Issues in International Cooperative Research Why not Asian, African or Latin American 'Esprits'?*

Dines Bjørner, Technical University of Denmark†

ABSTRACT

Joint international research in software engineering stands low in contrast to joint research in computer science.

In this personal account of more than 25 years of experience in joint international research projects around or based on formal methods the author reviews a number of facets of past joint R&D projects. A proposal for a framework for regional collaborative research in software engineering: methods and technology is finally put forward.

An extensive, commented software engineering terminology is included. It can be referenced for terms not otherwise defined. An extensive bibliography gives useful references. An extensive index should likewise prove useful.

1 Introduction

This note briefly examines some of the issues of international research collaboration in software engineering.

In the main it seems that more resources, than for "one-on-one" computing science research, are needed in order to achieve useful results. Software engineering seeks practically useful results, so they require a large audience.

International collaboration in software engineering research received, it is claimed, a significant, positive boost, with the European Community's ESPRIT (European Strategic Programme in Information Technology) programmes, incl. the BRA (Basic Research Action). We will briefly review our own, mostly positive experience from our long and deep participation in this programme. And we will comment on what seems to be the current state of this programme.

The United Nations University established in 1992 an International Institute for Software Technology, UNU/IIST, located in Macau, but active on four continents: Asia, South America, (Eastern) Europe and Africa. We will briefly outline what we consider UNU/IIST's success in propagating research and software engineering methods to more than 20 developing countries and countries in transition.

The SEA (Software Engineering Association of Japan) has conducted a most commendable joint programme on sound software engineering practice by organising a number of international symposia in Far and South East Asia, notably China. UNU/IIST has been supported by these symposia. We very briefly review this SEA effort.

All this brings us to suggest that regions or continents, as in Europe, try establish local (regional) versions, i.e. adaptations of the ESPRIT/BRA programmes.

We finally asses the possibilities of success of such programmes.

2 Software Engineering vs. Computer Science Research

By software engineering we understand the practice of domain engineering, requirements engineering and software design.

By computer science we understand the study of and knowledge about the 'things' (data and pro-

*Invited presentation for a similarly titled panel at the Asia pacific Forum on Software Engineering Monday 20 April 1998, Kyoto, Japan — in connection with ICSE'98.

†Software Systems Section, Department of Information Technology, DK–2800 Lyngby, Denmark. Fax: ++ 45-45.88.45.30; E–Mail: db@it.dtu.dk; Web: http://www.it.dtu.dk/~db

cesses) that can reside inside computers (what they are).

How does software engineering relate to computer science? Well, through computing science, or as we shall here call it: programming methodology. By computing science, cum programming methodology we understand the study of and knowledge about how to construct these 'things' (how to build them).

Software engineering centers around programming methodology but incorporates a great many other facets, usually taken over from traditional engineering.

The problem is, however, that software engineering is not easily comparable to other engineering disciplines. There are many reasons for this:

- **Science, Engineering & Technology:**

  Engineering is the process of "walking" between science and technology, that is: of creating technology based on scientific insight, and, vice-versa, analysing technologies in order to ascertain their possible scientific value (e.g. UML, Java).

- **Computability vs. Mother Nature:**

  First we have that no laws of nature govern the properties (behaviour) etc. of the data and processes inside computers. They are instead subject to the mathematical laws of recursive function theory (meta-mathematics, mathematical logic).

- **A Different Experimental Science:**

  Secondly, since the inevitable experiments by the practitioners of computer science and programming methodology are not ruled by laws of natural sciences, these experiments are much more like those of mathematicians. But they are, especially for programming methodology researchers, different in that they are method and methodology oriented. The mathematician mostly wishes, risking a rather superficial statement, to capture new mathematics or exciting theories around older math. The programming methodology researcher is primarily — or ought primarily to be — interested in principles, techniques and tools of programming methods and relations between such.

  The (theoretical) computer scientist is much more like the mathematician: wishing to build new theoretical models of various computing aspects: type theories, concurrency, distribution, "knowledge representation", proof theories, etc.

  In my mind much too few researchers, everywhere, are concerned with programming methodology. In Europe we see England, with Oxford as a leading place, as a country where programming methodology stands very high on the agenda. France has in later years, where they had one of the strongest schools in theoretical computer science, also moved into formally based, or at least formally understood software engineering. In my opinion, quite a nice and good surprise. In Denmark, besides my own influence, and no doubt also due to some early directions taken by Peter Naur, we are also strong in programming methodology. In the US they are strong at the two ends of a perceived spectrum from theoretical computer science, at one end, to very practical, non-programming methodology oriented software engineering, at the other end — but not in programming methodology!

- **Method & Methodology:**

  The terms method and methodology are important in this paper.

  By a method we understand a number of principles for analysing, selecting and applying a number of techniques and tools in order efficiently to construct an efficient artifact (here software).

  By methodology we understand the study and knowledge about methods.

  Since there is no one method that will bring us safely through any reasonable sized software development project we have to rely on several methods: methods for the "grand state" design of a large, distributed system, methods for the "nitty gritty" design or real-time,

safety critical and electronic equipment embedded components of the larger system, and so on.

- **'Formal Methods':**

  A software development method may be characterised as formal if the following holds: (i) the specification and programming languages are formal (have a formal syntax, a precise formal mathematical semantics, and a formal, mathematical logical proof system); and (ii) there are sets of rules (i.e. design calculi) that help guide or support the developer in transforming abstract specifications to executable code. Thus, by a formal method we do not mean that the principles are formal, only that some of the techniques and tools are. One observes that we really mean: formal specification and calculation when we say formal method.

  Programming methodology researchers study both the informal and the formal aspects of formal methods. It is somehow easier to study and produce papers on the formal aspects: yet another formal specification language, yet another twist to the mathematical semantics or the proof system of such a language, etc. It is more difficult, it seems — and at least to this author, whose primary occupation it is — to combine the study of the formal aspects of so-called formal methods with the informal ones: the principles of specification, calculation, abstraction and of modelling.

- **Descriptions at All Levels:**

  There are no physical artifacts when a software developer is at work.There are descriptions and descriptions and descriptions and ...:

  - **Domain Descriptions:**

    In our mind, proper software development is based on, or first develops a proper description of the application domain — a description which is void of any reference to not-already existing computing (and IT) in the domain. Such a domain description is usually developed in stages of development: from gross assumptions ("big LIES") via lesser and lesser assumptions ("increasingly smaller Lies") to a domain description that captures "all that is relevant" at least to the next development stage of requirements (whereby the limit of all the lies is said to be the truth!). Usually we develop a domain description which covers a broader (sometimes and preferably much broader) spectrum than needed for the next, the requirements stage. The domain is relatively invariant.

  - **Requirements Descriptions:**

    From a proper domain description we develop, also in close interaction with the stake-holders of the domain, the requirements. Again a set of increasingly more precise descriptions. Terms of the requirements are terms of the domain and must therefore be precisely described in the domain description. The requirements description is void of any reference, in principle, as to how the desired (required) software is to be implemented. The requirements are relative fixed.

  - **Software Design:**

    Software design — like domain engineering and like requirements engineering — involve many steps of development (depending on the way in which these steps are carried out, referred to as refinement).

    * **Software Architecture:**
      Now, given the computing platform constraints, the software architecture describes, in committed ways, the external interfaces and the observable behaviour over these. That is: all the things that human users or other software (either already or to be elsewhere developed) can experience. Again descriptions transpire. The software architecture is much less fixed than the requirements.
    * **Program Organisation:**
      And we could go on: to program organisation descriptions which de-

scribe all internal interfaces — often imposed in order to exploit existing platforms, or to ensure efficiency, or to ensure security, etc., etc. The program organisation possibilities are legio!

* **Etc.**

The software developers are "creating" these descriptions, formally or informally — and if formally, then in a spectrum from systematic, via rigorous to formal. The software developer reasons about the text: proves — informally, model-theoretically or proof--theoretically, or checks (model-theoretically) — properties about the individual specifications (descriptions, texts) or about transitions (relations) between documents (such as correctness, validation, etc.).

Lest it should be overlooked, the author of this invited keynote paper, is of the strong opinion that:

- Much too little research in done in most countries in the area of programming methodology.
- Much too little research and transfer is done in many countries wrt. formal methods.
- Much too many so-called theoretical computer science researchers are wasting our time, their time and somebody's money in contributing to the above!

  Namely by not understanding that computer science is both a mathematical and an engineering discipline and they must alternate!

We owe our jobs in universities to the actual use of computers, to the actual production of both hardware and — to us, worldwide — of software.

3 Three International R&D Efforts

3.1 The European Community ESPRIT Programme

The ESPRIT[1] Programme, of which the 5'th three–four year Framework is now being initiated has changed the European research scene in computer science, programming methodology and software engineering. It has mostly benefited the software engineering, then the programming methodology, and — rather sparingly — the computer science communities, as could be expected.

ESPRIT projects were primarily concerned with pre-competitive R&D projects in IT, including Software.

3.1.1 The Main Programme I have myself had projects in the precursor to ESPRIT (the Multi--Annual [IT] Programme) and in ESPRIT. I was one of the instigators of the ESPRIT 415: RAISE, and the ESPRIT 5383: LaCoS projects. The former created the RAISE Method, the RAISE Specification Language, RSL, a successor to VDM (and hence also, really, VDM-SL), and the RAISE Tool Set. The latter project tested out RAISE on seven medium scale software industries in Europe. RAISE stands for *Rigorous Approach to Industrial Software Engineering*. LaCoS stands for *Large scale development of Computing Sysrems using formal methods*. I have taken part in other projects as well. Common to all these projects are (and were) that they involved at least one industry partner, at least two, legally unrelated entities from different community countries, and that often they had a large proportion of academic, typically university institute or department participation. Financing was 50% community coverage of standard costs plus overhead. For an industry it meant a 50% contribution if the project was in line with that industry's R&D plans. For a university the other 50% were our ordinary salaries plus the university overheads. Thus it meant that university groups often could hire several researchers to work almost exclusively of the ESPRIT projects. As European academics also staffed the ESPRIT boards and took part in the pre-, ongoing and post-

[1] ESPRIT: European Strategic Programme for Research in IT, Information Technology

evaluation it meant that we had a beneficial influence over topics and research methodologies.

Many were the initially uneven partners ("strange bed-fellows") that were brought together. As years went on partners became "in-bred", knew each other from beforehand and worked well together. In some instances one had to "sleep" with partners, i.e. endure such, which did little or no work. Because of differences in national accounting practices we all witnessed some rather creative such techniques.

On the whole I can recommend any region of the world to consider a transplant — a totally business process re-engineered version, however!

3.1.2 The Basic Research Actions The BRA (Basic Research Action) was a subpart of ESPRIT geared more to research than to actual prototype, pre-competitive product development.

I was involved with one such project: **ProCoS** for Provably Correct Systems. Instigated by Tony Hoare it had two phases of which I was the director of the first, very successful phase.

It brought leading programming methodology researchers together regularly from three different countries and six different universities. Although there were culture differences, nationally and scientifically (some were more computer science than programming methodology oriented, etc.), the research productivity was very high and probably much, much higher total than if the six had each been left to their own devices.

Through the external review process, often with scientists from other community countries than the partner countries, results were critically assessed and, when appropriate, rapidly disseminated.

I do not believe that any national R&D programme of the kind that ESPRIT enabled could be as successful. At least in my country I see such programmes being far too political with the result that they really do not produce much in terms of research, let alone new results.

3.1.3 Political Demands The ESPRIT and the ESPRIT/BRA programmes have now run for more than 12 years. On one side they have brought much scientific and technological progress, on another side they may not have brought Europe much farther away from what some European Commission staff may characterise as a US dominance in software.

The political demand, in all areas of ESPRIT have therefore been increasingly tuned to an agenda of significant impact on European IT by European Software, and a similar impact of European IT on the world.

When science and politics meet, science becomes the looser. When engineers and politicians gather nothing very good results.

3.1.4 An Assessment of ESPRIT + ESPRIT/BRA In the European Multi-Annual and in the ESPRIT programme projects (in which I was involved) we transferred our research into practical, engineering tools, but mostly methods and research results. Thus we were able to make a significant quantum step ahead, of a size that would have been unthinkable in a purely Danish technology or science support environment: (i) **Formal Definition of Ada** (jointly with notably Prof. Egidio Astesiano of Genova, Italy), (ii) **Ada Compiler Development**, (iii) **Formal Methods Assessment**, (iv) **RAISE** (jointly with notably STL [Standard Telephone Ltd., Harlow, UK]), (v) **LaCoS** (jointly with STL's successor: BNR [Bell Northern Research]), (vi) **ProCoS** (jointly with groups at Oxford Univ., Royal Holloway/London Univ., Kiel Univ., Oldenburg Univ., Århus univ.), etc. There is no doubt that were it not for these European Programmes formal methods might not have been as advanced as they actually are today — in Europe. Which brings me to also mention that for 10 years (1987–1998) the European Commission supported first VDM Europe, then its successor Formal Methods Europe (FME). VDM Europe/FME serves, still, to help industry be increasingly aware of formal methods and their tools. There is now a whole industry of formal methods consulting firms spread out over Northern Europe. I doubt it would have been there were it not for ESPRIT. And I am not exactly a fan of the EU!

So perhaps we have seen the good days of ESPRIT and ESPRIT/BRA in Europe. While it lasted it was good. Many closet engineers came out of the university cloak rooms, but much good research

also got a fine chance to be transferred into industry. What Europe, as a whole, lacked was the close relations that US universities, when among the best, have with US industry.

I seriously believe that these European IT cum CS+PM+SE programmes have helped and will continue to help European industry.

3.2 UNU/IIST

I was the first and founding UN Director of UNU/IIST. From its start 2. July 1992 and for exactly five years. As for our strategy and tactics, we took, guided also by my then Deputy, now the full, Director, Professor Zhou Chaochen, a research and post-graduate direction that owed much to the "spirit" in which we had seen several ESPRIT and especially ESPRIT/BRA projects flourish.

We established a number of joint university, industry and UNU/IIST R&D projects. In China, Vietnam, the Philippines, and elsewhere. With rather practical goals: software to support the dispatch of trains in China, software for the Ministry of Finance in Vietnam, models of and a foundation for software for a radio telephony system for The Philippines, an analytical basis for software for a generic manufacturing industry in The Philippines, China and Brazil, a formal basis for a multi-script processing system for Mongol (combined with any other language you may so wish), etc. In these projects Fellows from these and other countries would visit UNU/IIST for periods of at least 10 months, some more, to learn formal software engineering methods (PM) and to draft the most significant documents of their engineering life so far: extensive domain models of the application area, of requirements as derived from these, etc., on to software. RAISE [34, 35] became a focal point as it, as a method and with its specification languages, RSL, and tools, provided amply sufficient support for almost all the work. Where RAISE was insufficient we turned, naturally, to the Duration Calculi, see next!

In addition we established a training programme in computing science research around the scientific and programming techniques for software for real-time, embedded systems. Here the theoretical work centered around the continuous time, integrable interval temporal logics of the Duration Calculi [20, 24, 23, 18, 19, 39, 22, 40].

Finally we established a programme in which we trained post-graduate fellows, usually lecturers, in transferring our advanced computing science and software engineering courses to their local universities.

UNU/IIST was, and is, very successful in this endeavour. A five-year review commission set down by the previous rector of UNU, came, however, to a rather different conclusion: we shouldn't propagate formal methods since they were not generally accepted in industry. We should give courses in object oriented programming, Java and the like. Well, I am glad that the board of UNU/IIST had the good senses to see through this strangely put request. The current UNU/IIST is — so far — continuing the good work of the past UNU/IIST. But I trust that the new Director, Prof. Zhou Caochen, will bring needed renewal and new facets to UNU/IIST. *Why shouldn't we give the best we know to the world? Why withhold from developing countries a leading scientific and technological direction? Why give them "crap"?* We are glad that all the groups with whom we worked in more than 20 developing countries, besides being acutely aware of the situation wrt. possible 'formal methods' controversies, all fully and enthusiastically adopted our programme: from China to Argentina, from Russia to Indonesia, from Gabon to India!

People from Asian industry and, especially Chinese universities, often asked me: how big is UNU/IIST. It was as if they expected to hear: 50–70 staff, etc. So when I proudly replied ten times less, they wondered. The trick behind our unquestionable success in propagating sound software engineering methods was that all my staff were tuned to exactly the same melody: they basically all understood computer and computing science and software engineering in the same way, they basically all used formal techniques: specification and calculation, as a matter of fact. We did not believe in formal methods — we just use it, further researched it, and taught it! Small is not only beautiful, but small is also very effective! And with the right tools and techniques one can convincingly tackle very large scale

systems.

3.3 SEA's Outreach Programme

The SEA: Software Engineering Association of Japan is — seen in the international light of ACM and IEEE — a truly unique institution. Over the years the rank and file of SEA have steadfastly organised an amazingly refreshing series of software engineering (CASE and other technology) symposia in China, and I have been the lucky participant of several of these events: in Urumqi, in QuFu, in Kunming. My colleagues at UNU/IIST have now taken over my participation. These symposia — conducted at a liberating low-key level — brought together leading software engineering researchers from China and Japan, and, thanks to SEA's strong network, also featured top US software engineering researchers. Thanks to a open minded spirit many young Chinese and Japanese researchers first presented their papers at these symposia which also featured good, professional discussions.

We all owe a great deal to the opening up of international collaboration in software engineering research to Kouichi Kishida–sensei's steadfast guiding spirit, gentle mind, philosophical inclination and long-range foresight.

4 A Critique

Not all is 'rosy': perceived problems hinder internationalisation of software engineering research.

Which, then, are these problems? Some are:

- **Software Engineering & Technology Hypes**

  Engineering "walks the bridge" between science and technology — both ways: creating technology based on scientific insight; analysing ("industrial archaeology") some technologies in order to retrieve scientific values.

  Many so-called software engineering researchers are guided more by current commercial technologies than by any substance of scientific content. Hypes such as Object--orientedness are considered hypes by a scientific community which fails to see how these so-called technologies relate to serious scientific insight. As an example, most, if not all object-oriented "methods" fail to deliver abstraction, compositionality (refinement) and logical reasoning. That is: where software abstraction techniques has been seen as a major contributor to cleaner and clearer software developments and designs most object--oriented techniques do not provide for abstraction. Once an object-oriented technique has been applied and a document created, this technique does not, in general, allow for refinement: transforming easy-to-understand abstractions into descriptions for presumably efficient software. And finally no techniques are offered by any object-oriented technique (that I am aware of) for reasoning over object-oriented specifications (i.e. no means for proving properties). This puts object-orientedness significantly apart from the main stream of programming methodology, or is it vice-versa ? There may be wonderful insight in these object-oriented techniques, but is it not sad that they are primarily carried by such commercial, anti-science attitudes?

- **Formal Methods vs. Ad Hoc Methods**

  Tony Hoare has expressed:

  - **Maturity:** Use of a formal method is no longer an adventure; it is becoming routine.
  - **Convergence:** The choice of a formal method or tool is no longer controversial: they are chosen in relation to their purpose and they are increasingly used in effective combination.
  - **Cumulative progress:** Promise of yet further benefit is obtained by accumulation of tools, libraries, theories, and case studies based on the work of scientists from many schools which were earlier considered as competitors.

  In [38] Anthony Hall lists and dispels the following seven formal method "Myths":

  1. *Formal Methods can Guarantee that Software is Perfect*

2. *Formal Methods are all about Program Proving*
3. *Formal Methods are only Useful for Safety-Critical Systems*
4. *Formal Methods Require highly trained Mathematicians*
5. *Formal Methods Increase the Cost of Development*
6. *Formal Methods are Unacceptable to Users*
7. *Formal Methods are Not Used on Real, Large-Scale Software*

In [15] Jonathan P. Bowen and Michael G. Hinchey continue dispelling formal method myths:

8. *Formal Methods Delay the Development Process*
9. *Formal Methods are Not Supported by Tools*
10. *Formal Methods mean Forsaking Traditional Engineering Design Methods*
11. *Formal Methods only Apply to Software*
12. *Formal Methods are Not Required*
13. *Formal Methods are Not Supported*
14. *Formal Methods people always use Formal Methods*

And in [16] Jonathan P. Bowen and Michael G. Hinchey suggests ten rules of formal methods software engineering conduct:

I. *Thou shalt choose an appropriate notation*

II. *Thou shalt formalise but not overformalise*

III. *Thou shalt estimate costs*

IV. *Thou shalt shall have a formal methods guru on call*

V. *Thou shalt not abandon thy traditional development methods*

VI. *Thou shalt document sufficiently*

VII. *Thou shalt not compromise thy quality standards*

VIII. *Thou shalt not be dogmatic*

IX. *Thou shalt test, test, and test again*

X. *Thou shalt reuse*

Despite the above well-worded de--mystifications based on rather extensive evidence we still see parts of the software engineering world being very slow on the uptake.

Ad hoc ways of developing software, i.e. ways in which no proper use is made of available mathematics, still abound. In Denmark we say: OK, that's fine, then our software industry will indeed have at least one competitive edge when the real fight for software house survival sets in.

Examples of formal specification techniques and design calculi are:

B/AMN	[1, 49, 61]
CafeOBJ	[25, 54, 53, 31]
CoFI/CASL	[57, 58, 52, 17]
Duration Calculi	[20, 18]
Larch	[36, 37]
RAISE	[34, 35]
STeP/React	[50, 51]
VDM	[7, 47, 29]
Z	[59, 60]

- **Software Technology vs. Methodology**

  Programming methodologists, in addition to research into principles for selecting and applying techniques for the analysis and synthesis of domain, requirements and software design models, also need create tools. Typically such tools center around tools for the creation, analysis and transformation of domain, requirements and software models.

  Many so-called software engineering researchers waste their time, in my opinion, on building "tools" that do not have a proper methodological or scientific foundation. When they try show this "gadgetry" to me — "demo", I believe they call it — I have a very hard time understanding what is going on. While they are frantically hammering away at the keyboard and nervously clicking the mouse

etc., while they are excusing that the system reacts strangely just now, I am all the while asking for the one, two or at most three key concepts that best characterise their system. After half an hour the session usually is abandoned.

- **Pre-competitive Research:**

  Software engineering research typically requires a large-scale experiment around the development of actual, realistic software systems. This often necessitates commercial software house partners in addition to academic research groups. These commercial enterprises usually have a hard time understanding that one can indeed single out facets of the systems they wish to develop, facets that are "pre-competitive", that is: Whose openly inspectable development and the general awareness of this development does not hinder these enterprises in retaining their commercial rights nor their competitive edge. The European ESPRIT projects were very successful in this.

- **Confusion of Academic and Industrial Work**

  In many developing countries this is a serious problem and it is hard to tackle. But it is also a problem in most industrialised countries. The problem is that otherwise clever and scientifically serious research groups waste their talent on projects that ought better be carried out in industry. In developing countries such is often the case because the academy institutes and the university departments are extremely hard pressed to earn money. In industrialised countries some researchers — due to "fear of flying" — practice, under university protection, development which had better be done in industry. These "frustrated engineers"[2] attract the attention of the media and are often the "darlings" of their university chancellors. But no scientific progress is made!

  I pity those developing and other countries which tolerate this serious diversion, away from science, and in clear, but in all ways unfair and uneven competition with industry. The universities and academies are shooting themselves in the foot: they will loose years of research insight. They will miss generations of sufficiently well-educated candidates.

- **Cultural Differences ?**

  In my 20 years of lecturing around the world, outside the Judean/Hellenic/Christian world, I have found little evidence pointing in the direction that cultural: philosophical, religious, ethnic and other such factors materially influence basic attitudes towards the conception of software, research into its scientific foundations and its methodologies — in any but at most superficial ways. One such is treated next!

  But I may just be kidding myself. At least it could be a topic for discussion at the Asia Pacific Forum on Software Engineering Workshop, April 20–21, 1998 at ICSE'98.

- **One Country's Seeming Dominance?**

  The US, for better or worse, dominates the software engineering research scene, at least when seen from other places than Europe. Some years ago one could say, with some truth to it, that the disciplines, the topics, of theoretical computer science (complexity theory, etc.) and practical software engineering were the two main characteristics of US research in our area. It is only recently that US universities (etc.) have made significant contributions to programming methodology — except for significant research groups at CLInc. (the no longer existing Computational Logic Inc., Austin, Texas), SRI International (Menlo Park), Digital Equipment Corporations SRC (Systems Research Centre, Palo Alto), Stanford Univ. and CMU (Carnegie Mellon Univ.). There is something refreshing about US research in our area, but I seriously doubt its "hard nosed" attitude: *must have conclusive, experimental proof of superiority of formal methods over ad hoc techniques.* It seems that no allowance is given for the individual developers' intellectual satisfaction, feeling of elegance and sense of joy of conducting software development using formal techniques.

[2] Just like the many "frustrated mathematicians" who practice rather esoteric, some would say useless, theoretical computer science work

Although [38, 15] stands basically irrefutable many refuse to listen.

But the US state of affairs is rapidly changing.

5 Proposals

So what is our proposal. It is not formulated as a specific programme, only as an advice:

- **Advice #1:** *Intra– & Intercontinental Joint R&D*

  To boost local university research, and to make that research relevant to local industry while securing international relevance, create a number of "Path-finder" R&D projects that will bring university, academy and industry groups together around a common set of three objectives:

  1. New Software Technology for Industry
  2. New Software Methodology
  3. Increased Production of Professional Software Engineers

  Seek possible support from The World Bank *info***Dev** programme, from UNDP, ADB (Asian Development Bank) and others. Involve UNU/IIST in any such effort.

- **Advice #2:** *"Formal Methods"*

  To make a difference, and you'd better admit and accept it, base any such "path-finder" project on the simultaneously use of and research into further improved formal software development techniques (formal specification and design calculi) and tools.

  There are enough, and there will remain enough, software engineers that will try — in vain — to "fight" for the use of ad hoc methods. So to make a real difference join the real professional endeavour for responsible, trustworthy methods!

- **Advice #3:** *Infrastructure System Software*

  Let the "path-finder" projects evolve around:

  - **Infrastructure Systems Support**

    Examples of such infrastructure systems are:

    - *Transport Systems:*
      - *Railways* [9, 14, 21, 28, 13]
      - *Air Traffic* [5]
      - *Metropolitan Transport* [62, 55, 27]
      - *Shipping*
      - *&c.*
    - *Airline Business* [4]
    - *Manufacturing Systems:*
      - *The Market* [2, 45, 3, 44, 46]
      - *"Beyond CIM"* [32, 33]
      - *— including Robotics!* [8, 30]
    - *Government Administrative Systems*
      - *Ministry of Finance* [26]
      - *Ministry of Social Welfare*
      - *Health Care Systems*
      - *&c.*
    - *Financial Service Industry* [12, 48, 6]
      - *Banks*
      - *Insurance*
      - *Securities*
      - *Portfolio Management*
      - *Check* &c. *Clearing*
      - *&c.*
    - *Strategic, Tactical & Operational Resource Management:* [10, 56]
    - *&c.*

    To make sure that you understand what I am driving at, let me elaborate. The example is that of Railways.

    What is needed here, as in all the other examples, is a thorough understanding of the application domain void of any reference to requirements, let alone desired information technology. For railways this means, we believe, that domain models must be painstakingly established, models that cover "the entire railway spectrum": from the railway net with their lines and stations, with their rail units (linear, switches, crossovers, etc.), with the signalling that changes the states of paths through units and train routes, etc., via trains,

the movement of trains across the net, time--tables, scheduled and rescheduled traffic, including accidents, to passenger and freight services, rolling stock monitoring & control, marshaling, and net development: rail and signal equipment maintenance, new such, etc. The etcetera includes models of railway system rules & regulations, staff and user behaviour, economics, performance, etc. (again!). In other words: a rather sizable "chunk" of the domain. It can be done, and it must be done if we are to have any trust in any future software for even a tiny bit of railway system functioning. Also: there is no way to accomplish the goals, anyway near, other than using formal (combined, of course, with informal, synopsis and narrative) specification!

- **Advice #4:** *Domain/Requirements/Software*

  The projects (thus) should each cover the following research and development spectrum:

  - **Domain Engineering**
  - **Requirements Engineering**
  - **Software Architecture Design**
  - **Program (Structure) Organisation**

    and
  - **Formal Methods–based Tools (CASEs)**

- **Advice #5:** *Problem Frames*

  Software engineering is too broad a term. Professionalism can be achieved primarily for a well-delineated subset of applications. Such subsets could be linked directly to Michael Jackson's concept of problem frames [42, 43].

  We therefore suggest that an international research effort register a number of otherwise distinct consortia, say each working on a particular infrastructure support system, but such that across the consortia a non-trivial number of problem frames are represented:

  - Reactive system frame
  - Information system frame
  - Connection frame
  - Workpiece frame
  - Transaction processing frame
  - Decision support system frame
  - *&c.*

  The aim is to contribute, in a systematic way, to specialised frame-oriented methods: principles, techniques and tools.

- **Advice #6:** *"Pre–competitive"*

  Choose "pre–competitive" projects!

  Now it is easier to see that the above suggested class of joint R&D projects allow for much joint R&D of a nature that does not infringe upon competitive and proprietary aspects of participating commercial enterprises.

6 Summary

A plea has been made for medium-scale, 3–5 partner, 12–20 person international projects covering at least two countries of a region and at least one commercial and competitive software house — all as a means to further sharpen research into techniques in software engineering and into bases for new software systems. The former so as to result in the production of increasingly professional software engineers. The latter so as to result in the competitive survival of local software houses.

A plea has been rather personally and forcefully advanced that such projects be based on the increasingly accepted paradigms of formal methods and that they focus on the full spectrum of software engineering: from domains, via requirements, over software architectures to program structures (program organisation).

A World Congress on Formal Methods

The reader is kindly invited to take part, actively by submitting a paper (or two!), or "passively" by also participating in:

- **World Congress on Formal Methods**

  FM'99: : Toulouse, France, 20–24 September 1999

FM'99 is co-sponsored by ACM, AMAST, EATCS, ETAPS, EU (European Union), FME (Formal Methods Europe), IEEE Computer Society, IFIP, and many other societies.

FM'99 will feature a (i) Technical Symposium, (ii) a Tools Fair & Applications Demo Forum, (iii) a set of Users Group Meetings, and (iv) a set of Industry Tutorials (Formal Methods for Railways, Formal Methods for Telecommunications, Formal Methods for Avionics, Formal Methods for Hardware, etc.).

Program committee chairpersons of the Technical Symposium are Profs. Jeannette Wing, CMU (wing@cs.cmu.edu), and Jim Woodcock, Oxford (jim.woodcock@comlab.ox.ac.uk). The Symposium is planned around up to ten parallel sessions with a total of more than 150 papers and some 10 invited speakers. It will be the premier event in the now mature field of formal methods.

"Surf" to:

`http://www.it.dtu.dk/~db/fm99/cfp.ps`

for regular information on this congress.

B Software Engineering Terminology

B.1 Special Terminology

The wording of many of the definitions of this report may sound dogmatic. Prudent reflection will soon reveal that it is merely a set of reasonable and useful delineations.

1. Software Development:

   To us software development consists of three major components: domain engineering, requirements engineering and software design. Together they form software engineering.

   > `Discussion:` *This is a somewhat "bureaucratic" characterisation. Namely one given in terms of its "way of being handled" — who does it, rather than what it does!*

   Therefore: Software Development aims at constructing software — or as we shall later "enlarge" it: machines. It does it by also constructing models of the domain in which the software will reside, the requirements that the software must satisfy, etc. The present report will deal with the processes of software development.

2. Systems vs. Software Engineering:

   Perhaps the term 'software engineering' is too restrictive. Since any implementation of especially a larger software system entails procurement also of hardware, development will also include configuration and acquisition of hardware components. That larger concept: the development, procurement, installation, performance tuning, operation and disposal of computing systems (hardware $\oplus$ software) is therefore what we mean by systems engineering. Thus software engineering is part of systems engineering.

   > `Discussion:` *As eloquently pointed out by Michael Jackson [42] the term software engineering is probably much too broad a term, or it should be understood as a class term. As such it covers a set of specialised software engineer(ing specialtie)s. Mechanical engineering stands for rather separate groups of for example automotive, heat/water/ventilation, hydrological, nautical, aero-nautical, and many other engineering specialities. Software engineering is still far from having identified suitably specialised such groups — except perhaps for compiler designers. We refer to item 12 (page 20) for hints at what such groups might be.*

3. Linguistic Notions:

   (a) Descriptions & Documents:

   All stages of software development results in descriptions and documents. The two terms are almost synonymous: description refer to the semantic content of the syntactic document. We describe and document domains, requirements, software architectures, program organisations, etc. We sometimes also, again synonymously, refer to these descriptions as

Definitions (as f.ex. for a domain model or a requirements model), sometimes as Specifications (as f.ex. for a software architecture model), yes even as Designs (as f.ex. for a program organisation model).

software engineering management takes the syntactic, document view of development; whereas programming takes the semantic, description view.

(b) Concordant Documents:

A set of documents, spanning the spectrum of descriptions of domains, requirements, software architectures, program organisations, etc., form a set of *concordant* descriptions, and within each of these we may also need alternative, complementary descriptions — which form another set of *concordant* descriptions.

Two or more documents are said to be concordant wrt. each other if they all purport to present descriptions of basically the same thing — but each emphasising different, but related aspects.

We shall later introduce pragmatic notions of perspectives, facets, aspects and views. These represent equivalence classes of concordant documents.

(c) The Informal Languages of Indications, Options and Actions:

As pointed out by Jackson [43] the informal language of domain descriptions is indicative: "what there is", that of requirements descriptions is optative: "what there should be", and that of software design descriptions is imperative: "do this, do that — how to do it!".

(d) Descriptive and Prescriptive Theories:

We could also use the terms descriptive and prescriptive theories in lieu of indicative and optative descriptions.

(e) The Formal Languages of descriptions:

In contrast, the languages of formal descriptions are mathematical, and in mathematics we cannot distinguish between indicative, optative and imperative moods. Such distinctions are meta-linguistic, but necessary. Similarly with the various equivalence classes of concordant documents: perspectives, facets, aspects and views.

(f) Description Techniques:

We refer to Jackson [42, 43]: "Phenomenology — *recognising* and *capturing the significant elementary phenomena* of the subject of interest (domain, requirements, software) *and their relationships. Say as much as is necessary, with perfect clarity, but no more. ... Choose and express abstractions and generalisations* formally *in order* necessarily *to bring an informal reality under intellectual control.*"

Constituent techniques [43] are those of:

- `Designations:`
  That is: system identification. Establishing the informal relationship between real world phenomena and their description identifiers.
- `Definitions:`
  The definition of concepts based on real world phenomena.
- `Refutable Assertions:`
  The usually axiomatic expression of real world properties.

4. Machine:

   The aim of software development is to create software. That software is to function on some hardware. Together we call the executing software ⊕ hardware for the machine. The machine is, in future, to serve in the (future) domain as part of the (future) system.

   Since domain engineering and requirements engineering aim at descriptions that may eventually lead to procurement of both software and hardware we shall refer to software development leading to a machine.

5. Domain Concepts:

   Two approaches seem current in today's 'domain engineering': one which takes its departure point in model–oriented, Mathematical Semantics specification work (and which again basically represents the 'Algorithmic' school), and one which takes its departure point in

knowledge engineering — an outgrowth from AI and Expert Systems. The latter speaks of Ontologies. For now we focus on the former approach.

(a) Domain = System ⊕ Environment ⊕ Stake-holders:

By domain we roughly understand an area of human or other activity. We "divide" the fomain into system, environment and stakeholder. All are part of a perceived world.

> `Discussion:` *Examples of domains are: railways, air traffic, road transport, or shipping of a region; a manufacturing industry with its consumers, suppliers, producers and traders; a ministry of finance's taxation, budget and treasury divisions as manifested through government, state, provincial and city offices and their functions; the financial service industry, or just one enterprise in such an industry (a bank, an insurance company, a securities broker, or a combination of these); etcetera.*
> *Since we are developing software packages that serve in these domains it is important that the software developers are presented with, or themselves help develop precise descriptions (models, see later) of these domains.*
> *Our argument here parallels that given for compiler development: we must first know the syntax and semantics of the (source, target and implementation) languages involved.*

(b) System:

By system we understand a part of the domain. The system is typically an enterprise. Once the machine has been installed in the system then it becomes a part of a new domain wrt. future software development.

> `Discussion:` *A railway System consists of the railway net (lines, stations, signalling, etc.), the rolling stock (locos, passenger waggons, freight cars, etc.) and trains, the time tables and train journey plans, etc. A description of the railway domain must make precise the structure and components of the railway systems as well as all the behaviours it may exhibit.*
> *Identification of the system is an art.*
> *Please note that when we speak of a system we do not refer to a computing system.*

(c) Environment:

By environment we understand that part of the perceived world which interacts with the system. Thus the system complement wrt. "the perceived world", i.e. the environment, together with the system and stakeholder makes up the domain of interest.

> `Discussion:` *The Environment of an air traffic system includes the weather (the meteorology) and the topology of the geographical areas flown over.*
> *Identification of the Environment is an art.*
> *Since the Environment interacts with the System (and hence potentially with the Machine to be built) it is indispensable that we describe (incl. formally model) that part of the Environment which interacts.*

(d) Stakeholder = Clients ⊕ Customers ⊕ Staff:

By stakeholder we mean any of the many kinds of people that have some form of "interest" in the (delivered) machine: enterprise owners, managers, operators and customers of the enterprise: within the system or in the environment.

Discussion: *Stakeholders of a ministry of finance include government ministers, ministry staff and tax payers,*
Identifying all relevant stakeholders is an art.
Since also they interact with the System (and hence potentially with the Machine to be built) it is indispensable that we describe (incl. formally model) possible stakeholder interactions with the System.

(e) Client:

By client we understand the legal entity which procures the machine to be developed. The client is one of the stakeholders, and must be considered a main representative of the system.

Discussion: *A financial enterprise Client is usually the appropriate level executive who specifically contracts some software to serve in the enterprise.*

(f) Staff:

By staff we understand people who are employed in, or by, the system: who works for it, manages, operates and services the system. staff are a major category of stakeholders.

(g) Customer:

By customer we understand the legal entities (people, companies), within the system, who enter into economic contracts with the the client: buys products and/or services from the client, etc. customers form another main category of stakeholders: outside the system, but within the domain.

Discussion: *We have identified important components of a domain. The software engineers — in collaboration with domain stakeholders — face the further tasks of specifically identifying the exact components to be considered for a given Domain.*
That 'identification' is still an art: requires experience and cannot be settled before preliminary modelling experiments have been concluded.

(h) Domain Engineering
= Recognition
↔ Capture
↔ Model
↔ Analysis
↔ Theory:

Domain Engineering, through the processes of domain acquisition and domain modelling, establishes models of the fomain. A domain model is — in principle — void of any reference to the machine, and strives to describe (i.e. explain) the fomain **as it is**. domain analysis investigates the domain model with a view towards establishing a domain theory. The aim of a domain theory is to express laws of the fomain.

Discussion: *The Domain Engineer could be a special version of a Software Engineer — one who could be specially trained both as a Software Engineer, in general, and as a "Domain Expert", in particular.*

(i) Domain Recognition:

System identification is an art! To recognise which are the important phenomena in the domain, and which phenomena are not (important) is not a mechanistic "thing".

(j) Domain Capture
= Acquisition
↔ Modelling:

Discussion: *We make a distinction between the "soft" processes of domain acquisition: linguistic and other interaction with stakeholders, and domain modelling: the "hard" processes of writing down, in both informal and formal notations, the domain model.*

The domain capture process, when actually carried out, often becomes confused with the subsequent requirements capture process. It is often difficult for some stakeholders and for some developers, to make the distinction. It is an aim of this report to advocate that there is a crucial distinction and that much can be gained from keeping the two activities separate. They need not be kept apart in time. They may indeed be pursued concurrently, but their concerns, techniques and documentation need be kept strictly separate.

(k) Ontology:

What we call domain models some researchers call ontology — almost!

In the 'Enterprise Integration and in the 'Information Systems communities ontology means: "formal description of entities and their properties". Ontological analysis is applied to modelling the domain of (manufacturing) enterprises and such systems (typically management systems) whose implementation is typically database oriented.

(l) Domain Model:

By a Domain Model we understand an abstraction of the Domain.

`Discussion:` *Usually we expect a Domain Model, i.e. a Description of the Domain to be presented both informally and formally.*
The informal Description typically consists of a Synopsis which summarises the Model, a Terminology which for every professional term of the Domain defines that term, and a Narrative which — in a readable style — describes how the terms otherwise relate. The formal Model is then expressed in some formal specification language and can be subject to Calculations using a Design Calculi of that notation. The model thus presents the syntax, semantics and, possibly also, the pragmatics of terms of the Domain. Not the syntax and semantics of the professional language spoken by Staff of the Domain System, but just the crucial terms.

(m) Domain Modelling Techniques

Domain modelling usually proceeds by constructing a partial specification (type space, functions and axioms) for each of a number of domain perspectives and similarly one for each domain facet.

(n) Description Technology:

Crucial concepts in domain modelling include:

- system identification,
- i.e. enumeration of designations [43],
- formulation of definitions and
- expression of possibly refutable assertions.

The latter typically in the form of constraints on types and functions.

(o) Domain Perspective:

Domain perspectives reflect the conception of the domain business as seen by various stake-holders.

(p) Domain Facet:

Domain facets reflect some more 'technical, pragmatic decomposition' of the domain together with a 'separation of concerns'. Specification typically proceeds from intrinsic facets, via support technology facets and rules & regulations facets to staff facets, etc.

(q) Domain Model Analysis:

By Domain Analysis we understand informal and formal analyses of the Domain and of the resulting Model — whether informal or formal.

`Discussion:` *The purposes of the analyses can be to ascertain*

whether a component and/or its behaviour qualifies as a component (etc.) of the Domain, and for such included components analyses may reveal Model properties not immediately recognised as properties of the Domain. Note the distinction being made here: the Domain as it exists "out there", and the Model as an abstraction thereof and which "exists" on the (electronic) "paper" upon which the Model is represented.

(r) Domain Theory:

The purpose of Domain Analysis is to also establish a Theory of the Domain, or rather: of the Models purported to represent the Domain!

`Discussion:` *Examples of theorems in a theory of railways could be: (1) (Kirschhoffs law for trains:) "Over a suitably chosen time interval (say 24 hours) the number of trains arriving at any station, minus the number of trains taken out of service at that station, plus the number of trains put into service at that station, equals the number of trains leaving that station"; (2) (God doesn't play dice:) "Two trains moving down a line cannot suddenly change place"; (3) (No Ghost Trains) "If at two times 'close to each other' (say seconds apart) a train has been observed on the railway net, then that train is on the railway net somewhere between the two original observation positions at any time between the two original observation times". Etc.*
Failure to record essential theorems may result in disastrously erroneous software.
Ability to identify and establish appropriate theorems is an art and takes years!

(s) Domain Model Validation

An informal process whereby informal and formal specification parts are related and where these again are related to the "real world" domain (system identification)

6. Requirements Concepts:

Requirements, as we have seen, form a bridge between the larger Domain and the "narrower" software which is to serve in the Domain.

(a) Requirements = System $\oplus$ Interface $\oplus$ Machine:

Requirements issues are either such which concern (i) machine support of the system, (ii) human (and other) interfaces between the system and the machine, or (iii) the machine itself.

Requirements describes the system as the stakeholders **would like to see it**.

(b) Functional & Non-Functional Requirements:

Functional requirements include the concepts and facilities to be offered by the desired software.Non-functional requirements emphasise such less tangible issues as performance, user dialogue interface, dependability, etc.

(c) Requirements Engineering
= Capture
$\leftrightarrow$ Model
$\leftrightarrow$ Analysis
$\leftrightarrow$ Theory:

Requirements Engineering, through the process of requirements capture, establishes models of the requirements. The "conversion" from requirements information obtained through requirements elicitation, via requirements modelling to requirements models is called requirements capture. Requirements Models are formally derived from and extends domain models. Requirements Engineering also

analyses requirements models, in order to derive further properties of the requirements.

> `Discussion:` *We hope the reader observes the "similarity" in the components of domain engineering vs. those of requirements engineering.*

(d) Requirements Capture
= Elicitation
↔ Modelling:

Remarks similar to those under Domain Capture — item 5j (page 15) apply.

(e) Requirements Model:

A specification of the requirements. Usually in the form of a set of partial specifications, one for each requirements aspect.

(f) Requirements Modelling Techniques:

Requirements "reside in the domain", and are hence primarily projections of their type space and functions. Functional techniques deal with projections, resolving domain/requirements dichotomies and extending domains. Non-functional techniques deal with machine notions: computing platform, system dependability and maintainability, and with computer human interface issues: user-friendliness, graphic user interfaces, dialogue management, etc.

(g) Requirements Model Analysis:

By Requirements Analysis we understand informal and formal analyses of the Requirements and of the resulting Model — whether informal or formal.

> `Discussion:` *The purposes of the analyses can be to ascertain whether a component and/or its behaviour qualifies as a component (etc.) of the Requirements, and for such included components analyses may reveal Model properties not immediately recognised as properties of the Requirements. Note the distinction being made here: the Requirements as it exists "out there" — among Stake-holders, and the Model as an abstraction thereof and which "exists" on the (electronic) "paper" upon which the Model is represented.*

(h) Requirements Theory:

The purpose of Requirements Analysis is to also establish a Theory of the Domain, or rather: of the Models purported to represent the Domain!

7. Software Concepts:

(a) Software Design
= Software Architecture Specification
↔ Program Organisation Specification
↔ Refinements
↔ Coding:

Software Design, through the process of design ingenuity, proceeds from establishing a software architecture, to deriving a program organisation, and from that, in further steps of design reification, also called design refinement, constructing the "executable code".

(b) Software Architecture:

A software architecture description specifies the concepts and facilities offered the user of the software — i.e. the external interfaces.

Usually functional requirements "translate" into software architecture properties.

(c) Program Organisation:

A program organisation description specifies internal interfaces between program modules (processes, platform components, etc.).

Usually non-functional requirements "translate" into program organisation design decisions.

(d) Refinement:

Design Refinement covers the derivation from the requirements model of the software architecture, of the program organisation from the software architecture,

and of further steps of concretisations into program code.

8. Creation — Acquisition, Elicitation and Invention:

   All stages and steps of the software development process involves creation: domain acquisition & domain modelling, requirements elicitation & requirements modelling, and design ingenuity. This human process of invention leads to the construction of informal as well as formal descriptions.

9. Systematic, Rigorous and Formal Development:

   The software development may be characterised as proceeding in either a systematic, a rigorous or even, in parts, a formal manner — all depending on the extent to which the underlying formal notation is exploited in reasoning about properties of the evolving descriptions.

   (a) *Formal Notation:*
   By a formal notation we understand a language with a precise syntax, a precise semantics (meaning), and a proof system. By "a precise ..." we usually mean "a mathematical ...".

   (b) *Systematic Use of Formal Notation:*
   By a systematic use of formal notation we understand a use of the notation in which we follow the precise syntax and the precise semantics.

   (c) *Rigorous Use of Formal Notation:*
   By a rigorous use of formal notation we understand a systematic use in which we additionally exploit some of the 'formality' by expressing theorems of properties of what has been written down in the notation.

   (d) *Formal Use of Formal Notation:*
   By a formal use of formal notation we understand a rigorous use in which we fully exploit the 'formality' by actually proving properties.

   (e) *Formal Method* $\approx$ *Formal Specification* $\otimes$ *Calculation:*
   We refer to item 3 (page 21) for a definition of 'method'.
   The methods claimed today to be formal methods may be formal, but are not methods in the sense we define that term! Since we do not believe that a method for developing software: from domains via requirements, can be formal, but only that use of the notations deployed may be, we (now) prefer the terms: formal specification and calculation.

   (f) *Design Calculi — or Formal Systems:*
   By a design calculus we understand a formal system consisting of a formal notation and a set of precise rules for converting expressions of the formal notation into other such, semantically 'equivalent' expressions.

10. Satisfaction = Validation $\oplus$ Verification:

    The domain acquisition and requirements elicitation processes alternate with domain modelling and requirements modelling, respectively, and these again with securing satisfaction.

    (a) Validation:
    In this report we are not interested in the crucial process of interactions between software developers (i.e. software engineers, which we see as domain engineers, requirements engineers and software designers) and the stakeholders. validation is thus the act of securing, through discussion, etc., with the stakeholders that the domain model correctly reflects their understanding of the domain.

    (b) Verification:
    Let $\mathcal{D}$, $\mathcal{R}$ and $\mathcal{S}$ stand for the theories of the fomain, requirements and software. Then verification:

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

    shall mean that we can verify that the designed software satisfies the requirements in the presence of knowledge (i.e. a theory) about the fomain.

11. Software Engineering:

    Software Engineering is the combination of domain engineering, requirements engineering and software design, and is seen as the process of going between science and technology. That is, of developing descriptions on the basis of scientific results using mathematics — as in other engineering branches — and of understanding (the constructed fomain of) existing (software) technologies by subjecting them to rigorous domain analysis.

12. Frame Specialisation:

    > `Discussion:` *In item 2 (page 12) we discussed the problem of software engineering being seemingly as a too wide field. And we hinted that specialisation might be a natural way of achieving a level of professionalism achieved in traditional engineering fields. In this item we will briefly introduce the concept of problem frames and give example of distinct such frames.*

    A problem frame is well-delineated part of all the problems to which computing might be applied — such that this frame offers a precise set of principles, techniques and tools for software development, and such that this 'method' fits the frame "hand in glove".

    - *Principal Parts and Solution Task*

    Following Jackson [42, 43] we think of a (problem) frame as consisting of its principal parts and a solution task. The principal parts are (1.) the domain — which exists a-priori — and (2.) the requirements. The solution task is that of developing the software — something relatively new! Tackling an application problem consists initially of analysing it **into** a frame, including a multi-frame with clearly identified part-frames.

    We explain a few frames and otherwise refer to [42, 43, 11]:

    (a) *Translation Frame:*

    The principal parts are: (i) two formalised languages (syntax and semantics), source and target; (ii) the concrete form of the syntactic representations of either: the source usually in the form of a BNF grammar for textual input, the target usually in the form of an internal ("electronic") data structure; (iii) user requests for compilation from source to target; (iv) the compiler; and (v) the translation function. (i-ii) form the domain, (iii-iv-v) the requirements.

    The solution task now involves developing the compiler using a well-defined set of techniques and tools: lexical scanner generators, possibly error-correcting parser generators, attribute grammar interpreters, etc.

    (b) *Reactive Systems Frame:*

    The principal parts are (i) the dynamic (temporal, real-time) "real world"; (ii) its observable variables [output], (iii) its controllable variables [input]; (iv) user (or other system) requests for the monitoring and/or control of the "real world"; (v) the monitoring & control (software etc.) system; and (vi) the specific monitoring & control functions (optimisation, safety, dependability, etc.). Items (i-ii-iii) form the domain, (iv–vi) the requirements.

    The solution task now involves control theoretic and real-time, safety critical software design principles, techniques and tools.

    It seems that Jackson refers to the reactive systems frame as the environment-effect frame [42].

    (c) *Information Systems Frame:*

    The principal parts are almost as for reactive systems (i-ii) except that there is no desire for control, and the issues of safety criticality, real-time and dependability are replaced by (vi) (observable) information security and the need for usually "massive" information storage (for statistical and other purposes); (iv) the requests are concerned with the visualisation of observed information and

computations over these; (v) the system is thus more of an information (monitoring) system; and (vi) the functions include specifics about the visualisation and other processing.

The solution task can perhaps best be characterised in terms of the principles, techniques and tools for example offered by Jackson's JSD method [41, 42].

(d) *Connection Frame:* See [42, 43, 11] for details.

(e) *Workpiece Frame:* See [42, 43, 11] for details.

(f) *Transaction Frame:* See [42, 43, 11] for details.

(g) *Multi–frame:* See [42, 43, 11] for details.

Usually a problem is not reducible to a single of the frames mentioned above (and some of these, due to requirements, often "overlap"). In such cases we have a multi-frame, a frame being best characterised in terms of hopefully reasonable well-delineated (sub-) frames.

(h) *&c.*

B.2 General Terminology

Many more terms are used in the subject field of this report: in its science and in its engineering. Sometimes with unclear meanings, and not always with the same meaning from paper to paper. We shall therefore try delineate also important general concepts.

Some dogmas:

1. Computer Science:

   Computer Science, to us, is the study and knowledge of the foundations of the artifacts that might exist inside computers: the kinds of information, functions and processes (i.e. type theory), models of computability and concurrency; bases for denotational, algebraic and operational semantics; specification and programming language proof theories; automata theory; theory of formal languages; complexity theory; etc.

2. Computing Science:

   Computing Science, to us, is the study and knowledge of how to construct the artifacts that are to exist inside computers. Successful computing science results in a useful programming methodology.

   The present report "falls", subject-wise, somewhere between computing science and software engineering.

3. Method:

   By a method we understand a set of *principles* of *analysis*, and for *selecting* and *applying techniques* and *tools* in order *efficiently* to *construct efficient artifacts* — here software.

4. Methodology:

   By methodology we understand the study and knowledge about methods. Since we can assume that no one software development method will suffice for any entire construction process we need be concerned with methodology.

5. Software:

   By software we understand all the documentation that is necessary to *install, operate, run, maintain* and *understand* the executable code, as well as that *code* itself and the *tools* that are needed in any of the above (i.e. including the original development tools).

6. Software Technology:

   By software technology we understand sets of software tied to sets of specific platforms. (By a platform we mean "another" machine!)

7. Programming:

   Programming is a subset of activities within software engineering which focus on the systematic, via rigorous to formal creation of descriptions using various design calculi.

8. Engineering:

   Engineering is the act of constructing technology based on scientifically established results and of understanding existing technologies scientifically.

9. Engineer:

   Engineers perform engineering and use, as a tool, mathematics. It is used in order to model, analyse, predict, construct, etc. software engineers reason about the artifacts they construct, be they (fomain, requirements, software architecture, program organisation, etc.) model descriptions (i.e. definitions or specifications) or program code.

10. Technician:

    Technicians use technologies: they compose, use and "destroy" them — without necessarily using mathematics.

11. Technologist:

    Technologists are technicians who manage technologies: perceive, demand, produce, procure, market and deploy technologies.

This report views software engineering as hinted above: As the act of going between science and technology, using mathematics — wherever useful.

REFERENCES

[1] Jean-Raymond Abrial. *The B Book: Assigning Programs to meanings*. Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, England, 1996.

[2] Cleta Milagros Acebedo. An Informal Domain Analysis for Manufacturing Enterprises. Research Report 62, UNU/IIST, P.O.Box 3058, Macau, March 1996.

[3] Cleta Milagros Acebedo and Erwin Paguio. Manufacturing Enterprise Simulation: A Business Game. Research Report 64, UNU/IIST, P.O.Box 3058, Macau, March 1996.

[4] Dao Nam Anh and Richard Moore. Formal Modelling of Large Domains — with an Application to Airline Business. Technical Report 74, UNU/IIST, P.O.Box 3058, Macau, June 1996. Revised: September 1996.

[5] D. Bjørner. A Software Engineering Paradigm: From Domains via Requirements to Software. Research report, Dept. of Information Technology, Technical University of Denmark, Bldg.345/167–169, DK–2800 Lyngby, Denmark, July 1997.

[6] D. Bjørner. Towards a Domain Theory of The Financial Sevice Industry. Research report, Dept. of Information Technology, Technical University of Denmark, Bldg.345/167–169, DK–2800 Lyngby, Denmark, July 1997.

[7] D. Bjørner and C.B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[8] Dines Bjørner. Formal Models of Robots: Geometry & Kinematics. Research Report 6, UNU/IIST, P.O.Box 3058, Macau, March 15 1993. For abbreviated version see Chapter 3 in *A Classical Mind*, Festschrift for C.A.R. Hoare, Prentice-Hall International, Publ., 1994, pp 37–58.

[9] Dines Bjørner. Prospects for a Viable Software Industry — Enterprise Models, Design Calculi, and Reusable Modules. Technical Report 12, UNU/IIST, P.O.Box 3058, Macau, 7 November 1993. Appendix — on a railway domain model — by Søren Prehn and Dong Yulin, Published in *Proceedings from first ACM Japan Chapter Conference*, March 7–9, 1994: World Scientific Publ., Singapore, 1994.

[10] Dines Bjørner. Models of Enterprise Management: Strategy, Tactics & Operations — Case Study Applied to Airlines and Manufacturing. Technical Report 60, UNU/IIST, P.O.Box 3058, Macau, January – April 1996.

[11] Dines Bjørner. Michael Jackson's Problem Frames: Domains, Requirements and Design. In Li ShaoYang and Michael Hinchley, editors, *ICFEM'97: International Conference on Formal Engineering Methods*, Los Alamitos, November 12–14 1997. IEEE Computer Society.

[12] Dines Bjørner. Models of Financial Services & Industries. Research Report 96, UNU/IIST, P.O.Box 3058, Macau, January 1997. Incomplete Draft Report.

[13] Dines Bjørner, Chris W. George, Bo Stig Hansen, Hans Laustrup, and Søren Prehn. A Railway System, Coordination'97, Case Study Workshop Example. Research Report 93, UNU/IIST, P.O.Box 3058, Macau, January 1997.

[14] Dines Bjørner, Dong Yu Lin, and Søren Prehn. Domain Analyses: A Case Study of Station Management. Research Report 23, UNU/IIST, P.O.Box 3058, Macau, 9 November 1994.

[15] J.P. Bowen and M. Hinchey. Seven More Myths of Formal Methods. Technical Report PRG–TR–7–94, Oxford Univ., Programming Research Group, Wolfson Bldg., Parks Road, Oxford OX1 3QD, UK, June 1994. Shorter version published in LNCS Springer Verlag FME'94 Symposium Proceedings.

[16] J.P. Bowen and M. Hinchey. Ten Commandments of Formal Methods. Technical report, Oxford Univ., Programming Research Group, Wolfson Bldg., Parks Road, Oxford OX1 3QD, UK, 1995.

[17] M. Cerioli, A. Haxthausen, B. Krieg-Brückner, and T. Mossakowski. Permissive Sub-sorted Partial Logic in CASL. In *Proceedings, AMAST'97*, volume 1349 of *Lecture Notes in Computer Science*. Springer–Verlag, 1997.

[18] Zhou Chaochen. Duration Calculi: An Overview. Research Report 10, UNU/IIST, P.O.Box 3058, Macau, June 1993. Published in: *Formal Methods in Programming and Their Applications,* Conference Proceedings, June 28 – July 2, 1993, Novosibirsk, Russia; (Eds.: D. Bjørner, M. Broy and I. Pottosin) LNCS 736, Springer-Verlag, 1993, pp 36–59.

[19] Zhou Chaochen and Michael R. Hansen. Lecture Notes on Logical Foundations for the Duration Calculus. Lecture Notes, 13, UNU/IIST, P.O.Box 3058, Macau, August 1993.

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

[21] Zhou Chaochen and Yu Huiqun. A duration Model for Railway scheduling. Technical Report 24b, UNU/IIST, P.O.Box 3058, Macau, May 1994.

[22] Zhou Chaochen, Dang Van Hung, and Li Xiaoshan. A Duration Calculus with Infinite Intervals. Research Report 40, UNU/IIST, P.O.Box 3058, Macau, February 1995. Published in: *Fundamentals of Computation Theory*, Horst Reichel (ed.), pp 16-41, LNCS 965, Springer-Verlag, 1995.

[23] Zhou Chaochen, Anders P. Ravn, and Michael R. Hansen. An Extended Duration Calculus for Real-time Systems. Research Report 9, UNU/IIST, P.O.Box 3058, Macau, January 1993. Published in: *Hybrid Systems*, LNCS 736, 1993.

[24] Zhou Chaochen and Li Xiaoshan. A Mean Value Duration Calculus. Research Report 5, UNU/IIST, P.O.Box 3058, Macau, March 1993. Published as Chapter 25 in *A Classical Mind*, Festschrift for C.A.R. Hoare, Prentice-Hall International, 1994, pp 432–451.

[25] Razvan Diaconescu and Kokichi Futatsugi. Logical Semantics of CafeOBJ. Research Report IS-RR-96-0024S, JAIST, 1996.

[26] Do Tien Dung, Le Linh Chi, Nguyen Le Thu, Phung Phuong Nam, Tran Mai Lien, and Chris George. Developing a Financial Information System. Technical Report 81, UNU/IIST, P.O.Box 3058, Macau, September 1996.

[27] Myatav Erdenechimeg, Richard Moore, and Yumbayar Namsrai. MultiScript I: The Basic Model of Multi-lingual Documents. Technical Report 105, UNU/IIST, P.O.Box 3058, Macau, June 1997.

[28] Yu Xinyiao et al. Stability of Railway Systems. Technical Report 28, UNU/IIST, P.O.Box 3058, Macau, May 1994.

[29] John Fitzgerald and Peter Gorm Larsen. *Modelling Systems: Practical Tools and Techniques*. Cambridge University Press, 1997–1998.

[30] Yang Lu Fu Hongguang and Zhou Chaochen. A Computer-Aided Geometric Approach to Inverse Kinematics. Research Report 101, UNU/IIST, P.O.Box 3058, Macau, April 1997.

[31] Kokichi Futatsugi and Razvan Diaconescu. *CafeOBJ Report — Definition of the Language*. To appear in 1998 as book in the AMAST series at World Scientific

[32] Jan Goossenaerts and Dines Bjørner. An Information Technology Framework for Lean/Agile Supply-based Industries in Developing Countries. Technical Report 30, UNU/IIST, P.O.Box 3058, Macau, 1994. Published in *Proceedings of the International Dedicated Conference on Lean/Agile Manufacturing in the Automotive Industries*, ISATA, London, UK.

[33] Jan Goossenaerts and Dines Bjørner. Interflow Systems for Manufacturing: Concepts and a Construction. Technical Report 31, UNU/IIST, P.O.Box 3058, Macau, 1994. Published in *Proceedings of the European Workshop on Integrated Manufacturing Systems Engineering*.

[34] The RAISE Language Group. *The RAISE Specification Language*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1995.

[35] The RAISE Method Group. *The RAISE Method*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1992.

[36] J. Guttag, J.J. Horning, and J.M. Wing. Larch in Five Easy Pieces. Technical Report 5, DEC SRC, Dig. Equipm. Corp. Syst. Res. Ctr., Palo Alto, California, USA, 1985.

[37] John V. Guttag, James J. Horning, S.J. Garland, K.D. Jones, A. Modet, , and J.M. Wing. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, Springer-Verlag New York, Inc., Attn: J. Jeng, 175 Fifth Avenue, New York, NY 10010-7858, USA, 1993.

[38] Anthony Hall. Seven Myths of Formal Methods. *IEEE Software*, 7(5):11–19, 1990.

[39] Dang Van Hung and Zhou Chaochen. Probabilistic Duration Calculus for Continuous Time. Research Report 25, UNU/IIST, P.O.Box 3058, Macau, May 1994. Presented at *NSL'94 (Workshop on Non-standard Logics and Logical Aspects of Computer Science, Kanazawa, Japan, December 5–8, 1994)*, submitted to *Formal Aspects of Computing*.

[40] Dang Van Hung and Phan Hong Giang. A Sampling Semantics of Duration Calculus. Research Report 50, UNU/IIST, P.O.Box 3058, Macau, November 1995. Published in: *Formal Techniques for Real-Time and Fault Tolerant Systems*, Bengt Jonsson and Joachim Parrow (Eds), LNCS 1135, Spriger-Verlag, pp. 188–207, 1996.

[41] M. Jackson. *System Design*. Prentice-Hall International, 1985.

[42] Michael Jackson. Problems, methods and specialisation. *Software Engineering Journal*, pages 249–255, November 1994.

[43] Michael Jackson. *Software Requirements & Specifications: a lexicon of practice, principles and prejudices*. ACM Press. Addison-Wesley Publishing Company, Wokingham, nr. Reading, England; E-mail: ipc@awpub.addwes.co.uk, 1995. ISBN 0-201-87712-0; xiv + 228 pages.

[44] T. Janowski and C.M. Acebedo. Virtual Enterprise: On Refinement Towards an ODP Architecture. Research Report 69, UNU/IIST, P.O.Box 3058, Macau, May 1996.

[45] Tomasz Janowski. Domain Analysis for Manufacturing: Formalization of the Market. Research Report 63, UNU/IIST, P.O.Box 3058, Macau, March 1996.

[46] Tomasz Janowski and Rumel V. Atienza. A Formal Model For Competing Enterprises, Applied to Marketing Decision-Making. Research Report 92, UNU/IIST, P.O.Box 3058, Macau, January 1997.

[47] C.B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, second edition, 1990.

[48] Souleymane Koussoubé. Knowledge-Based Systems: Formalisation and Applications to Insurance. Research Report 108, UNU/IIST, P.O.Box 3058, Macau, May 1997.

[49] K. Lano. *The B Language and Method, A Guide to Pratical Formal Development*. Springer-Verlag, Formal Approaches to Computing and Information Technology (FACIT). Ed.: S.A. Schuman, 1996.

[50] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive Systems: Specifications*. Addison Wesley, 1991.

[51] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive Systems: Safety*. Addison Wesley, 1995.

[52] P.D. Mosses. COFI: The Common Framework Initiative for Algebraic Specification and Development. In M. Bidoit and M. Dauchet, editors, *TAPSOFT'97*, volume 1212 of *Lecture Notes in Computer Science*. Springer–Verlag, 1997.

[53] Ataru T. Nakagawa, Toshimi Sawada, and Kokichi Futatsugi. *CafeOBJ manual (for system version 1.3)*. 142 pages.

[54] Kokichi Futatsugiand Ataru Nakagawa. An Overview of CAFE Specification Environment – An Algebraic Approach for Creating, Verifying, and Maintaining Formal Specifications over Networks. In *ICFEM'97: International Conference on Formal Engineering Methods*. IEEE Computer Society, IEEE CS Press, November 1997.

[55] Nikolaj Nikitchenko. Towards Foundations of the General Theory of Transport Domains. Research Report 88, UNU/IIST, P.O.Box 3058, Macau, December 1996.

[56] Roger Noussi. An Efficient Construction of a Domain Theory for Resources Management: A Case Study. Research Report 107, UNU/IIST, P.O.Box 3058, Macau, May 1997.

[57] CoFI Task Group on Language Design. CASL — The Common Algebraic Specification Language Summary. Available at http://www.brics.dk/Projects/CoFI/-Documents/CASL/Summary/, 1997.

[58] CoFI Task Group on Semantics. CASL — The CoFI Algebraic Specification Language (version 0.97) Semantics. Available at http://www.brics.dk/Projects/CoFI/Notes/S-4/, 1997.

[59] J. Michael Spivey. *Understanding Z: A Specification Language and its Formal Semantics*, volume 3 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, UK, January 1988.

[60] J. Michael Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire HP2 4RG, UK, 1989.

[61] John Wordsworth. *Software Engineering with B*. Addison-Wesley Longman, 1996.

[62] Tan Xinming. Enquiring about Bus Transport–Formal Development using RAISE. Technical Report 83, UNU/IIST, P.O.Box 3058, Macau, September 1996.