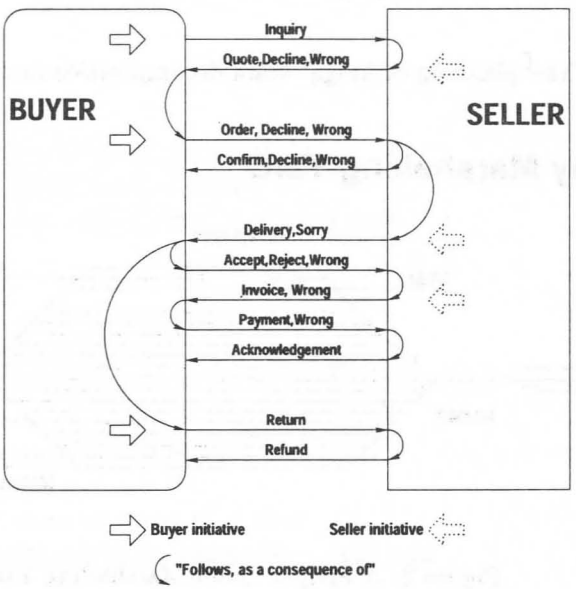

Figure 3: "Classical" E–Trading Transactions

[12] AMORE: Algorithmic Methods for Optimising Railways in Europe

In the case of the government, business and citizen infrastructure interactions we can postulate the following *domain*, ie. not necessarily computer & electronic communication supported, transactions:

$\mathcal{G}2\mathcal{G}$: Government institutions, $\mathcal{G}$, buy services from other $\mathcal{G}$, and $\mathcal{G}$ sell services to other $\mathcal{G}$. The $\mathcal{G}$ pay with monies (obtained through taxes, etc.), respectively offer free services, in return. $\mathcal{G}2\mathcal{B}$: $\mathcal{G}$ buy services, or request taxes, from businesses ($\mathcal{B}$), and pay, respectively offers free services, in return. $\mathcal{G}2\mathcal{C}$: $\mathcal{G}$ buy services (hire), or request taxes, from citizens
r:88 ($\mathcal{C}$), and pay, respectively offer free services, in return. $\mathcal{B}2\mathcal{G}$: Businesses ($\mathcal{B}$) buy services from $\mathcal{G}$, and pay $\mathcal{G}$ for these either through having already paid taxes or by paying special fees. $\mathcal{B}2\mathcal{B}$: $\mathcal{B}$ buy merchandise or services from other $\mathcal{B}$, and $\mathcal{B}$ offer merchandise or services to other $\mathcal{B}$. $\mathcal{B}$ usually pay for these outright. $\mathcal{B}2\mathcal{C}$: $\mathcal{B}$ buy services from citizens: ie. hire
r:89 temporary or permanent staff (employment), and $\mathcal{B}$ pay for these through salaries. $\mathcal{C}2\mathcal{G}$: Citizens ($\mathcal{C}$) obtain services from $\mathcal{G}$ (passport, drivers licence, etc., health–care, education, safety and security, etc.) and $\mathcal{C}$ pay for these either by paying special fees or through having already paid taxes. $\mathcal{C}2\mathcal{B}$: $\mathcal{C}$ buy merchandise from $\mathcal{B}$, and $\mathcal{C}$ pay for this. $\mathcal{C}2\mathcal{C}$: Two or more $\mathcal{C}$ together enter into political "grass–root" organisations, or leisure–time hobby club activities, or just plainly arrange meetings (incl. BBQ parties); and the two or more $\mathcal{C}$ "pay" for this by being "together".

3.3.1 Traders: Buyers and Sellers

r:90

Above we have stressed that also government (institutions) are part of the more general concept of `E-Business`, some aspects of contractual obligations, and a seeming "symmetry" between partners to any such contract (ie. buy, sell, etc.). As such we have stressed that "The Market" consists of buyers and sellers, whom we, as one, refer to as traders.

3.3.2 Traders: Agents and Brokers

An *agent*, to us, while we are still only discussing the *domain*, is a trader that acts (in a biased manner) on behalf of usually one other trader (either a buyer, or a seller), vis—a–vis a number of other traders (sellers, respectively buyers), in order to secure a "best deal". A *broker*, to us, while we are still only discussing the *domain*, is a trader that acts (in a neutral manner) on behalf one or more buyers and one or more sellers in order to help them negotiate a "deal."

3.3.3 Schematic Transactions

r:91

Sequences of contractual transactions can be understood in terms of "primitive" transactions:

A buyer `inquires` as to some merchandise or service. A seller may respond with a `quote`. A buyer may `order` some merchandise or service. A seller may `confirm` an order. A seller may `deliver` an order. A buyer may `accept` a delivery. A seller may send an `invoice`. A buyer may `pay` according to the invoice. A buyer may `return`, within warranty period, a
r:92 delivery. And a seller may `refund` such a return.

We have, deliberately, used the "hedge" 'may':

A trader may choose an action of `no response`, or a trader may inform that a transaction was `misdirected`, or a trader may `decline` to `quote`, `order`, `confirm`, `deliver`, `accept`, `pay` or `refund` !

3.3.4 Formalisation of Syntax

τ:93

```
type
  Trans == Inq | Ord | Acc | Pay | Rej
           Qou | Con | Del | Acc | Inv | Ref
           NoR | Dec | Mis
```

τ:96

The first two lines list the 'buyer', respectively the 'seller' initiated transaction types. The third line lists common transaction types.

U below stand for unique identifications, including time stamps (T), Sui for surrogate information, and MQP alludes to merchandise identification, quantity, price.

τ:94

```
  U, T, Su1, Su2, MQP
  Inq = MQP × U
  Qou = (Inq|Su1) × Inf × U
  Ord = Qou|Su2 × U
  Con = Ord × U
  Del = Ord × U
  Acc = Del × U
  Inv = Del × U
  Pay = Inv × U
  Rej = Del × U
  Ref = Pay × U
  NoR = Trans × U
  Dec = Trans × U
  Mis = Trans × U
value
  obs_T: U → T
```

τ:97

τ:98

In general we model, in the *domain*, a "subsequent" transaction by referring to a complete trace of unique, time stamped transactions. Thus, in general, a transaction "embodies" the transaction it is a manifest response to, and time of response.

τ:95

Figure 4 attempts to illustrate possible transaction transitions between buyers and sellers.

Figure 4: Buyer / Seller Protocol

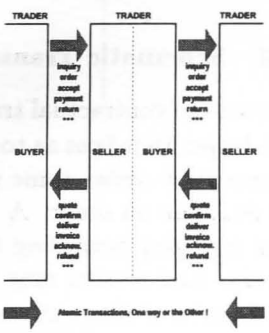

Figure 5: Trader=Buyer+Seller

3.3.5 "The Market"

Figure 5 attempts to show that a trader can be both a buyer and a seller. Thus traders "alternate" between buying and selling, that is: Between performing 'buy' and performing 'sell' transactions.

Figure 6 attempts to show "an arbitrary" constellation of buyer and seller traders. It highlights three supply chains. Each chain, in this example, consists, in this example, of a "consumer", a retailer, a wholesaler, and a producer.

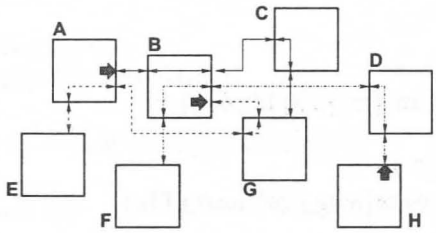

Figure 6: A Network of Traders and Supply Chains

3.3.6 Formalisation of Process Protocols

"The Market" consist of n traders, whether buyers, or sellers, or both; whether additionally agents or brokers. Each trader τ_i is able, potentially to communicate with any other trader $\{\tau_1, \ldots, \tau_{i-1}, \tau_{i+1}, \ldots, \tau_n\}$. We omit treatment of how traders come to know of one another. We focus only on the internal and external non–determinism which is always there, in the *domain*, when transactions are selected, sent and received.

Our model is in a variant of CSP, but expressed "within" RSL [5].

```
type
  Θ
  Idx = {| 1..n |}
value
  sys: (Idx ~m~> Θ) × n:Nat → Unit
  sys(mθ,n) ≡ || { tra(i)(mθ(i)) | i:Idx }

  tra: i:Idx → Θ →
    in {tc[j,i]|j:Idx•i≠j} out {tc[i,j]|j:Idx•i≠j} Unit
  tra(i)(θ) ≡ tra(i)(nxt(i)(θ))

  nxt: i:Idx → Θ →
    in {tc[j,i]|j:Idx•i≠j} out {tc[i,j]|j:Idx•i≠j} Θ
  nxt(i)(θ) ≡
    let choice = rcv ⌈⌉ snd in
    cases choice of
      rcv→receive(i)(θ), snd→send(i)(θ)
    end end
```

Annotations: The *system* is the parallel combination of *n traders*. *Traders* communicate over channels: *tc[i,j]* — from *trader i* to *trader j*. Each *trader* is modelled as a process which "goes on forever", but in steps of *next* state transitions. The *next* state transition non—deterministically (internal choice, ⌈⌉) "alternates" between expressing willingness to *receive*, respectively desire to *send*. In "real life", ie. in the *domain*, the choice as to which transactions are taken is *non–deterministic*. And it is an *internal choice*. That is: The *choice* is not influenced by the *environment*. τ:99 τ:100

Formalisation:

```
receive: i:Idx → Θ → in {tc[j,i]|j:Idx•i≠j} Θ
receive(i)(θ) ≡
  [] {let msg=tc[j,i]? in
        update_rcv_state(msg,j)(θ) end|j:Idx}

update_rcv_state: i:Idx × Θ → Θ
```

Annotations: [13] *update_rcv_state* is not a protocol function. *update_rcv_state* (not shown) describes the deposit of *msg* in a repository of received messages. If *msg* is a response to an earlier sent transaction, *msg_o*, then *update_rcv_state* describes the removal of *msg_o* from a repository of sent messages. *remove_sent_msg* (not shown) models the situation where no response (*nor*) is (ever) made to an earlier sent message. Once the internal non–deterministic choice (⌈⌉) has been made: Whether to *receive* or *send*, the *choice* as to whom to 'receive from' is also *non–deterministic*, but now *external* ([]). That is: *receive* expresses willingness to receive from any other *trader*. But just one. As long as no other *trader j* does not send anything to *trader i* that *trader i* just "sits" there, "waiting" — potentially forever. This is indeed a model of the real world, the *domain*. A subsequent *requirement* may therefore, naturally, be to provide some form of *time out*. A re–specification of *receive* with time out is a *correct implementation* of the above. τ:101 τ:102

```
send: i:Idx → Θ → in {tc[i,j]|j:Idx•i≠j} Θ
send(i)(θ) ≡
  let choice = ini ⌈⌉ res ⌈⌉ nor in
  cases choice of
    ini → send_initial(i)(θ),
    res → send_response(i)(θ),
    nor → remove_received_msg(θ) end end
```

Either a *trader*, when communicating a transaction, chooses an *initial (ini)* one, or chooses one which is in *response (res)* to a message received earlier, or chooses to *not respond (nor)* to such an earlier message The choice is again non–deterministic internal. In the last case the state is updated by *non–deterministically internal choice* (not shown) *removing* the, or an earlier *received message*. τ:103

[13] Please note that we are not always RSL "conformant": The receive signature deploys a non–standard 'dependent' type usage: The leftmost *i* binds the following *is*.

Note that the above functions describe the internal as well as the external non–determinism of protocols. We omit the detailed description of those functions which can be claimed to not be proper protocol description functions — but are functions which describe more the
r:104 particular *domain* at hand: Here "The Market".

```
send_initial: i:Idx → Θ → out {tc[i,j]|j:Idx•i≠j} Θ
send_initial(i)(θ) ≡
  let choice = buy ⌈⌉ sell in
  cases choice of
    buy → send_init_buy(i)(θ),
    sell → send_init_sell(i)(θ) end end

send_response: i:Idx → Θ → out {tc[i,j]|j:Idx•i≠j} Θ
send_response(i)(θ) ≡
  let choice = buy ⌈⌉ sell in
  cases choice of
    buy → send_res_buy(i)(θ),
    sell → send_res_sell(i)(θ) end end
```

In the above functions we have, perhaps arbitrarily chosen, to distinguish between *buy* and *sell* transactions. Both *send_initial* and *send_response* functions — as well as the four auxiliary
r:105 functions they invoke — describe aspects of the protocol.

```
send_init_buy: i:Idx → Θ → out {tc[i,j]|j:Idx•i≠j} Θ
send_init_buy(i)(θ) ≡
  let choice = inq ⌈⌉ ord ⌈⌉ pay ⌈⌉ ret ⌈⌉ ... in
  let (j,msg,θ') = prepare_init_buy(choice)(i)(θ) in
  tc[i,j]!msg ; θ' end end

send_init_sell: i:Idx → Θ → out {tc[i,j]|j:Idx•i≠j} Θ
send_init_sell(i)(θ) ≡
  let choice = quo ⌈⌉ con ⌈⌉ del ⌈⌉ inv ⌈⌉ ... in
  let (j,msg,θ') = prepare_init_sell(choice)(i)(θ) in
  tc[i,j]!msg ; θ' end end
```

prepare_init_buy is not a protocol function, nor is *prepare_init_sell*. They both assemble an initial buy, respectively sell message, *msg*, a target trader, *j*, and update a send repository
r:106 state component.

```
send_res_buy: i:Idx → Θ → out {tc[i,j]|j:Idx•i≠j} Θ
send_res_buy(i)(θ) ≡
  let (θ',msg)=sel_update_buy_state(θ),
      j=obs_trader(msg) in
  let (θ'',msg') = response_buy_msg(msg)(θ') in
  tc[i,j]!msg'; θ'' end end

send_res_sell: i:Idx → Θ → out {tc[i,j]|j:Idx•i≠j} Θ
```

```
send_res_sell(i)(θ) ≡
  let (θ',msg)=sel_update_sell_state(θ),
      j=obs_trader(msg) in
  let (θ'',msg') = response_sell_msg(msg)(θ') in
  tc[i,j]!msg'; θ'' end end
```

sel_update_buy_state is not a protocol function, neither is *sel_update_sell_state*. They both describe the selection of a previously deposited, buy, respectively a sell message, *msg*, (from it) the index, *j*, of the trader originating that message, and describes the update of a received messages repository state component. *response_sell_msg* and *response_buy_msg* both effect the assembly, from *msg*, of suitable response messages, *msg'*. As such they are partly protocol functions. Thus, if *msg* was an *inquiry* then *msg'* may be either a *quote*, *decline*, or a *misdirected* transaction message. Etcetera.

3.3.7 Requirements

Projection Synopsis: We focus only on the communication between traders. We basically ignore the "content" of any transaction, and shall instead focus on automating certain sequences of transactions.

Instantiation Synopsis: Whereas the domain model of traders was a model, essentially, intrinsically, of human operations, we now try to automate as much as possible the response to received transactions. Thus, as an example: (1) If a consumer order can be delivered by the retailer, without human (retailer staff) intervention, it will be done so. (2) If a consumer order cannot be delivered by the retailer, but that retailer can re-order from a wholesaler, who can deliver — both atomic transactions without human (retailer and wholesaler staff) intervention, it will be done so. (3) And if a consumer order cannot be delivered by the retailer, but that retailer can re-order from a wholesaler, who then cannot deliver, but must re–order from producer, who can deliver — all atomic transactions without human (retailer, wholesaler and producer staff) intervention, it will be done so. Figure 7 attempts to show the three cases listed above. There might be delays, waiting times, between order receipt and delivery and/or re–ordering, across the supply–chain.

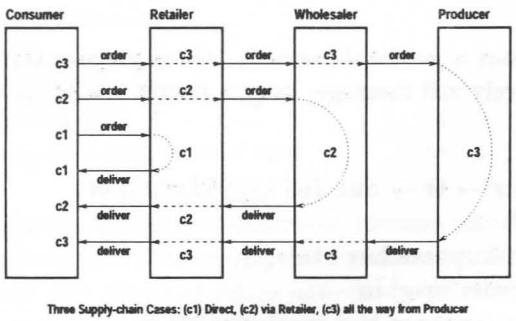

Figure 7: E–Business "Supply" Chain

τ:111

Extension Synopsis: We introduce electronic traders and brokers. They permit arbitrarily wide inquiries: Potentially to all providers (retailers, wholesalers, or producers) of specified merchandise (or services), offers ("confirmations") of merchandise (or services) to all "takers" (consumers, retailers, or wholesalers), first–come–first serve ("auction"-like) orders, *&c.* These roughly sketched domain requirements are considered extensions as they might not be humanly feasible in actual domains.

τ:112

Initialisation Synopsis: Due to our projection we need only consider how traders, agents and brokers initially, and in an ongoing way, come to know of one another. We omit details — *"left as an exercise"*.

3.3.8 Discussion

τ:113

There is an urgent need to *bring semantics to bear on electronic transactions,* to study the *spectrum* of $\mathcal{G}$, $\mathcal{B}$, and $\mathcal{C}$ *inter–* and *intra–transactions,* and to apply *modal logics* and *speech act theories* to *automated families* of *autonomous agents* and *brokers.* We believe that we first need establishing models like the above in order go on with these urgent studies.

3.4 Logistics

τ:114

lisboa/wfslogi

The ability to transport freight from well–nigh any spot on the globe to well–nigh any other spot on the same globe is a test of *"infra–structure–hood"*.

The term 'logistics' is here taken to cover an ability of managing the interfaces between: clients who send and receive freight items with logistics firms who organise the transport and deal with transport companies who offer transportation by means of conveyors with whom they also interface; and with hubs where various kinds of conveyor types congregate. We first narrate some facets of logistics, then suggest a formal model.

3.4.1 Informal View

τ:115

Freight Transport: Freight transport, as abstractly designated by Figure 8, may evolve by means of various conveyors: from truck (A→B) to train (B→C) to ship (C→F) to train (F→G) and finally by truck (G→H). Intermediate transfer places are called hubs: Truck terminals, railway stations, harbours and airports.

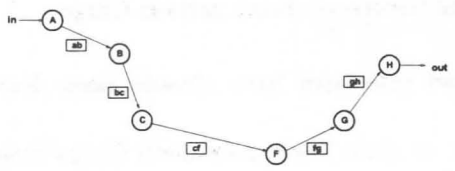

Figure 8: A Freight Transport

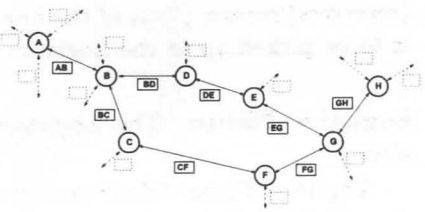

Figure 9: Part of a Logistics Net

A bill–of–lading (BoL, see Figure 10) describes the composition of individual transports (ab, bc, cf, fg, and gh).

τ:116

Logistics Nets: Many routes composed from different kinds of conveyors between hubs are normally possible. Figure 9 indicates alternative routing between hubs (here B and G).

τ:117

A Freight Transport Trace: To manage freight transports logistics firms need be able to trace the whereabouts of freights. Figure 10 indicates (by bullets •) "interesting" points: When freight enters first hub (once), when it leaves a hub on a conveyor (five times), when it is on a conveyor (five times), when it enters a hub from a conveyor (five times), and when it finally leaves a hub for a receiver (once).

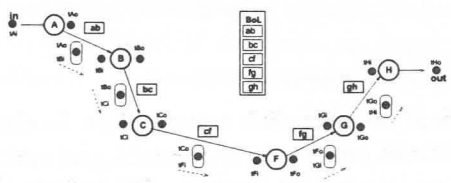

Figure 10: The Trace of a Specific Transport

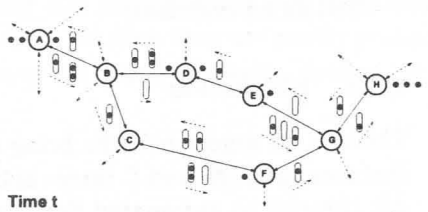

Figure 11: A State of a Logistics Net

τ:118

The Dynamic State of A Logistics Net: At any one time many items of freight are being transported on a diversity of conveyors between many (pairs of) hubs. Figure 11 indicates, for an arbitrary point in time, t, such a state of freights (the bullets •) and conveyors (the vertically oriented rounded "boxes" "containing" zero, one or two •s.
We now model a logistics system consisting of clients (senders and receivers of freights), logistics firms (which arrange transport for clients), transport companies which own conveyors, and hub (enterprises). We model each of these (clients, firms, companies, conveyors and hubs) as processes.

τ:119

Clients: Clients (potential senders) inquire with Logistics Firms for virtual BoLs — these are like quotes — and receives possibly several such in return. Clients (senders) deliver (typically with a virtual BoL as reference) to Logistics Firms actual Cargo and receives Copies of a relevant, unique BoL. Clients (senders and/or receivers) inquire as to the whereabouts (a trace) of some cargo based on a/the Copy of its BoL and receives trace information. Clients (receivers) receive ("out of the blue", but most likely expectedly so) information that a Cargo is to be picked up at the Logistics Firm[14]. Clients (receivers) Fetch arrived Cargo.

τ:120

Logistics Firms: The Logistics Firm to Client interfaces have already been described above.

Logistics Firms deliver one or more (possibly, or most likely unrelated) Cargo including their BoLs to Hubs. Logistics Firms receive notification from Hubs that initial Hub and/or intermediate Hub Cargo has departed on a Conveyor (onto which it has been loaded). Logistics Firms receive notification from Hubs that intermediate Hub and/or final Hub Cargo has

[14] You may wish to model also or instead the option or alternative that the Logistics Firm delivers the cargo to Client (receiver)

arrived — and if final, that it can be fetched. Logistics Firms Fetch finally arrived Cargo. Logistics Firms request Time and Fee Table Information from Transport Companies and receives such information. Logistics Firms inquire as to Availability for (Cargo, departure, etc., specified) Cargo space and receive temporary confirmation of such (or later, close in time, etc.) space and conditions, or unavailability of space. Logistics Firms bindingly reserve Cargo space on designated departures, etc.

τ:121

Transport Companies: The Transport Company to Logistics Firm interfaces have already been described above.

The Transport Companies receive information from their Conveyors that (such and such) Cargo has been unloaded at a Hub. The Transport Companies receive information from their Conveyors that (such and such) Cargo has been loaded at a Hub. And hence that the conveyors has been at a hub.

τ:122

Hubs: The Hub to Logistics Firm interfaces have already been described above.

The Hubs receive, hence unloaded, Cargo, from Conveyors. The Hubs delivers, hence load, Cargo, to Conveyors.

Conveyors: The Conveyor to Hub interfaces and the Conveyor to Transport Company interfaces have already been described above.

Hence there is no more to describe concerning external interfaces.

3.4.2 System Formalisation

τ:123

There seems to be just the following *"players"* (See Figure 12): Clients, logistics firms, transport companies, hubs, and conveyors, modelled as processes (ie. as behaviours), and *"interfaces"* Clients to/from logistics firms, logistics firms to/from transport companies and to/from hubs, transport companies to/from conveyors and to/from hubs, and conveyors to/from hubs.

τ:124 modelled in terms of channel communications.

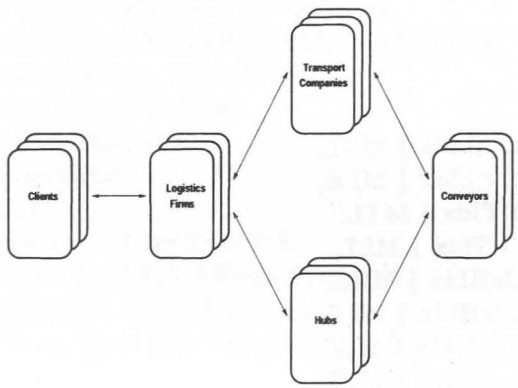

Figure 12: Process and Channel Graph of a Logistics System

τ:125

States:

type
KIdx, LIdx, TIdx, HIdx, CIdx
KΣ, LΣ, TΣ, HΣ, CΣ
KΣM = KIdx $\overrightarrow{m}$ KΣ, LΣM = LIdx $\overrightarrow{m}$ LΣ
TΣM = TIdx $\overrightarrow{m}$ TΣ, HΣM = HIdx $\overrightarrow{m}$ HΣ
CΣM = CIdx $\overrightarrow{m}$ CΣ
value
kΣm:KΣM, ℓΣm:LΣM,
tΣm:TΣM, hΣm:HΣM,
cΣm:CΣM

System Process:

sys: **Unit** → **Unit**
sys() ≡
	{ cli(i)(kσm(i))	i:KIdx }	
	{ log(i)(ℓσm(i))	i:LIdx }	
	{ tra(i)(tσm(i))	i:TIdx }	
	{ hub(i)(hσm(i))	i:HIdx }	
	{ con(i)(cσm(i))	i:CIdx }

Transaction Message Types: We do not detail the concrete types of messages. τ:126

type
MKL = ..., MLK = ...,
MLT = ..., MTL = ...,
MLH = ..., MHL = ...,
MHC = ..., MCH = ...,
MTC = ..., MCT = ...

Channels:

channel
{ ckℓ[k,ℓ] | k:KIdx, ℓ:LIdx } MKL,
{ cℓk[ℓ,k] | k:KIdx, ℓ:LIdx } MLK,
{ ctℓ[t,ℓ] | ℓ:LIdx,t:TIdx } MTL,
{ cℓt[ℓ,t] | ℓ:LIdx, t:TIdx } MLT,
{ chℓ[h,ℓ] | ℓ:LIdx,h:HIdx } MHL,
{ cℓh[ℓ,h] | ℓ:LIdx, h:HIdx } MLH,
{ chc[h,c] | h:HIdx, c:CIdx } MHC,
{ cch[c,h] | h:HIdx, c:CIdx } MCH,
{ ctc[t,c] | t:TIdx, c:CIdx } MTC,
{ cct[c,t] | t:TIdx, c:CIdx } MCT

3.4.3 Client Formalisation

τ:127

Client Types:

```
type
  BoL
  MKL == Inq_BoL | VBoL
         | Delivery | CBoL
         | Inq_Trace | Trace
         | Received | Fetch | Cargo
  Inq_BoL:: VCargo × HIdx × HIdx × Qual-set
  Qual == fastest | shortest | cheapest | reliably | safest | ...
  VBoL
  Delivery :: RCargo × HIDx × HIdx × Qual-set × VBoL
  CBoL
  Inq_Trace :: CBoL
  Trace
  Received :: LIdx × CBoL
  Fetch :: CBoL
  RCargo
  VCargo
value
  obs_VBoL: BoL → VBoL
  obs_CBoL: BoL → CBoL
  obs_VCargo: RCargo → VCargo
```

τ:128

Client Auxiliary Functions:

```
type
  Trace
  Inq_Pred_Info :: LIdx×VCargo×HIdx×HIdx×Qual-set
  Dlvr_Pred_Info :: LIdx×RCargo×HIdx×HIdx×Qual-set×VBoL
  Trace_Pred_Info, Fetch_Pred_Info :: LIdx×CBoL
  Inq_Upda_Info :: Inq_Pred_Info×VBoL-set
  Dlvr_Upda_Info :: Dlvr_Pred_Info×CBoL
  Trace_Upda_Info :: Trace×CBoL
  Received :: LIdx × CBoL
  Cli_Pred_Info = Inq_Pred_Info | Dlvr_Pred_Info
                | Trace_Pred_Info | Fetch_Pred_Info
  Cli_Upda_Info = Inq_Upda_Info | Dlvr_Upda_Info
                | Trace_Upda_Info | Received
value
  cli_pred: Cli_Pred_Info → KΣ → Bool
  cli_upda: Cli_Upda_Info → KΣ → KΣ
```

τ:129

Client Processes:

```
value
  cli: i:KIdx → KΣ →
       in,out { ckℓ[i,ℓ] | ℓ:LIdx }
       in { cℓk[ℓ,i] | ℓ:LIdx } Unit
  cli(i)(kσ) ≡
    let kσ′ = (inq(i)(kσ)∏dlvr(i)(kσ)∏trace(i)(kσ)∏fetch(i)(kσ))
                ∏ rcvd(i)(kσ) in
    cli(i)(kσ′) end

  inq: i:KIdx → KΣ → in,out { ckℓ[i,ℓ] | ℓ:LIdx } KΣ
  inq(i)(kσ) ≡
    let ipi=mk_Inq_Pred_Info(ℓ,vc,f,t,qs):Inq_Prd_Info
          • cli_pred(ipi)(kσ) in
    ckℓ[i,ℓ] ! mk_Inq_BoL(vc,f,t,qs) ;
    let vbols = ckℓ[i,ℓ] ? in cli_upd(mk_Inq_Upd(ipi,vbols))(kσ)
    end end

  dlvr: i:KIdx → KΣ → in,out { ckℓ[i,ℓ] | ℓ:LIdx } KΣ
  dlvr(i)(kσ) ≡
    let dpi=(ℓ,rc,f,t,qs,vbol):Dlvr_Pred_Info
          • cli_pred(dpi)(kσ) in
    ckℓ[i,ℓ] ! mk_Delivery(rc,f,t,qs,vbol) ;
    let cbol = ckℓ[i,ℓ] ? in cli_upd(mk_Dlvr_Upd(dpi,cbol))(kσ)
    end end
```

τ:130

```
  trace: i:KIdx → KΣ → in,out { ckℓ[i,ℓ] | ℓ:LIdx } KΣ
  trace(i)(kσ) ≡
    if ∃ cbol:CBoL • cli_pred(mk_Trace_Pred_Info(ℓ,cbol))(kσ)
    then
      let tpi=mk_Trace_Pred_Info(ℓ,cbol):Trace_Info
            • k_trace_pred(ℓ,cbol)(kσ) in
      ckℓ[i,ℓ] ! mk_Inq_Trace(cbol) ;
      let mk_Trace(trace) = ckℓ[i,ℓ] ? in
      cli_upd(mk_Trace_Upd(trace,cbol))(kσ)
      end end
    else kσ end

  rcvd: i:KIdx → KΣ → in { cℓk[ℓ,i] | ℓ:LIdx } KΣ
  rcvd(i)(kσ) ≡
    let rcvd = []{ cℓk[ℓ,i] ? | ℓ:LIdx } in
    cli_upd(rcvd)(kσ)
    end

  fetch: i:KIdx → KΣ → in,out { ckℓ[i,ℓ] | ℓ:LIdx } KΣ
```

```
fetch(i)(kσ) ≡
  let fpi=mk_Fetch_Pred_Info(ℓ,cbol):Fetch_Pred_Info
        • cli_pred(fpi)(kσ) in
  ckℓ[ i,ℓ] ! mk_Fetch(cbol) ;
  let rcargo = ckℓ[ i,ℓ] ? in
  cli_upd(mk_Fetch_Upd(fpi,rcargo))(kσ)
  end end
```

3.4.4 Logistic Firm Formalisation

r:131

Function Types:

```
value
  ℓ_inq_vbols:
    VCargo×HIdx×HIdx×Qual-set → LΣ → VBoL-set
  ℓ_inq_upd:
    KIdx×VCargo×HIdx×HIdx×Qual-set → LΣ → LΣ
  ℓ_dlvr_cbol:
    (RCargo×HIdx×HIdx×Qual-set)×VBoL → LΣ → CBoL
  ℓ_dlvr_upd:
    (RCargo×HIdx×HIdx×Qual-set)×(CBoL×VBoL) → LΣ → LΣ
  ℓ_trace:
    CBoL → LΣ → LΣ
  ℓ_trace_upd:
    CBoL×Trace → LΣ → LΣ
  ℓ_fetch:
    CBoL → LΣ → Cargo
  ℓ_fetch_upd:
    CBoL×Cargo → LΣ → LΣ
  ℓ_rcvd:
    LΣ → Received
  ℓ_rcvd_upd:
    KIdx×CBoL → LΣ → LΣ
```

r:132

Logistics Firm Processes:

```
value
  log: ℓ:LIdx → LΣ →
       in,out { ckℓ[i,ℓ] | i:KIdx }
       out { cℓk[ℓ,i] | i:KIdx }
       in,out { cℓh[i,ℓ] | i:HIdx }
       in,out { chℓ[ℓ,i] | i:HIdx }
       in,out { cℓt[i,ℓ] | i:TIdx }
       in,out { ctℓ[ℓ,i] | i:TIdx } Unit

  log(ℓ)(ℓσ) ≡
```

```
    let lσ' =
        cli_log(l)(lσ) ∏ log_cli(l)(lσ)
        ∏ log_hub(l)(lσ) ∏ log_tra(l)(lσ) in
    log(l)(lσ') end
```

Client Initiated → Logistics Firm Transactions: τ:133

```
value
  cli_log: l:LIdx → LΣ →
      in,out { ckl[i,l] | i:KIdx } out { clk[l,i] | i:KIdx } Unit
  cli_log(l)(lσ) ≡
    let lσ' =
      [] { let msg = kl[i,l] ? in
           cases msg of
             mk_Inq_BoL(vc,f,t,qs) →
               let vbols = l_inq_vbols(vc,f,t,qs)(lσ) in
               kl[i,l]!vbols;
               l_inq_upd((i,vc,f,t,qs),vbols)(lσ) end,
             mk_Delivery((rc,f,t,qs),vbol) →
               let cbol = l_dlvr_cbol((rc,f,t,qs),vbol)(lσ) in
               kl[i,l]!cbol;
               l_dlvr_upd((rc,f,t,qs),(cbol,vbol))(lσ) end,
             mk_Inq_Trace(cbol) →
               let trace = l_trace(cbol)(lσ) in kl[i,l]!trace;
               l_trace_upd(cbol,trace)(lσ) end,
             mk_Fetch(cbol) →
               let cargo = l_fetch(cbol)(lσ) in kl[i,l]!cargo;
               l_fetch_upd(cbol,cargo)(lσ) end
           end end | i:KIdx } in
    log(l)(lσ') end
```

Logistics Firm Initiated → Client Transactions: τ:134

```
value
  log_cli: l:LIdx → LΣ → out { clk[l,i] | i:KIdx } Unit
  log_cli(l)(lσ) ≡
    if l_rcvd_pred(lσ)
      then
        let (i,cbol) = l_rcvd(lσ) in
        clk[l,i] ! mk_Received(l,cbol) ;
        log(l)(l_rcvd_upd(i,cbol)(lσ)) end
      else log(l)(lσ)
    end
```

Etcetera. We leave it — as an exerise — to the reader to decipher the formulas.

3.4.5 Discussion

τ:135

The two previous examples: electronic–business, and freight transport logistics, share many properties. Besides similar forms of transactions, apparent need for modalities and speech act concepts, distribution and concurrency, *&c.*, they both model the transaction–oriented flow of material, information, and control. The next example emphasises the additional flow of people.

3.5 Health–care

τ:136

lisboa/hecs

The health–care sector "features" many stake–holders:

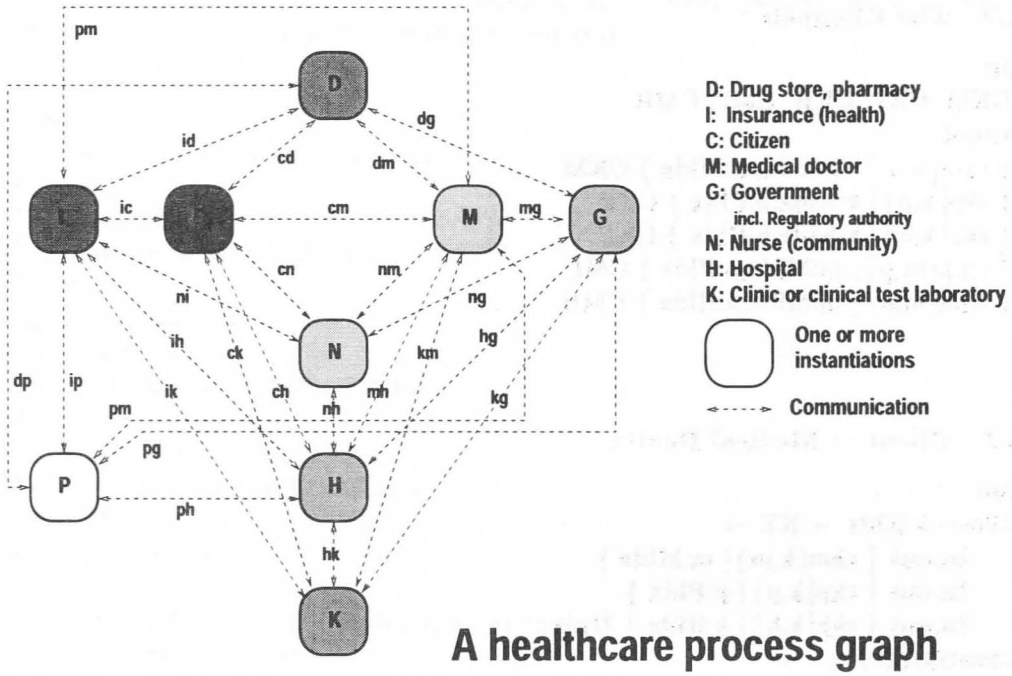

Figure 13: Stake–holders of the Healthcare Sector

τ:137

Figure 13 designates drug stores health insurance — citizens medical doctors — national board of health ministry of health — community nurses pharmaceutical industry hospitals — clinics and their pairwise interactions.

Without much further ado, we embark on a formalisation.

3.5.1 The System States and Process

τ:138

type

$K\Sigma$, $M\Sigma$, $P\Sigma$, $H\Sigma$, ...

KIdx, MIdx, PIdx, HIdx, ...

$\Theta = (\text{KIdx} \xrightarrow[m]{} K\Sigma) \times (\text{MIdx} \xrightarrow[m]{} M\Sigma)$

$\quad \times (\text{PIdx} \xrightarrow[m]{} P\Sigma) \times (\text{HIdx} \xrightarrow[m]{} H\Sigma)$

```
            × ...
value
  (ikσ,imσ,ipσ,ihσ,...):Θ
  sys: Θ → Unit
  sys(ikσ,imσ,ipσ,ihσ,...) ≡
    || { client(i)(ikσ(i)) | i:KIdx } ||
    || { doctor(i)(imσ(i)) | i:MIdx } ||
    || { pharmacy(i)(ipσ(i)) | i:PIdx } ||
    || { hospital(i)(ihσ(i)) | i:HIdx } || ...
```

3.5.2 The Channels

τ:139

```
type
  CKM, CKP, CKH, CMP, CMH, ...
channel
  { ckm[k,m] | k:KIdx,m:MIdx } CKM
  { ckp[k,p] | k:KIdx,p:PIdx } CKP
  { ckh[k,h] | k:KIdx,h:HIdx } CKH
  { cmp[m,p] | m:MIdx,p:PIdx } CMP
  { cmh[m,h] | m:MIdx,h:HIdx } CMH
  ...
```

3.5.3 Client ↔ Medical Doctor

```
value
  client: k:KIdx → KΣ →
      in,out { ckm[k,m] | m:MIdx }
      in,out { ckp[k,p] | p:PIdx }
      in,out { ckh[k,h] | h:HIdx } Unit
  client(k)(kσ) ≡
```

τ:140

```
type
  KChoice == meddoc | pharma | ...
value
  client(c)(kσ) ≡
[1] let choice = meddoc ⌈⌉ pharma ⌈⌉ ... in
[2] let kσ′ =
[3]   cases choice of
[4]     meddoc →
[5]       let d = select_meddoc(kσ) in
[6]       ckm[c,d]!kσ ;
[7]       ckm[c,d]? end,
[8]     pharma →
[9]       let p = select_pharma(kσ) in
```

```
[10]     ckm[c,p]!kσ ;
[11]     ckp[c,p]? end,
           ...
         end in
[12] client(c)(kσ′) end end
```

τ:141

(1) A client chooses either to go to the doctor, or to the pharmacy, or ... (4) If the client decides to go to the doctor, (5) a selection of which doctor has to be made, (6) and the client goes to that doctor, (7) and returns from that doctor (2) in a new state. (8) If the client decides to go to the pharmacy, (9) a selection of which pharmacy has to be made, (10) and the client goes to that pharmacy, (11) and returns from that pharmacy, (2) in a new state.
τ:142 Etcetera. (12) The client continues in that new state.

```
type
  MChoice == citizn | pharma | ...
value
  doctor(d)(dσ) ≡
[1] let choice = citizn | pharma | ... in
[2] let dσ′ =
[3]  cases choice of
[4]   citizn →
[5]    ‖ { let kσ = ckm[c,d]? in
[6]        let (kσ′,dσ′)=handle_c(kσ,dσ) in
[7]        ckm[c,d]!kσ′ ;
[8]        dσ′ end end | c:CIdx }
[9]    ⫿ dσ,
[10]  pharma →
[11]   ‖ { let pσ = ckm[d,p]? in
[12]       let (pσ′,dσ′)=handle_p(pσ,dσ) in
[13]       ckm[d,p]!pσ′ ;
[14]       dσ′ end end | p:PIdx }
[15]   ⫿ dσ,
         ...
       end in
[16] doctor(d)(dσ′) end end
```

τ:143

(1) The medical doctor choose whether to respond to a visit from a client, or a call from the pharmacy, or ... (3,4) In case of declaring willingness to receive a client, (5–8) some client is handled, (9) or no client is handled — say for the case of no clients in the waiting room ! (5) A client is received, (6) treated, (7) sent away, (8) and the medical doctor prepares for
τ:144 next action (2) in a new state. (3,10) In case of declaring willingness to receivecall from a pharmacy, (11–14) some pharmacy call is handled, (15) or no pharmacy call is handled — say for the case of no such calls ! (11) A pharmacy call is accepted, (12) handled, (13) terminated, (14) and the medical doctor prepares for next action (2) in a new state. (16) The medical doctor continues in that new state.

τ:145

3.5.4 Remaining Interactions

Similar analysis and description must be carried out for the full spectrum of the health–care sector stake–holder intra– and inter–actions:

- *Client* ↔ *Pharmacy*
- *Client* ↔ *Hospital*
- *Medical Doctor* ↔ *Pharmacy*
- *Medical Doctor* ↔ *Hospital*
- *&c.*

We have just intimated what is to be done !

3.5.5 Discussion

τ:146

The model is cursory. It does, however, model indeterminicay: Impatient clients returning from a medical doctor's waiting room withiut being treated. Irrational medical doctors never choosing to listen to pharmacy calls. Etcetera.

3.6 Transaction Scripts

τ:147
lisboa/wfstran

3.6.1 The Problem

We have seen in the three previous examples how tasks were carried out by a distributed set of operations either on freight (as for the logistics example), or on potential or real merchandise (as for the electronic business example), or on people — in the form of patients. 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. τ:148

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".[15] 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, this *discussion & analysis*, respectively search for a *common abstraction, a general concept.* τ:149

3.6.2 Clients, Work Stations, Scripts and Directives

τ:150

There are *clients* and there are *work stations*. Clients initialise and *interpret scripts*. A script is a set of *time–interval* stamped collection of *directives*. 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

[15] 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.

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
r:151 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. A directive may either be one which prescribes one, or another, of a small set of *operations* to take place — with the possible effect that, at operation completion, one or more directives
r:152 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.

3.6.3 A Simple Model of Scripts

r:153

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) ~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')
```

r:154

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.

3.6.4 A Simple Model of Work Flow

τ:155

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 }
```

τ:156

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.[16] $c\omega$, $w\omega$ 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.

τ:157

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) ≡
```

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

```
    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
```

τ:158

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.

τ:159

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 }
```

τ:160

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.

τ:161

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 • s ∈ s(t',t'') in
      cases 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
```

τ:162

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.

3.6.5 Discussion

τ:163

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: τ:164
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 τ:165
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: $t'\leq t''<t$.
τ:166 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 *clients* and *work stations* as seen from the *client* side. That "same" interface as seen from the side of *work stations* 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 !".
τ:167 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
τ:168 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.
τ:169 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.

3.7 Discussion — So Far !

τ:170
lisboa/wfsdis **Interpretation as Health–care System:** The *Transaction Script Work Flow* example can be interpreted, amongst many alternatives, as an abstract health–care system. Clients are potential patients. Scripts are their plans for visits to family physician, pharmacy, clinical test laboratory and hospital. And these latter are seen as work stations. Of course the work stations themselves have scripts. Etcetera !

General Comments: The electronic business requirements, as illustrated by Figure 7, can
τ:171 be handled by agent clients and appropriate directives: The buyer initialises the agent to 'go to retailer' and simple 'retail order' directives. The latter is either fulfilled by the retailer (hence the agent is released and returns to buyer with goods), or replaced by 'go to wholesale seller', 'wholesale order' and 'go to retailer' directives, with the 'wholesale order directive' either being fulfilled by the wholesale seller (hence releases the agent to return to retailer), or, etcetera !

Similar scripts can be associated with logistics 'bill–of–ladings', etc.

In connection with appropriate simple speech act primitives the script and transaction notion can be made rather sophisticated !

4 Type/Value "Systems"

τ:172
lisboa/infointr

4.1 Intuition

To motivate our treatment of types and values we first exemplify documents of infrastructure components. Then we look at some generic issues of such documents.

4.1.1 The Problem

Infrastructure component systems have certain characteristic features. One of them is the kind of information and/or documents that "flavour" the component. Below we hint at some examples.

4.1.2 Patient Medical Records

τ:173

A patient medical record is a typical (a core) document of health–care systems. The base part of a patient medical record contains administrative information. Descriptive parts contains doctor's notes (annamnese, measurements, diagnostics, etc.). Goal parts set out plans for treatment.

type
  Base, Pn, Desc, Goal
  PMR = Base × (Pn × Prob)
  Prob = Desc × Goal × SubP × OldP
  SubP = Pn $\overrightarrow{m}$ Prob
  OldP = Prob

τ:174

As treatment goes on original problem may change: It may be replaced by an altogether new formulation. Or it may be decomposed into separately treatable sub–problems. Or both.

Patient medical records need be studied using computing science principles and techniques.

4.1.3 Bill–of–Lading

τ:175

Bills–of–lading seem the core document of logistics.

A bill–of–lading consists of administrative information, base information (weight, physical measures, value, etc.), a hub of origin, and a list of next hubs with "in–between" conveyor departure and arrival times and conveyor index.

type
  BoL = Admin × Base×HIdx×(T×CIdx×T×HIdx)*

4.1.4 Product Catalogue

τ:176

Product price or service fee catalogues seem the core documents of electronic business.

To product (or service category) names correspond administrative information, a price/fee table, and delivery conditions (including times). A price/fee table may designate quantity rebates.

type
CTLG = Pn $\overrightarrow{m}$ Admin×(Q $\overrightarrow{m}$ Itm_price)×Del

4.1.5 Discussion

τ:177

We have — ever so briefly — hinted at some infrastructure component document types. We next look at two facets of such documents: Their being originals and copies, etc. And their being displayed, in the domain or by computer.

4.2 Documents: Originals, Copies, Editions and Physics

τ:178
lisboa/infodocs

Another set of facets of document creation and flow handling are those of copies versus originals, copies of copies, a veritable hierarchy of documents, and their versions.

4.2.1 Originals, Masters and Copies

Narrative: Let there be a notion of an original (document). And let there be a notion of copying a document, from a master, whether original or already a copy. Let there be the notions of global time and locations. Originals are created from information at a time and at a location. Copies are made from (master) documents at a time and at a location.

τ:179

Formalisation:

type
I, T, L
D′ == O | C
O == mkOrig(τ:T,ℓ:L,i:I)
C == mkCopy(τ:T,ℓ:L,d:D)
D = {| d:D′ • wfD(d) |}
value
wfD: D′ → **Bool**
wfD(d) ≡
cases d **of**
mkCopy(d′) → τ(d) > τ(d′), _ → **true end**

τ:180

Comments: The above model represents, perhaps more a requirements model than a domain model. This is so since the model describes the property that from any document one can uniquely observe whether it is an original or a copy, and, if a copy, one can observe the time and place of the copying and the document from which it was copied is "kept intact" ! Nothing is lost.

value
>: T × T → **Bool**
create: T × L × I → O
copy: T × L × D → C
axiom

```
∀ t,t':T •
   t>t' ∧ ~(t=t'∨t'>t)
   t'>t ∧ ~(t=t'∨t>t')
   t=t' ∧ ~(t>t'∨t'>t)
```

τ:181

Axioms: In general we can express the following properties that are not directly modelled, but can be described through axioms: No two documents can be made: Created or copied, at the same time and (conjunction) location. If two documents are copied from the same (ie. third) document then they are copied at either different locations at the same time, or at the same location at different times, or at different locations and different times.

```
axiom
   ∀ d,d':D •
      d ≠ d' ⇒
         ~(τ(d) =τ(d') ∧ ℓ(d)=ℓ(d'))
         ∧ ((τ(d)=τ(d') ∧ ℓ(d)≠ℓ(d'))
            ∨ (τ(d)≠τ(d') ∧ ℓ(d)=ℓ(d'))
            ∨ (τ(d)≠τ(d') ∧ ℓ(d)≠ℓ(d')))
```

4.2.2 Editions

τ:182

Documents, once created or once copied, can be subject to editing. What may appear to be physically one document may be subject to series of repeated edits and/or the basis for repeated copying.

```
type
   Δ = O | C | E
   C' == mkCopy(τ:T,ℓ:L,δ:Δ)
   C = {| c:C' • wfΔ(c) |}
   E' == mkEdit(i:I,τ:T,ℓ:L,δ:Δ)
   E = {| e:E • wfΔ(e) |}
value
   wfΔ: (C'|E') → Bool
   wfΔ(mkCopy(t,_,δ)) ≡ t>τ(δ)
   wfΔ(mkEdit(i,t,_,δ)) ≡ t>τ(δ)
   edit: I × T × L × Δ → E
   edit(i,t,l,δ) ≡ mkEdit(i,t,l,δ) pre t>τ(δ)
```

The story on copying etcetera should now be modified.

4.2.3 The Physics of Documents

τ:183

Narrative: No two documents can occupy the same physical, including electronic, ie. spatial location. This goes for originals, copies and editions. The spatial location of a document may change, dynamically. To locate a document is to observe that it is indeed in some spatial location. We could decide to make the spatial location a property of the observer, the one

who"finds", who locates a document, and not of that document. Or we could decide, vice–versa, that with every document there is an inert dynamic property, namely its spatial location. If we choose the latter we become involved in the following modelling:

τ:184

Formalisation of Document Locatability:

```
type
  DOC, L
value
  obs_L: DOC → L
  obs_D: DOC → D
  copy: T × DOC → DOC
  next_to: L × L → Bool
axiom
  ∀ t:T,doc:DOC •
    let doc′ = copy(t,doc) in
      ℓ(obs_D(doc′)) = obs_L(doc′) ∧
      next_to(obs_L(doc),obs_L(doc′)) end
```

The axiom expresses that at the time a document is copied the copy location (ℓ(obs_D(doc′))) is that of the new document location (obs_L(doc′)), and the location of the master and its copy are located next_to one another. The mereology of 'next_to' is an interesting one to study !

τ:185

Monotonicity: Time progresses, it is assumed "smoothly", hence observable document locations, if they change, change accordingly. Thus a monotonicity axiom is required. Let us assume that from a document we can also observe the global time, and project it, at any time, onto its base document (one that is "stripped" of global time and spatial location):

```
value
  obs_T: DOC → T
  proj_D: DOC → T → D
axiom
  ∀ doc:DOC, t,t′:T •
    let (d,d′) = (proj_D(doc)(t),proj_D(doc)(t′))
    in version(d,d′) end
```

4.2.4 Discussion

τ:186

The models above, of temporal properties of documents, is problematic — and should probably be tackled rather more profoundly. Probably we ought instead introduce a notion of time–varying function over documents:

```
type
  Doc, T
  DOCS = T → Doc
```

value
  obs_L: Doc → L
  ...

etcetera.

In general we need, however, resort to more seriously worked out ontologies of time and mereologies of parts and wholes. But this takes us too far in this paper.

4.3 GUIs: Graphic User Interfaces and Databases

τ:187
lisboa/infogui

We have seen them: Nicely formatted GUIs: Graphic User Interface (computer screen) "windows", with "clickable" icons, scroll–down curtains, sub–windows, and tables with rows and columns and all of these being either "on" or "off" (ie. "clicked" or "not clicked"), with explanatory texts, and with possibly associated values that the user can "fill in". We now present a formal model of a "rich" class of GUIs.

4.3.1 GUIs: Graphic User Interfaces

τ:188

The GUI Display: All icons, curtains, sub–windows and tables have names:

type
  GUI = Txt × Icns × Crts × Wins × Tbls
  Icns = In $\overrightarrow{m}$ Icon
  Crts = Cn $\overrightarrow{m}$ Curt
  Wins = Wn $\overrightarrow{m}$ Wind
  Tbls = Tn $\overrightarrow{m}$ Tabl
  Icon = OnOff × Txt × VAL
  Curt = OnOff × Txt × Disp*
  Wind = OnOff × Txt × GUI
  Tabl = OnOff × Txt × REL
  Disp == mkI(i:Icon) | mkC(c:Curt)
    | mkW(w:Wind) | mkT(t:Tabl)
  REL = TPL-**set**
  TPL = An $\overrightarrow{m}$ VAL
  OnOff == off | on

Curtain entry "values" may be GUI windows themselves.

τ:189

GUI Types & Values: For end–users to design own GUIs tools are provided, tools which imply a GUI type concept:

type
  GTyp = IcnTyps × CrtTyps × WinTyps × TblTyps
  IcnTyps = In $\overrightarrow{m}$ IcnTyp
  CrtTyps = Cn $\overrightarrow{m}$ CrtTyp
  WinTyps = Wn $\overrightarrow{m}$ WinTyp
  TblTyps = Tn $\overrightarrow{m}$ TblTyp
  IcnTyp = Typ

CrtTyp = DTyp*
WinTyp = Gtyp
TblTyp = RelTyp
DTyp == mkITyp(it:IcnTyp)|mkCTyp(ct:CrtTyp)
| mkWTyp(wt:WinTyp)|mkTTyp(tt:TblTyp)
RelTyp = An $\overrightarrow{m}$ Typ
Typ == integer | boolean | text | character
VAL = **Int** | **Bool** | **Text** | **Char**

GUI type concept implementation amounts to a GUI window, usually with border icons, etc., for "clicking, dragging & dropping" designed window entities.

4.3.2 Data Bases

r:190

Window (icon, table) values usually reflect values of fields and rows of relational database types:

type
RelDB = (Rn $\overrightarrow{m}$ RTyp) × (Rn $\overrightarrow{m}$ RELN)
RelTyp = An $\overrightarrow{m}$ ETyp
RELN = RTpl-**set**
RelTpl = An $\overrightarrow{m}$ EVAL
ETyp == integer | boolean | text | character
EVAL = **Int** | **Bool** | **Text** | **Char**

r:191

The GUI window icon and table values displayed are obtained by attribute (An) reference to unique key value (KeyTpl) designated tuples of named relations (Rn) of an underlying relational database.

type
TblTyp = Rn
Typ == integer | boolean | text | character | mkR(r:Ref)
Ref = Rn × KeyTpl × An
KeyTpl = RelTpl

4.3.3 Discussion

r:192

Notice a "homomorphism" between GUIs (ie. "values") and types. Appropriate (primitive) operations can now be defined for effecting value reflections: Screen display vs. database contents, for updating the database "through" screen updates, *&c.*

We leave the explication here.

4.4 Discussion

r:193
lisboa/infodis

We have lifted a veil over some non–standard ways of looking at types and values, and we have sketched a general GUI vs. database mechanism.

In our forthcoming text book ([3]) we bring the above type/value design ideas together with the work flow system design concept shown earlier.

5 Conclusion

τ:194
lisboa/conc

5.1 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 sketching claimed engineering disciplines of software development: Denotational, concurrency, type/value, logic, agents and language–based knowledge engineerings.

τ:195

The type/value GUI example also reflected a denotational engineering facet: A (GUI) type denoting a possibly infinite collection of values (ie. GUI windows).

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* [10].

An Apology: It is lamentable that my examples did not illustrate uses of other than `RSL` [5]. It ought also have contained examples of uses of one or another `Duration Calculus` [11, 12, 13, 14, 15, 16, 17]. I apologise.

5.2 "What is an Infrastructure ?"

τ:196

An infrastructure is a collection of infrastructure components. There is synchronisation and communication between and within the components. We have shown only the latter.

An infrastructure component is a language: The professional, specialised jargon language spoken by professionals and users of the infrastructure component. We have focused on several such languages: The language of "the market", whether ordinary or electronic; the language of logistics, whether ordinary or electronic; the language of transaction scripts and directives, whether ordinary or electronic; *&c.*

τ:197

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 !

5.2.1 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 !

6 Bibliographical Notes

τ:198

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. It contains so many relevant papers and references that the below should suffice.

References

[1] Dines Bjørner. *"What is a Method ?" — A Essay of Some Aspects of Domain Engineering*, chapter 9, pages 177–205. IFIP WG2.3. Springer, New York, N.Y., USA, 2002. Programming Methodology: Recent Work by Members of IFIP Working Group 2.3. Eds.: Annabelle McIver and Carroll Morgan.

[2] Dines Bjørner. Domain Engineering, Elements of a Software Engineering Methodology — Towards Principles, Techniques and Tools — A Study in Methodology. Research report, Dept. of Computer Science & Technology, Technical University of Denmark, Bldg. 343, DK–2800 Lyngby, Denmark, 2000. One in a series of summarising research reports [18, 19].

[3] Dines Bjørner. *Software Engineering: Theory & Practice.* (Publisher is being contacted), 2002. These Lecture Notes represent the author's *Chef d'Œvre* — the summary of more than 25 years of research, development and teaching.

[4] 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.

[5] 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.

[6] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[7] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[8] A.W. Roscoe. *Theory and Practice of Concurrency.* Prentice–Hall, 1997.

[9] Steve Schneider. *Concurrent and Real-time Systems — The CSP Approach.* Worldwide Series in Computer Science. John Wiley & Sons, Ltd., Baffins Lane, Chichester, West Sussex PO19 1UD, England, January 2000.

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

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] Dines Bjørner. Requirements Engineering, Elements of a Software Engineering Methodology — Towards Principles, Techniques and Tools — A Study in Methodology. Research report, Dept. of Computer Science & Technology, Technical University of Denmark, Bldg. 343, DK–2800 Lyngby, Denmark, 2000. Not yet released. Meanwhile refer to [3]. One in a series of summarising research reports [2, 19].

[19] Dines Bjørner. Software Design: Architectures and Program Organisation, Elements of a Software Engineering Methodology — Towards Principles, Techniques and Tools — A Study in Methodology. Research report, Dept. of Computer Science & Technology, Technical University of Denmark, Bldg. 343, DK–2800 Lyngby, Denmark, 2000. Not yet released. Meanwhile refer to [3]. One in a series of summarising research reports [2, 18].

7 Figures for Section 3.2

r:199

Figure 14 on the next page shows a net, two lines, two stations, and around 64 units: nine switches, one simple crossover, one switchable crossover, and 11 tracks.

Figure 15 on the following page diagrammatically abstracts instances of the four most basic forms of units: A linear unit with two connectors, a simple switch unit with three connectors, a switchable crossover with four connectors, and a simple crossover with four connectors.

Figure 16 on page 53 shows the possible states of two kinds of units: The four possible states of a linear unit, and the nine possible states of a switch unit. Actual linear or switch units need not span all these states.

Figure 17 on page 53 shows the possible directional captures and "freeings" of units during movements: Situation [0] (0) depicts an initial train position. In situation [1] (1) no captures nor "freeings" have occurred during actual movement. In situation [2] (2) Capture of one unit has occurred wrt. [1] (1). In situation [3] (3) "freeing" of one unit has occurred wrt. [2] (2). In situation [4] (4) Both capture of one and "freeing" of one unit has occurred wrt. [3] (3).

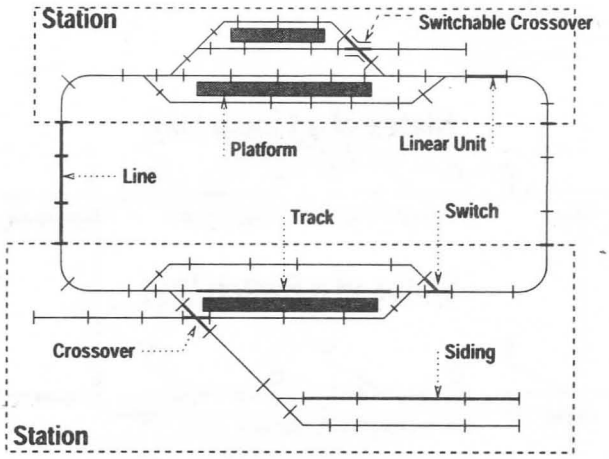

Figure 14: A Sample Railnet

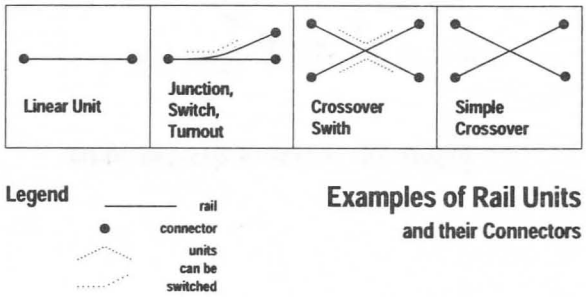

Figure 15: Four Rail Units

Figure 18 on page 54 shows a simple route from one platform track in one station, to a siding track in another station. The figure labels all the route units: $u1 - u20$.

Figure 19 on page 54 shows nine trains at two "neighbouring times": Three have not moved: $tn3, tn4, tn10$. The others have.

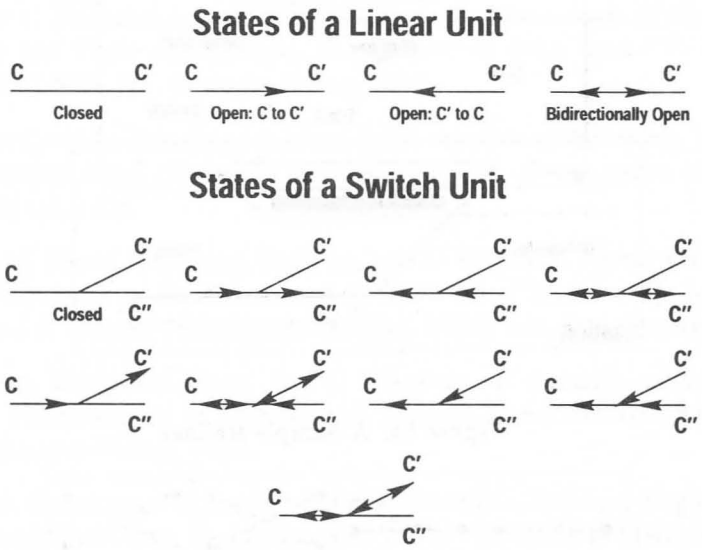

Figure 16: States of Sample Units

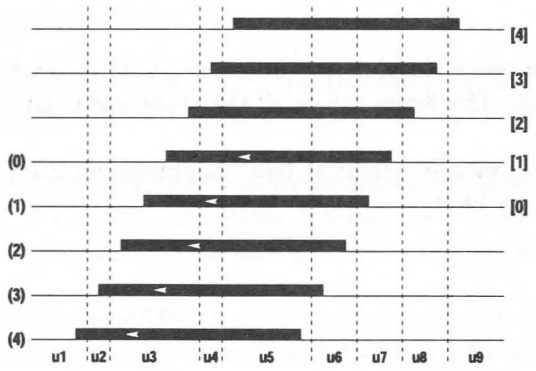

Figure 17: Possible Train Movements

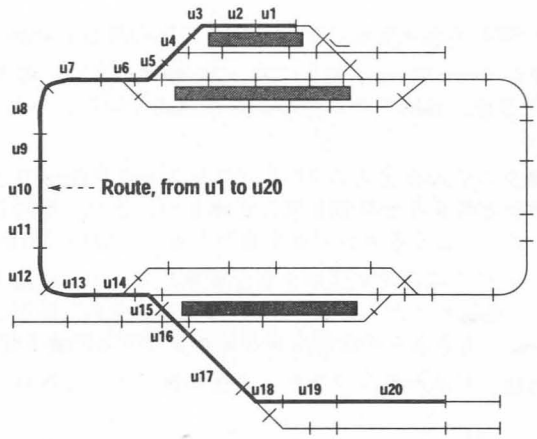

Figure 18: Route of a Rail Net

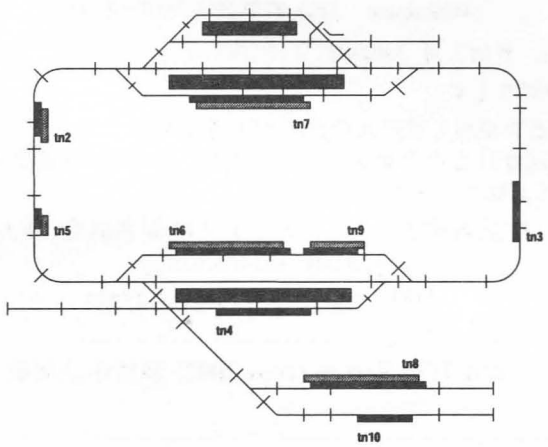

Figure 19: Train Traffic

SEA　特別 Forum (June, 2002)

IT インフラストラクチャの構築における情報工学の役割

ー コンピュータサイエンスの未来 ー

参 加 者 募 集

ひさしぶりに東京で開催される SEA の年次総会に先だって，表記の特別 Forum(会員限定)を開催します．

講師は，デンマーク工科大学教授の Dines Bjorner 先生です． Bjorner 先生は，数年前まで国連大学ソフトウェア研究所 (マカオ)の初代所長として在任され，ISFST その他の国際会議について，われわれ SEA の活動にいろいろご協力いただきました．

今回の Forumでは，未来の国家システムを支える IT インフラストラクチャの分析や構築にさいして，コンピュータサイエンスやソフトウェア工学がどのような役割を果たすべきか，また，そのことがこれらの分野における研究や教育あるいは実践のあり方にとってどのようなインパクトを与えるかについてお話しいただきます．

IT インフラストラクチャとは，人びとの日常生活や主要な産業のビジネス活動を支援する社会経済的な諸要素の集合であり，それには交通システム，金融サービス産業，病院/福祉システム，E-ビジネス支援環境などが含まれます．それがどのような重要性を持つかは，さきごろの金融システムのトラブルの例を見るまでもなく，自明の事実です．

そういう意味で，今回の Forumは，われわれソフトウェア技術者にとってきわめて有益なものだと思います．奮ってご参加ください。

********** 開催要領 **********

1. 日 時： 2002年 6月 17日 (月) 13:30 ～ 17:00

2. 場 所： 労働スクエア東京 第701会議室 (東京・中央区 新富1-13-14)

3. プログラム (予定)：

13:00～13:30	受付
13:30～15:30	**講演：Informatics of Infrastructure (A Future of Computer Science)** Prof. Dines Bjorner (technical Unversity of Denmark) [逐次サマリー通訳：岸田孝一 (SRA-KTL)]
15:30～16:00	Break Time
16:00～17:00	自由討論: **IT インフラストラクチャをめぐる諸問題** Discussant: SEA 幹事会メンバー有志

4. 参加費： SEA 正会員 2,000円，賛助会員 3,000円

5. 定 員： 50 名 (先着順にて締切ります)．

6. 申込み方法： 下の申込用紙に必要事項を御記入の上， SEA 事務局まで E- Mail お申込みください．会場地図等をお送りします．なお，参加費は当日会場受付にてお支払いください(領収書を差し上げます)．申込受付後のキャンセルは原則としてお断りします．

申込み宛先： ソフトウェア技術者協会(SEA)
E-Mail: sea@sea.or.jp
URL: http://www.iijnet.or.jp/sea

.....

SEA 特別 Forum (June 2002) 参加申込用紙

氏名(ふりがな)：＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿ (＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿)

会社名：＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿

部門・役職：＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿

住所：(〒＿＿＿＿＿) ＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿

Tel: ＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿ Fax: ＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿

E-Mail: ＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿

種別(該当欄にチェック): □ 正会員(No.＿＿＿＿＿＿) □ 賛助会員(No.＿＿＿＿＿＿)

参加費：＿＿＿＿＿＿＿＿ 円

ソフトウェア技術者協会
〒160-0004　東京都 新宿区 四谷 3 - 12 丸正ビル 5F
Tel: 03 - 3356 - 1077　Fax: 03 - 3356 - 1072
E-mail: sea@sea.or.jp
URL: http://www.iijnet.or.jp/sea