The Triptych Process Model [1]
Process Assessment and Improvement

Dines Bjørner

Computer Science and Engineering
Informatics and Mathematical Modelling
Technical University of Denmark
DK-28000 Kgs.Lyngby
Denmark

Graduate School of Information Science
Japan Adv. Inst. of Science & Technology
1-1, Asahidai, Tatsunokuchi
Nomi, Ishikawa 923-1292
Japan

bjorner@gmail.com

June 14, 2006. Compiled November 17, 2006

Dedicated to Kouichi Kishida

Abstract

The triptych[2] approach to software engineering proceeds on the basis of carefully monitored and controlled possibly iterated progression through domain engineering and requirements engineering to software design.

In this paper we will outline these three phases, show the many stages of development within each and also indicate the many steps within each stage. We will ever so briefly touch upon informal narration and formal description (prescription and specification) of domains (requirements and software designs), and the verification (theorem proving, model checking and testing) and validation of domain descriptions (requirements prescriptions and their relations to domain descriptions, as well as the software design specifications and their relations to requirements prescriptions). The importance of process management and its relations to software process assessment (SPA) and software process improvement (SPI) will then be underscored. Our measuring "stick" is that set up by Watts Humphrey's capability maturity model (CMM). We will suggest and discuss seven assessment and eight improvement categories. In closing we will have some few words to say about software procurement.

Contents

[1] Seminar at Tokyo University in Honour of Kouichi Kishida's 70'th Anniversary

[2] The term 'triptych' covers the three phases of software development: domain description, requirements prescription and software design.

1 The Triptych Dogma

1.1 Background

In the past, as exemplified in major software engineering textbooks [1, 2, 3, 4, 5, 6], software engineering focused on requirements engineering and software design. The triptych dogma extends the two (requirements engineering and software design) into three (domain engineering plus the two phases already mentioned).

1.2 The Dogma

- Justifying requirements prescriptions:
  - Before software can be designed
  - we must understand the requirements.
- Justifying domain descriptions.
  - Before requirements can be prescribed
  - we must understand the domain.
- Justifying the triptych:
  - First analysing and describing the (application) domain,
  - then analysing and prescribing the requirements, and
  - finally analysing and specifying the software design and code.

1.3 New Aspects

The relatively new aspect of software development is here 'domain engineering'. This new aspect "translates" into a number of new methodological aspects of domain and requirements engineering. The next, the major section will survey these aspects. All of this is covered extensively in volume 3 of the three volume book [7, 8, 9]. All 11 figures in this paper are re-used from [9](by permission from the book publisher).

2 The Triptych Process Models and Documents

2.1 Common Aspects

2.1.1 Process Models

The triptych process model is the composition of three process models: one each for domain engineering, requirements engineering and software design. We hint at this composition in Fig. 1 on the following page.

The internals of the three boxes (i.e., phases of development) of Fig. 1 on the next page are outlined in Figs. 4 on page 7, 8 on page 11 and 9 on page 12, respectively Fig. 11 on page 14.

The DO edges of Fig. 1 on the following page enter topmost boxes of respective Figs. 4 on page 7, 8 on page 11 and 11 on page 14.

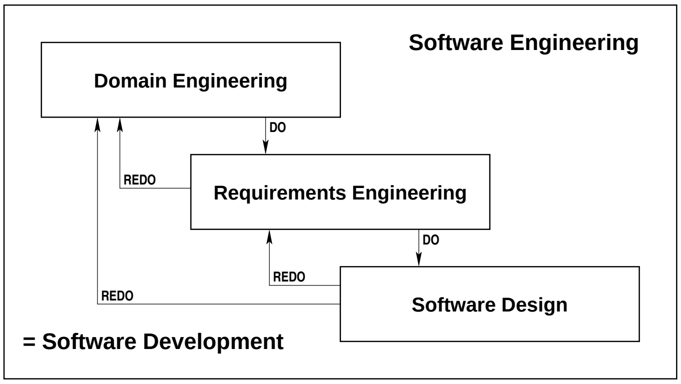

Figure 1: A simplified view of the triptych process model

The REDO edges of Fig. 1 enter whichever boxes of Figs. 4 on page 7, 8 on page 11 and 9 on page 12, respectively Fig. 11 on page 14 that are found to be most appropriate. (More on this later.)

2.1.2 Documents

Common to all three phases of software development are that they primarily manifest themselves in documents. Figure 3 on page 6, Figs. 5 on page 9, 6 on page 10, and 7 on page 11, and Fig. 10 on page 13, to be commented later, illustrate the breadth, depth and quite substantial number of such resulting documents. And common to each set of such documents is the more-or-less administrative "working out" of *information document*, cf. items 1 of Figs. 3 on page 6, 5 on page 9, 6 on page 10, 7 on page 11, and 10 on page 13. Figure 2 extracts item 1. from Figs. 3 on page 6, 5 on page 9, 6 on page 10, and 7 on page 11, and 10 on page 13.

1. Information
   (a) Name, Place and Date
   (b) Partners
   (c) Current Situation
   (d) Needs and Ideas
   (e) Concepts and Facilities
   (f) Scope and Span
   (g) Assumptions and Dependencies
   (h) Implicit/Derivative Goals
   (i) Synopsis
   (j) Standards Compliance
   (k) Contracts
   (l) The Teams
      i. Management
      ii. Developers
      iii. Client Staff
      iv. Consultants
   (m) Plans
      i. Project Graph
      ii. Budget
      iii. Funding
      iv. Accounts
   (n) Management
      i. Assessement
      ii. Improvement
         A. Plans
         B. Actions

Figure 2: Informative documents

Let us briefly review the import of Fig. 2. In any of the three phases of development, domain engineering, requirements engineering and software design, the information implied by the table-of-contents of Fig. 2 on the facing page must be carefully worked out. Take items 'Assumptions and Dependencies', and 'Implicit/Derivative Goals'. The description, prescription or design work to be done in the phase to which the information documents apply rely on assumptions and dependencies. These must be fully understood, hence documented before any proper development takes place. Consider items 'Current Situation', 'Needs and Ideas', and 'Concepts and Facilities'. The current situation which apparently warrants the proper development must be recorded. It might change thus necessitating change of development. Development — of whichever of the three phases — would not be undertaken unless someone, the customer and/or the developer, has some needs for the (approximately) expected results of the development, and, as well, has some ideas as how (methodologically) to basically develop whatever is to be developed (a domain description, a requirements description, a software design). The customer and/or developer also, initially have made some thoughts of the core concepts and facilities around which the development is expected to take place. All of this need be properly recorded as any review of project status occurs in the pragmatic context of 'Assumptions and Dependencies', 'Implicit/Derivative Goals', 'Current Situation', 'Needs and Ideas', and 'Concepts and Facilities'.

2.2 The Domain Engineering Process Model

We first rough-sketch narrate the stages and steps of the domain engineering development of a domain model, then review the documents that should emanate from such development. Finally we diagram an essence of the narration and the document table-of-contents.

But first some words on domain models.

2.2.1 Domain Models

A main result of domain engineering development, as applied to some specific application domain[3], is a domain model. Domain models are in the form of descriptions. Domain descriptions describe what there is, and as it is. There is no presumption of requirements implied by these descriptions. They are not requirements prescriptions. By analogy, physicists [domain engineers] are describing mother nature [application domains] and engineers [requirements engineers and software designers] are prescribing and implementing requirements.

2.2.2 Domain Engineering, A Narrative

The domain engineering triptych dogma, and as argued in Chaps. 8–17 of [9], advocates (item 2.) the following stages of description development (after work on information documents [items 1.a–l] have been duly completed): (2.a) identification of as wide a spectrum of domain stakeholders, (2.b) acquisition of domain understanding, (2.c) establishment (and subsequent, throughout all stages, use and maintenance) of a domain terminology (ontological terms), (2.d) understanding and rough-sketching all relevant business processes, (2.e) domain modelling (all domain facets), and (2.f) the domain model completion (including consolidation). Intertwined with the domain description parts (item 2., subitems (a–f)) are the analysis parts with (3.a) analysis aiming at identifying inconsistencies, conflicts and

[3] Examples of domains are: (1) the financial service industry as a whole, (1.1) a bank, (1.1.1) a bank's mortgage lending business; (2) the transportation industry as a whole, (2.1) a railway system, (2.1.1) an interlocking system; etcetera.

incompletenesses, (3.b) domain validation, (3.c) domain verification,and (3.d) possible work on establishing a domain theory.

The new thing here is all of items 1.–2.–3.

2.2.3 Domain Engineering Documents

1. Information
   (a) Name, Place and Date
   (b) Partners
   (c) Current Situation
   (d) Needs and Ideas
   (e) Concepts and Facilities
   (f) Scope and Span
   (g) Assumptions and Dependencies
   (h) Implicit/Derivative Goals
   (i) Synopsis
   (j) Standards Compliance
   (k) Contracts
   (l) The Teams
      i. Management
      ii. Developers
      iii. Client Staff
      iv. Consultants
   (m) Plans
      i. Project Graph
      ii. Budget
      iii. Funding
      iv. Accounts
   (n) Management
      i. Assessement
      ii. Improvement
         A. Plans
         B. Actions
2. Descriptions
   (a) Stakeholders
   (b) The Acquisition Process
      i. Studies
      ii. Interviews
      iii. Questionnaires
      iv. Indexed Description Units
   (c) Terminology
   (d) Business Processes
   (e) Facets:
      i. Intrinsics
      ii. Support Technologies
      iii. Management and Organisation
      iv. Rules and Regulations
      v. Scripts
      vi. Human Behaviour
   (f) Consolidated Description
3. Analyses
   (a) Domain Analysis and Concept Formation
      i. Inconsistencies
      ii. Conflicts
      iii. Incompletenesses
      iv. Resolutions
   (b) Domain Validation
      i. Stakeholder Walkthroughs
      ii. Resolutions
   (c) Domain Verification
      i. Model Checkings
      ii. Theorems and Proofs
      iii. Test Cases and Tests
   (d) (Towards a) Domain Theory

Figure 3: Domain engineering document table-of-contents

Figure 3 summarises the plenitude of highly interrelated sets of documents that must all be carefully worked out and carefully correlated.

2.2.4 Domain Engineering Stages and Steps

Figure 4 on the facing page diagrams, in box-and-edge form, the stages and steps of domain engineering development and their interrelations. The diagram does not give a correct "pic-

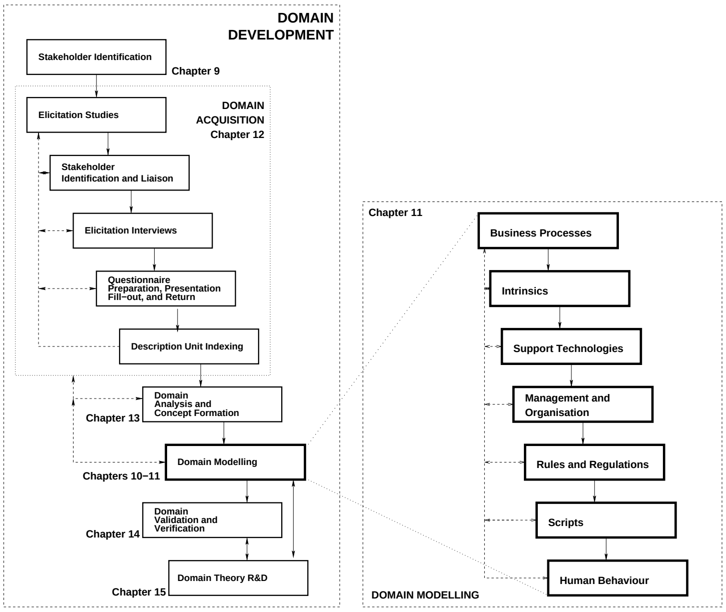

Figure 4: The domain engineering process model diagram

ture" of the necessity for iteration: going "backwards and forwards" across the development, i.e., across the diagram. Obviously, having a precise understanding of the syntax, semantics and pragmatics of boxes and edges, helps developers and their managers monitor and control (including "contain") iterations.

2.3 The Requirements Engineering Process Model

We first rough-sketch narrate the stages and steps of the requirements engineering development of a requirements model, then review the documents that should emanate from such development. Finally we diagram an essence of the narration and the document table-of-contents.

But first some words on "the machine" and on requirements models.

2.3.1 The Machine

Requirements is about prescribing the machine: the hardware and the software which shall implement the requirements. The machine resides in the domain. Once developed we shall sometimes refer to that domain as the environment of the machine — with the machine + that environment becoming a new domain.

2.3.2 Requirements Models

A main result of requirements engineering development, as applied to some specific application domain[4], is a requirements model. Domain models are in the form of descriptions.

[4]Examples of domains are: (1) the financial service industry as a whole, (1.1) a bank, (1.1.1) a bank's mortgage lending business; (2) the transportation industry as a whole, (2.1) a railway system, (2.1.1) an interlocking system; etcetera.

Requirements prescriptions prescribe what there should be.

2.3.3 Requirements Engineering, A Narrative

The requirements engineering triptych dogma, and as argued in Chaps. 18–26 of [9], advocates (item 2.) the following stages of prescription development (after work on information documents [items 1.a–l] have been duly completed): (2.a) identification of as wide a spectrum of requirements stakeholders, (2.b) acquisition of requirements statements, (2.c) rough-sketching first ideas of a requirements model in order to ("eureka") discover un-formulated requirements, (2.d) establishment (and subsequent, throughout all stages, use and maintenance) of a requirements terminology (ontological terms), and (2.e) requirements modelling of all requirements facets: (2.e.i) business process reengineering (BPR),

(2.e.ii) domain requirements, (2.e.iii) interface requirements, (2.e.iv) machine requirements, and (2.e.v) completion of a full requirements prescription. Intertwined with the requirements prescription parts (item 2., subitems (a–e)) are the analysis parts with (3.a) analysis aiming at identifying inconsistencies, conflicts and incompletenesses, (3.b) requirements validation, (3.c) requirements verification, and (3.d) possible work on establishing a requirements theory.

The new things here are the way in which (2.b) 'acquisition of requirements statements' is pursued, and items (2.c) and (2.c subitems i., ii., and iii.). Essentially (2.b) questionnaires are formulated on the basis of assumed existing domain specifications.

Essentially the questionnaires and the rough sketching of a domain and interface requirements model, after analysis of the requirements statements (3.a), is pursued basically as follows (2.e.ii): which of the entities, functions, events and behaviours described in the domain model must be partially or fully supported by the machine being requirements prescribed? Must those (entities, functions, events and behaviours) being so selected (i.e., projected) be made more determinate, and/or more concretely instantiated, and/or extended, and/or fitted with, or to other, elsewhere developed requirements? Similar for business processes of the "original" domain. Usually they need be reconsidered (2.e.i). Etcetera.

2.3.4 Requirements Engineering Documents

Figures 5 on the next page, 6 on page 10 and 7 on page 11 summarise the plenitude of highly interrelated sets of documents that must all be carefully worked out and carefully correlated.

2.3.5 Requirements Engineering Stages and Steps

Figure 8 on page 11 and 9 on page 12 diagram, in box-and-edge form, the stages and steps of requirements engineering development and their interrelations. The diagram does not give a correct "picture" of the necessity for iteration: going "backwards and forwards" across the development, i.e., across the diagram. Obviously, having a precise understanding of the syntax, semantics and pragmatics of boxes and edges, helps developers and their managers monitor and control (including "contain") iterations.

2.4 The Software Design Process Model

We first rough-sketch narrate the stages and steps of software design development of a software architecture (etc.), then review the documents that should emanate from such development.

1. Information
   (a) Name, Place and Date
   (b) Partners
   (c) Current Situation
   (d) Needs and Ideas (Eurekas, I)
   (e) Concepts & Facilities (Eurekas, II)
   (f) Scope & Span
   (g) Assumptions & Dependencies
   (h) Implicit/Derivative Goals
   (i) Synopsis (Eurekas, III)
   (j) Standards Compliance
   (k) Contracts, with Design Brief
   (l) The Teams
      i. Management
      ii. Developers
      iii. Client Staff
      iv. Consultants
   (m) Plans
      i. Project Graph
      ii. Budget
      iii. Funding
      iv. Accounts
   (n) Management
      i. Assessement
      ii. Improvement
         A. Plans
         B. Actions

Figure 5: Requirements engineering document table-of-contents: information documents

Finally we diagram an essence of the narration and the document table-of-contents.

2.4.1 Software Design, A Narrative

The software design process is here simplified into four stages (Fig. 10 on page 13 items 2.a–d): software architecture design, component design, module design, and (module) program coding. Each of these may consist of two or more steps of development (cf. Fig. 11 on page 14). Between adjacent steps there is a correctness obligation (V:MC:T, verification, model checking and testing). Verification proofs usually are of the kind: $\mathcal{D}, \mathcal{S} \models \mathcal{R}$ which means that the proof that the $\mathcal{S}$oftware implements the $\mathcal{R}$equirements entails reference to the $\mathcal{D}$.

2.4.2 Software Design Documents

Figure 10 on page 13 summarises the plenitude of highly interrelated sets of documents that must all be carefully worked out and carefully correlated.

2.4.3 Software Design Stages and Steps

Figure 11 on page 14 diagram, in box-and-edge form, the stages and steps of software design development and their interrelations. The diagram does not give a correct "picture" of the necessity for iteration: going "backwards and forwards" across the development, i.e., across the diagram. Obviously, having a precise understanding of the syntax, semantics and pragmatics of boxes and edges, helps developers and their managers monitor and control (including "contain") iterations.

3 Review of the Triptych Process

3.1 The Process Model: Diagrams and Tables-of-content

We have surveyed the (mainly) software development processes according to the triptych dogma. We have seen that these processes can be diagrammed and also be "mapped" onto

2. Prescriptions
   (a) Stakeholders
   (b) The Acquisition Process
      i. Studies
      ii. Interviews
      iii. Questionnaires
      iv. Indexed Description Units
   (c) Rough Sketches (Eurekas, IV)
   (d) Terminology
   (e) Facets:
      i. Business Process Re-engineering
         - Sanctity of the Intrinsics
         - Support Technology
         - Management and Organisation
         - Rules and Regulation
         - Human Behaviour
         - Scripting
      ii. Domain Requirements
         - Projection
         - Determination
         - Instantiation
         - Extension
         - Fitting
      iii. Interface Requirements
         - Shared Phenomena and Concept Identification
         - Shared Data Initialisation
         - Shared Data Refreshment
         - Man-Machine Dialogue
         - Physiological Interface
         - Machine-Machine Dialogue
      iv. Machine Requirements
         - Performance
            – Storage
            – Time
            – Software Size
         - Dependability
            – Accessability
            – Availability
            – Reliability
            – Robustness
            – Safety
            – Security
         - Maintenance
            – Adaptive
            – Corrective
            – Perfective
            – Preventive
         - Platform
            – Development Platform
            – Demonstration Platform
            – Execution Platform
            – Maintenance Platform
         - Documentation Requirements
         - Other Requirements
      v. Full Reqs. Facets Doc.

Figure 6: Requirements engineering document table-of-contents: prescription documents

tables-of-content of the documents resulting from respective phases.Of course there is much more to these three phases, their very many stages (within phases), and their potentially very many more steps (within stages) than can be covered in paper form.

3.2 Process Model Semantics

Diagrams, such as those of Figs. 1, 4, 8–9 and 11, reflect some pragmatics, has some syntax and embodies, hopefully some semantics. We wish, here, to emphasise the semantics:

> *What is important to mention here, justifying this separate section, is that each of the boxes of the description, prescription and software design parts of Figs. 4 on page 7, 8 on the next page, 9 on page 12 and 11 on page 14 and each of their interconnecting edges embody a clear set of method principles, techniques and tools with many of these techniques also being pursuable formally and supported, or supportable, by theory-based tools.*

In the following we shall assume that the above *paragraph* on the semantics of the process model diagrams is taken for granted.

3. Analyses
   (a) Requirements Analysis and Concept Formation
      i. Inconsistencies
      ii. Conflicts
      iii. Incompletenesses
      iv. Resolutions
   (b) Requirements Validation
      i. Stakeholder Walk-through and Reports
      ii. Resolutions
   (c) Requirements Verification
      i. Model Checkings
      ii. Theorem Proofs
      iii. Test Cases and Tests
   (d) Requirements Theory
   (e) Satisfaction and Feasibility Studies
      i. Satisfaction: Correctness, unambiguity, completeness, consistency, stability, verifiability, modifiability, traceability
      ii. Feasibility: Technical, economic, BPR

Figure 7: Requirements engineering document table-of-contents: analytic documents

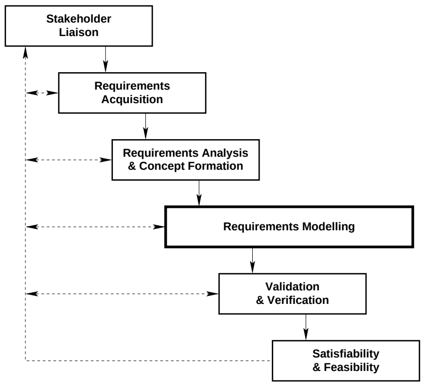

Figure 8: Diagramming a requirements process model

3.3 Informal versus Formal Development

The term 'development' covers any combination of the three phases: domain, requirements or software design only; domain+requirements or requirements+software design, or all three phases "more-or-less" consecutively.

Development can, as shown in Vol. 3 ([9]) of [7, 8, 9], be pursued **informally** or **formally**, and therefore in any "graded scale" combination of these.

0. Informal development means: no formalisation of domain descriptions, requirements prescriptions or software design specifications are attempted. Thus verification cannot be done using formal proofs or model checking. Only testing.

There are, roughly speaking three "points" on the semi-formal to formal scale of development.

1. Systematic development formalises domain descriptions, requirements prescriptions and software design specifications. But that is just about as much formalisation that is attempted.

2. Rigorous development extends systematic development by stating all "crucial"[5] properties and maybe even sketch or carry through the proof or model checking of properties.

[5]We do not here further characterise what we mean by 'crucial'.

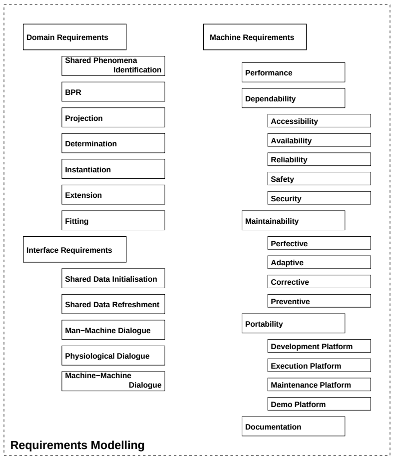

Figure 9: The requirements modelling stage

3. Formal development requires that all necessary (including correctness) properties are formally expressed and theorem proved or model checked.

The triptych paradigm allows for any of these latter three (1.–2.–3.) forms of development.

3.4 Adherence to Phases, Stages and Steps

It is important to stress the following assumption:

> *Adhering to the triptych paradigm, to us, means that all phases, stages and steps as outlined above are followed. This means that documents are produced as per the tables-of-contents shown in Fig. 3, Figs. 5–7 and Fig. 10.*

Our treatment, next, of process assessment and improvement, is based on, i.e., starts with the above assumption.

4 Process Assessment and Improvement Management

4.1 Notions of 'Process Assessment' and 'Improvement'

In order to speak of 'assessment' and 'improvement' we must identify that which is being assessed and improved: the results of following one set of method principles, techniques, tools and their management, over following another such set. Process **assessment** is now about judging adherence of a given process to its process model, pragmatically, semantically and syntactically (pss, usually in reverse order): to which (pss) degrees does the process fulfill what is "laid down" in the process model. Process **improvement** is then about changing the assessed development processes such that the results of using the changed processes are assessed to have been improved.

By "assessment" and "improvement" we first of all mean "assessing and improving documents". The documents are those emanating from activities denoted by nodes and edges of the process model.

1. Information
   (a) Name, Place and Date
   (b) Partners
   (c) Current Situation
   (d) Needs and Ideas
   (e) Concepts and Facilities and Facilities
   (f) Scope and Span
   (g) Assumptions and Dependencies
   (h) Implicit/Derivative Goals
   (i) Synopsis
   (j) Standards Compliance
   (k) Contracts
   (l) The Teams
      i. Management,
      ii. Developers,
      iii. Consultants
   (m) Plans
      i. Project Graph
      ii. Budget, Funding, Accounts
   (n) Management
      i. Assessement Plans & Actions
      ii. Improvement Plans & Actions
2. Software Specifications
   (a) Architecture Design ($S_{a_1} \ldots S_{a_n}$)
   (b) Component Design ($S_{c_{1_i}} \ldots S_{c_{n_j}}$)
   (c) Module Design ($S_{m_1} \ldots S_{m_m}$)
   (d) Program Coding ($S_{k_1}, \ldots, S_{k_n}$)
3. Analyses
   (a) Analysis Objectives and Strategies
   (b) Verification (S_{i_p}, $S_i \sqsupseteq_{L_i} S_{i+1}$)
      i. Theorems and Lemmas L_i
      ii. Proof Scripts $\wp_i$
      iii. Proofs Π_i
   (c) Model Checking ($S_i \sqsupseteq P_{i-1}$)
      i. Model Checkers
      ii. Propositions P_i
      iii. Model Checks $\mathcal{M}_i$
   (d) Testing ($S_i \sqsupseteq T_i$)
      i. Manual Testing
         - Manual Tests $M_{S_1} \ldots M_{S_\mu}$
      ii. Computerised Testing
         A. Unit (or Module) Tests C_u
         B. Component Tests C_c
         C. Integration Tests C_i
         D. System Tests $C_s \ldots C_{s_{i_{t_s}}}$
   (e) Evaluation of Adequacy of Analysis

Legend:

S Specification
L Theorem or Lemma
$\wp_i$ Proof Scripts
Π_i Proof Listings
P Proposition
$\mathcal{M}$ Model Check (run, report)
T Test Formulation
M Manual Check Report
C Computerised Check (run, report)
$\sqsupseteq$ "is correct with respect to (wrt.)"
$\sqsupseteq_\ell$ "is correct, modulo ℓ, wrt."

Figure 10: Software design document table-of-contents

Each such box and each such edge may have many documents "attached" to it, and each such document has its syntax, semantics and pragmatics. The syntax and semantics can usually be given very precise definitions. Hence we can, in a sense, objectively "measure" (assess) whether a document "lives up" to that syntax and that semantics! For pragmatics the "measure" is more subjective. To be able to "measure" process improvement one must therefore attach to each planned document for each box and each edge a "measure" of compliance. Is a document in 100% compliance with those syntactic, semantics and pragmatic measures or is it not? Or more precisely: where on a scale from 0 to 1 lies the quality of a document wrt. an "ideal".

SOFTWARE PROCESS ASSESSMENT 1 **Process Model Syntax and Semantics:** *In order to handle process improvement (à la CMM, from a lower to a higher level) — using the triptych approach — managers (as well as, of course, developers), must be* **intimately familiar with the syntax and semantics of the documents** *produced*

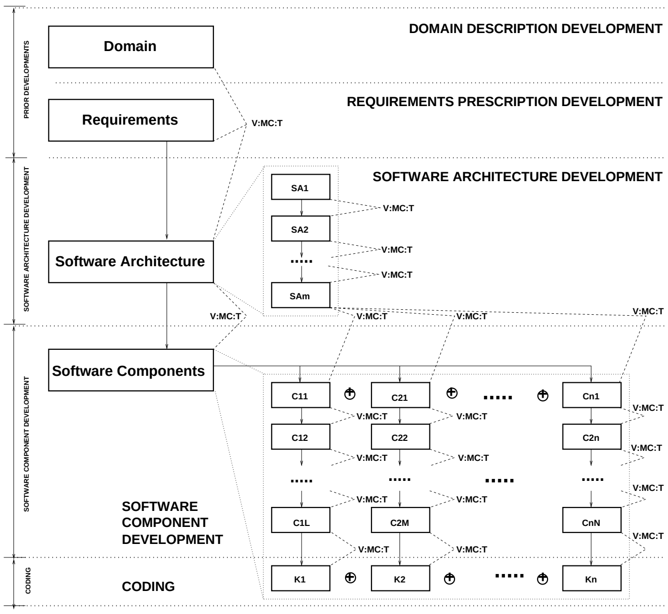

Figure 11: The software design development processes

and expected to be produced by process model node and edge activities. This is a strong requirement and can not be expected by just any software development organisation.

nd there are really no shortcuts.[6] *Process improvement — wrt. the precision of monitoring resource usage — is predicated on this assumption: that management is strongly based on professional awareness of triptych principles, techniques and tools. The "degree"*[7] *to which a development document adheres to the syntax and semantics of the relevant box or edge thus provides an assessment.*

Several groups, worldwide, the most well known is perhaps Praxis High Integrity Systems, http://www.praxis-his.com, practices this on a daily basis. So do many members of ForTIA: The Formal Techniques Industrial Association, www.fortia.org.

Software Process Improvement 1 **Process Model Syntax and Semantics:** *To improve this general aspect of the possible processes that developers and managers might be able to pursue under the banner of the Triptych Process Model one simply has to resort to* **education and training** *. There is no substitute.*

We choose here to **also** "anchor" our discourse of 'process improvement' by referring to the *Capability Maturity Model* (CMM) of Watts S. Humphrey (WSH) [5]. CMM postulates five levels of maturity of development groups. Level 1 being a lowest, in a sense "least desirable", and level 5 being the highest, "most desirable" level of professionalism that WSH finds useful

[6] In other branches of engineering project managers (i.e., project leaders) and developers, the "engineers at floor level" basically all have the same, normalising education. Hence they are intimately familiar with the syntax and semantics of their tasks. The problem is in software engineering.

[7] This "degree" notion is not defined here

to define. Process improvement, by a development group, is now the improvement of the development processes such that the group (i.e., the software house) advances from level i to level $i+j$ where i, j are positive numbers and $i+j$ is less than 6. So let us first review WSH's notion of CMM.

4.2 The CMM: Capability Maturity Model

The following subsection are "lifted" from http://en.wikipedia.org/wiki/Capability_Maturity_Model:

1. **Level 1, Initial**:At maturity level 1, processes are usually ad hoc and the organization usually does not provide a stable environment. Success in these organizations depends on the competence and heroics of the people in the organization and not on the use of proven processes. In spite of this ad hoc, chaotic environment, maturity level 1 organizations often produce products and services that work; however, they frequently exceed the budget and schedule of their projects.

   Maturity level 1 organizations are characterized by a tendency to over commit, abandon processes in the time of crisis, and not be able to repeat their past successes again.

2. **Level 2, Repeatable**:At maturity level 2, software development successes are repeatable. The organization may use some basic project management to track cost and schedule.

   Process discipline helps ensure that existing practices are retained during times of stress. When these practices are in place, projects are performed and managed according to their documented plans.

   Project status and the delivery of services are visible to management at defined points (for example, at major milestones and at the completion of major tasks).

   Basic project management processes are established to track cost, schedule, and functionality. The minimum process discipline is in place to repeat earlier successes on projects with similar applications and scope. There is still a significant risk of exceeding cost and time estimate.

3. **Level 3, Defined**:The organization's set of standard processes, which is the basis for level 3, is established and improved over time. These standard processes are used to establish consistency across the organization. Projects establish their defined processes by the organization's set of standard processes according to tailoring guidelines.

   The organization's management establishes process objectives based on the organization's set of standard processes and ensures that these objectives are appropriately addressed.

   A critical distinction between level 2 and level 3 is the scope of standards, process descriptions, and procedures. At level 2, the standards, process descriptions, and procedures may be quite different in each specific instance of the process (for example, on a particular project). At level 3, the standards, process descriptions, and procedures for a project are tailored from the organization's set of standard processes to suit a particular project or organizational unit.

4. **Level 4, Managed**:Using precise measurements, management can effectively control the software development effort. In particular, management can identify ways to adjust

and adapt the process to particular projects without measurable losses of quality or deviations from specifications.

Subprocesses are selected that significantly contribute to overall process performance. These selected subprocesses are controlled using statistical and other quantitative techniques.

A critical distinction between maturity level 3 and maturity level 4 is the predictability of process performance. At maturity level 4, the performance of processes is controlled using statistical and other quantitative techniques, and is quantitatively predictable. At maturity level 3, processes are only qualitatively predictable.

5. **Level 5, Optimizing**:

   Maturity level 5 focuses on continually improving process performance through both incremental and innovative technological improvements. Quantitative process-improvement objectives for the organization are established, continually revised to reflect changing business objectives, and used as criteria in managing process improvement. The effects of deployed process improvements are measured and evaluated against the quantitative process-improvement objectives. Both the defined processes and the organization's set of standard processes are targets of measurable improvement activities.

   Process improvements to address common causes of process variation and measurably improve the organization's processes are identified, evaluated, and deployed.

   Optimizing processes that are nimble, adaptable and innovative depends on the participation of an empowered workforce aligned with the business values and objectives of the organization. The organization's ability to rapidly respond to changes and opportunities is enhanced by finding ways to accelerate and share learning.

A critical distinction between maturity level 4 and maturity level 5 is the type of process variation addressed. At maturity level 4, processes are concerned with addressing special causes of process variation and providing statistical predictability of the results. Though processes may produce predictable results, the results may be insufficient to achieve the established objectives. At maturity level 5, processes are concerned with addressing common causes of process variation and changing the process (that is, shifting the mean of the process performance) to improve process performance (while maintaining statistical probability) to achieve the established quantitative process-improvement objectives.

4.3 Process Models and Processes

One thing is the process model, viz., the graph-like structures shown in, for example, Fig. 4 on page 7, Figs. 8 on page 11 and 9 on page 12, and Fig. 11 on page 14. (These are syntactic structures, but have semantic meanings.) Another thing is the actual usage of such models, that is, the actual processes that the software developers (domain, requirements and software design engineers) "steer through" when developing domain models, requirements models and software designs.

4.3.1 Graphs and Graph Traversal Traces

Assume some graph-like, let us call it, process model, see Fig. 12 on the facing page.

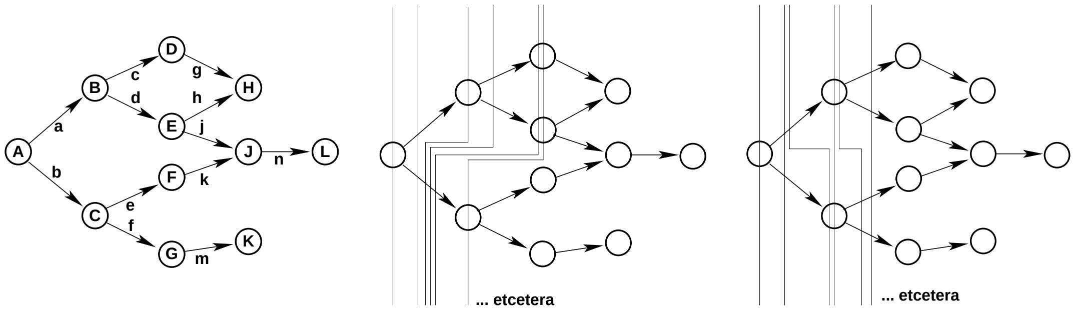

Figure 12: A graph (left) and two (incomplete) traversal traces (center and right)

- So Fig. 12 shows a process model and two traces.
  - REDOs, that is, iterations of phases, stages and steps lead to additional traces.
    - Let us call the totality (set) of these traces for OK traces.
  - And "jumping" or just "skipping" phases, stages and steps lead to further additional traces.
    - Let us call these "jumped" or "skipped" traces for NOK traces.
- A process model thus denotes a possibly infinite set of such traces.

The leftmost part of Fig. 12 shows an acyclic graph. The graph consists of distinctly labeled nodes and (therefrom distinctly labeled) edges. The center and right side of the figure shows some possible traversal traces. By a traversal trace we understand a sequence of wavefronts.

By a wavefront we understand a set of node and edge labels such that no two of these are on the same path from an input (i.e., in-degree 0) to an output (i.e., out-degree 0) node, and such that there is a contribution to the set from any path from an input to an output node.

The third wave of the two traces shown in the two rightmost figures can thus be represented by $\{B,b\}$ and $\{a,C\}$.

4.3.2 Process Models and Processes

A process model is here taken to be a graph: boxes denote activities that result in information and description, prescription or specification documents and edges denote analytic activities that result in documents that record results of (concept formation, consistency, conflict and completeness) analysis, verification, model checking, testing and possibly theory formation.

A development process is any trace over sets of these activities.

Figure 12's center figure thus portrays the following initial trace:

$$\langle\{A\},\{a,b\},\{B,b\},\{c,d,b\},\{D,E,b\},\{D,E,C\},...,\text{etcetera}\rangle$$

Thus a process model denotes a set of such traces.

4.3.3 Incomplete and Extraneous Processes

The trace:

$$\langle\{A\},\{a,b\},\{c,d,b\},\{D,E,b\},\{D,E,C\},...,etcetera\rangle$$

appears to have skipped the activity (phase, stage or step) designated by B. Loosely speaking we call such processes incomplete with respect to their underlying (i.e., assumed) process model (Fig. 12 on the preceding page, the leftmost graph).

The trace:

$$\langle\{A\},\{a,z\},\{X\},\{D,Y,b\},\{D,E,C\},...,etcetera\rangle$$

appears to have performed some activities (z, X, Y) not designated by the process model of Fig. 12 on the page before (the leftmost graph). Loosely speaking we call such processes extraneous (or ad hoc) with respect to their underlying process model.

4.3.4 Process Iterations

The trace

$$\langle\{A\},\{a,b\},\{B,b\},\{a,b\},\{B,b\},\{c,d,b\},\{B,b\},\{c,d,b\},\{D,E,b\},\{D,E,C\},...,etcetera\rangle$$

designates an iterated process. After action B in {B,b} the process "goes back" to perform action b (in {a,b}); and after (either of) actions c or d in {c,d,b} the process "goes back" to perform action B in {B,b}. Loosely speaking we call such processes iterated with respect to their underlying process model.

The above trace only shows simple "one step" (or stage or phase) "backward and then onward" iterations. But the REDO idea, also indicated in Fig. 1 on page 4, can be extended to any number of steps (etc.).

4.3.5 Degrees of Process Model Compliance

We can now define two notions of process model compliance, a syntactic and a semantic. The **syntactic** notion of **process model compliance** has to do with "the degree" to which an actual process matches a possibly iterated, i.e., an OK trace of a process model. The **semantic** notion of **process model compliance** is concerned with adherence to the semantics of boxes and edges.

We shall not, in this paper define these notions precisely — that should be done in a future paper.

Suffice it to summarise that an ongoing process, i.e., an ongoing software development project can be assessed wrt. its syntactic and its semantics compliance wrt. its process model. One can precisely state which activities have been omitted (incompleteness), and which activities were extraneous (or ad hoc). 7

We first deal with syntactic compliance, then, in the next section, with semantics compliance.

SOFTWARE PROCESS ASSESSMENT 2 **Syntactic Process Compliance:** *Given the generic process models diagrammed in Figs. 4 on page 7, 8 on page 11, 9 on page 12 and 11 on page 14, and given the project-specific software development graph as exemplified by Fig. 13 on the next page, one can now, in a process claimed to adhere to these models and graphs quite simply assess whether that actual process follows those diagrams.*

We assume that assessment takes place "regularly", that is, with a frequency higher than process wave transitions, that is, nore often than the process evolves through steps and stages. Otherwise it may be too late (or too cumbersome) to "catch and do" an omitted step.

SOFTWARE PROCESS IMPROVEMENT 2 **Syntactic Process Compliance:** *Adherence to the process model can, at least "formally" (wrt.), be improved by actually ensuring that the process steps and stages (or even phases) that were assessed to not having been performed, that these be performed.*

4.3.6 A "Base 0" for Triptych Developments

By a triptych development we mean a development which applies the principles, techniques and tools as prescribed by the triptych dogma. Either in a systematic, or in a rigorous, or in a formal way. **A triptych development process therefore, "by definition" has its base point at level 4 in the CMM scale.** This does not mean that a software development process which claims to follow the triptych dogma (or the software house within which that process occurs) at least measures at level 4. The dogma sets standards. The process may follow, or may not follow such standards. Whether they are followed or not is now an "easy" matter to resolve. The degree to which the dogma, in all its very many instantiations, is followed is now "fairly easy" to resolve. The "ease" (or "easiness") depends on how well developers and management understands the many triptych principles, techniques and tools, how well they understand the prescribed syntax and semantics of required documents, and on how well they understand their pragmatics, that is, the reason for these principles, techniques and tools.

The pragmatics is what makes management interesting. Well mastered pragmatics allows the managers leeway (i.e., discretion) in the dispatch of their duties, that is, allow them to skip (or "go light" on) certain activities, including choosing whether a step or even a stage should be performed "lightly" or more-or-less "severely", that is, be informal, or formal (and then in a scale from systematic via rigorous to formal).

SOFTWARE PROCESS ASSESSMENT 3 **Planned Syntactic and Semantics Compliance:** *If a process is assessed (SPA) to be in full compliance, syntactically and semantically with the process model then we claim that the software development in this case is at CMM level 4 (or higher).*

SOFTWARE PROCESS IMPROVEMENT 3 **Planned Syntactic and Semantics Compliance:** *If it is assessed that a process has not reached CMM level 4, and that at least CMM level 4 is desired, then one must first secure syntactic compliance, see process improvement #2 (Page 19), thereafter ensure that each of the steps (or stages, or phases) whose semantic compliance was assessed too low be redone and according to their semantic intents.*

4.4 Proactive Measures

The above spoke in general about assessment and improvement.

We are now ready to deal with more specific issues of process assessment and improvement. But first we need to refine our notion of process model.

4.4.1 Project Development Graphs

The process models (i.e., the graphs) are generic. They apply to any development — whatever the software. They must be instantiated to fit the particular problem frame (see [10] as well as Chap. 28 in Vol. 3 of [7, 8, 9]).

Figure 13 shows the project development graph that was used in the development of the Danish Ada compiler [11, 12] (1981–1984).

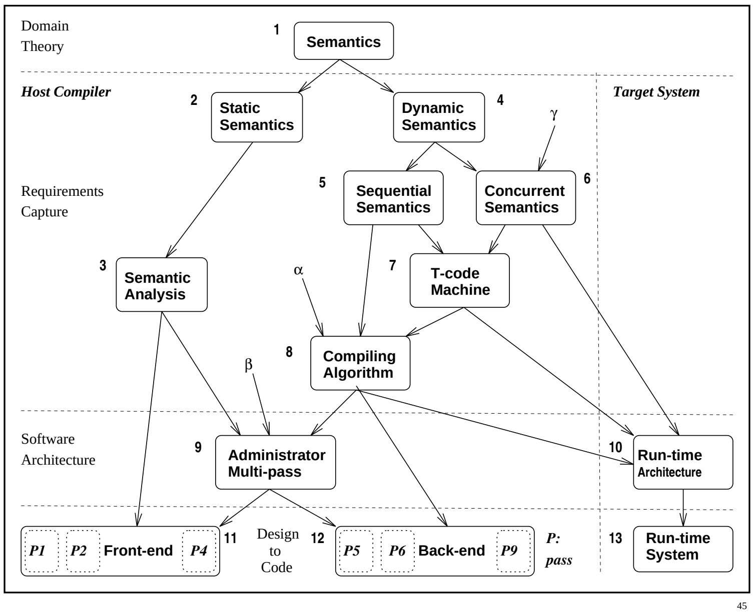

Figure 13: Project development graph: Compiler development

The top horizontal and dashed line of Fig. 13 separates domain engineering from requirements engineering. The domain engineering box ("Semantics") represents a simplification of the usual domain engineering process diagram. (You are to put that usual diagram into the "Semantics" box (a form of supplementation)!) The second horizontal and dashed line of Fig. 13 separates requirements engineering from software design. (Again you are to supplement the requirements engineering and software design boxes etc. of Fig. 13 with the generic process models for requirements engineering and software design.)

The software (domain, requirements, software design) development graphs in the sense of supplementation are orthogonal to process models. They allow more meaningful assignment of semantics to boxes and edges and they allow more specific management (planning, monitoring and control).

In this paper we do not show how to construct a resulting pull graph from the combination of the earlier process models with the later, domain specific graph.

4.4.2 Management

So far, in this paper, we have not dealt with management. Management[8] is about planning, and monitoring and controlling process resource usage — including the quality of the documents emanating from the use of resources. Planning is about scheduling and rescheduling processes and allocating and re- and deallocating resources to (from) processes.

A primary resource in software development is the set of domain and requirements engineers and the set of software designers. Other primary resources are the time, space and tools used by these developers.

Planning — Scheduling and Allocation: Planning starts with instantiating, selecting, or developing a new, tentative, software development graph and detailing (i.e., annotating) it wrt. process model concepts: phases (domain, requirements, software design), stages (stakeholder identification, acquisition, analysis, description (prescription, specification), verification, model checking, testing, validation, etc.), and make allowances for more crucial, detailed steps.

Based on the resulting software development graph management can, in a far more detailed (i.e., granular) way, ascribe resource usage (people, time, offices, equipment, software development tools) to each box and edge, and can schedule these in time and allocate them "in space".

> SOFTWARE PROCESS ASSESSMENT 4 **Resource Planning:** *How can one assess a software development project plan (i.e., graph), that is, something which designates something yet to happen? Well, one can compare to previous software development graphs purporting to cover "similar" (if not identical) development problems and their eventual outcome, that is, the process that resulted from following those graphs. Based on actual resource useage accounts one can now — "to the best of anyone's ability" — draw a software development graph and ascribe resource consumption estimates (time, people, equipment) to each and every node and edge. Thus 'assessment' here was "speculated assessment" of an upcoming project.*

Thus, if that 'speculated assessment' of an upcoming project is felt, by the assessors, i.e., the management, to be flawed, to be questionable, then one has to proceed to improvement:

> SOFTWARE PROCESS IMPROVEMENT 4 **Resource Planning:** *One must first improve the precision with which one designs the domain specific project development graphs. Then the precision with which we associate resource usage with each box and edge of such a graph. Etcetera. Some development projects are very much "repeats" of earlier such projects and one can expect improvement in project development graphs for each "repeat". Other projects are very much tentative, explorative, that is, are actually applied research projects — for which one only knows of a project development graph at the end of the project, and then that graph is not necessarily a "best such"!*

Monitoring & Controlling Resource Usage: As the project (i.e., the process) evolves management can now check a number of things: adherence to schedule and allocation, and adherence to the syntactic and the semantic notions of process model compliance.

[8] We restrict management to the below items. That is: we do not consider product management (which products to develop and in which sequence of deliverables) nor project funding.

Most process models do not possess other than rather superficial and then mostly syntactic notions of compliance. In the triptych process model semantic compliance is at the very core: Every box and every edge of the process models have precise syntax and semantics of the documents that are the expected results of these (box and edge) activities.

> SOFTWARE PROCESS ASSESSMENT 5 **Resource Useage:** *No problems here. As each step (of the development process) unfolds one can assess its compliance to estimated plan.*

Should a resource useage assessment reveal that there are problems (for example: all resources used well before completion of step) then something must be done:

> SOFTWARE PROCESS IMPROVEMENT 5 **Resource Useage:** *Well, perhaps not this time around, when all planned resources have already been consumed — no improvement can undo that — but perhaps "next" time around. An audit may reveal what the cause of the over-consumption was. Either a naïve, too low resource estimate, or unqualified staff, or some simple or not so simple mistakes? Improvement now means: make precautions to avoid a repetition.*

Resource usage is at a very detailed and accountable level and can thus be better assessed. Slips (usually excess usage) can be better foreseen and discovered and more clearly defined remedies, should milestones be missed or usage exceeded, can then be prescribed — including skipping stages and steps whose omission are deemed acceptable.

Skipping stages and steps result in complete, perhaps extraneous (ad hoc) processes. Given that management has an "ideal" process model and hence an understanding of desirable, possibly iterated processes, management can now better assess which are acceptable slips.

4.4.3 From Informal to Formal Development

By process improvement, to repeat and to enlarge on our previous characterisation of what is meant by process improvement, we understand something which improves the quality of resulting software. We "translate" the term 'resulting software' into the term 'resulting documents'. These documents can — as defined on in Sect. 3.3 (Page 11) — be developed eitherinformally (without any use of any formalism other than the final programming language[9]), or systematically formal, or rigorously formal or formally formal!

Informal Development: It is an indispensable property of the triptych approach to software development that the formalisable steps domain engineering, requirements engineering and software design be pursued in some systematic via rigorous to formal manner. Hence the informal aspects of development is restricted to the development of only the informative documents. Informative documents are usually "developed" by project leaders and managers. Hence an "upper" level of management is process assessing and possibly prescribing process improvements to a "lower" level of management!

[9]Thus we do not consider UML to be a formalism. For a "formalism" to qualify as being properly formal it must have a precise syntax, the syntax must have a precise semantics, and there must be a congruent proof system, that is, a set of proof rules such that the semantics satisfy the proof rules.

SOFTWARE PROCESS ASSESSMENT 6 **Informal Development of Informative Documents:** *We refer to Fig. 2 on page 4. That figure lists the kind of documents to be carefully developed — and hence assessed. Since no prescribed syntax, let alone formal semantics can be given for these documents — whose purpose is mainly pragmatic — assessment is a matter of style. It is easy to write non-sensical, "pat" informative documents which do not convey any essence, any insight. Assessment hence has to evaluate: dose a particular, of the many informative documents listed in Fig. 2 on page 4, really convey, in succinct form, an essence of the project being initiated?*

SOFTWARE PROCESS IMPROVEMENT 6 **Informal Development of Informative Documents:** *If an informative document is assessed to not convey its intended message succinctly, with necessary pedagogical and didactical "bravour", then it must be improved. Only "seasoned", i.e., experienced managers can do this.*

Systematic, Rigorous and Formal Development: The development of domain description, requirements prescription and software design documents as well as the development of analytic documents (tests, verification, model checking and validation) can be done in a spectrum from systematically via rigorously to formally.

SOFTWARE PROCESS ASSESSMENT 7 **Staff and Tool Qualification:** *Given the syntax and semantics of the specific step — in the process model — of the tasks to be assessed a (syntax and semantics) a knowledgeable person, a project (task) leader or a manager, can assess compliance. That assessment is greatly assisted by the software tools[10] that support activities of those tasks: If they can process the documents then something seems OK. If not, assessment will have to be negative.*

There are now two distinct, "extreme" reasons for a failure to meet assessment criteria — with any actual reason possibly being a combination of these two "extremes". One is that the quality of the staff performing the affected tasks is not up to expectations. The other is that the tools being deployed are not capable of supporting the problem solution task.

Staff Qualification: If the assessment of 'Systematic, Rigorous and Formal Development of Specifications and Their Analysis' is judged negative due to inadequate development decisions then we suggest the following kind of improvement.

SOFTWARE PROCESS IMPROVEMENT 7 **Staff Qualification:** *It is suggested that improvement, when deemed necessary, takes either of three forms: Either "move" from a systematic to a rigorous level of development, or from a rigorous to a formal level of developement when that is possible and redo the task(s) affected. Or educate and train staff to re-perform the affected task(s) more accurately (while remaining systematic, rigorous, or formal as the case may be. Or replace affected staff with better educated and trained staff and redo the task(s) affected. These kinds of improvement decisions are serious ones.*

[10] These software tools mainly support the use of the main tools, namely the specification languages, their transformation (or refinement) and their proof systems.

4.4.4 Tools

There are different categories of tools. Tools can serve management: for the design of software development graphs (a la Fig. 13 on page 20) and their "fusion" into the appropriate process model diagrams (a la Fig. 4 on page 7, Fig. 8 on page 11 and 9 on page 12, and Fig. 11 on page 14) and for the monitoring and control (i.e., assessment and improvement) of the process with respect to these diagrams. And tools can serve developers: syntactic and semantic description, prescription and software design tools as well as analytic tools: for testing, model checking and verification (proof assistance or theorem provers). These tools embody, that is, represent the formalisms of the textual or diagrammatic notations used — whether Alloy [13], B [14], CafeOBJ [15, 16, 17], Cas l [18, 19, 20], Duration Calculus [21, 22], LSCs [23, 24, 25], MSCs [26, 27, 28], Petri Nets [29, 30, 31, 32, 33], RAISE RSL [34, 35, 7, 8, 9], Statecharts [36, 37, 38, 39, 40], TLA+ [41, 42, 43], VDM-SL [44, 45, 46], or Z [47, 48, 49, 50]. Thus the formal notations of the above listed thirteen languages, whether textual or diagrammatic, or combinatons thereof, are tools, as are the software packages that support uses of these linguistic and analytic means.

Tool Qualification: If assessment of 'Systematic, Rigorous and Formal Development of Specifications and Their Analysis' is judged negative due to inadequate tools then we suggest the following kind of improvement:

> SOFTWARE PROCESS IMPROVEMENT 8 **Tool Qualification:** *Better tools must be selected and applied to the task(s) affected (i.e., judged negatively assessed). These tools are either intellectual, that is, the specification languages, whether textual or diagrammatic, and their refinement and proof systems, or they are the manifest software tools that support the intellectual tools. These are likewise a serious improvement decisions.*

4.5 Review of Process Assessment and Process Improvement Issues

We have surveyed, somewhat cursorily, a number of software process assessment and software process improvement issues. We characterise these from a another viewpoint below.

1. **Process Model Syntax and Semantics Assessment and Improvement:**

   We refer to Page 13.

   The issue here is whether the management and development staff really understands and, to a satifactory degree, can handle the triptych process model in all its myriad of phases, stages and steps, specificationally and analytically, and with all its myriad of documentation demands. If not, then they cannot be effectively assessed and subjected to "standard" improvement measures.

   This is an assessment (and improvement) issue which precedes proper project start.

2. **Syntactic Process Compliance Assessment and Improvement:**

   We refer to Page 19.

   This issue is a "going concern", that is, an ongoing, effort of regular assessment and possibly an occassional improvement. It merely concerns whether a mandated step (or stage or even phase) of development and its expected production of related documents has taken or is taking place.

3. **Planned Syntactic and Semantics Compliance Assessment and Improvement:**

   This is an assessment (and improvement) issue which, in a sense, sets a proper framework for the project: Does management wish to attain at least CMM level 4, or higher or lower? In that sense it precedes project start while determining the rigour with which the next assessments and improvements are to be pursued.

4. **Resource Planning Assessment and Improvement:**

   We refer to Page 21.

   This item of assessment and improvement takes place at project start and may have to be repeated when resource consumption exceeds plans. Assessment and improvement may involve "layers" of project leaders and management.

5. **Resource Useage Assessment and Improvement:**

   We refer to Page 22.

   This item of assessment and improvement takes place at regular intervals during an entire project and involves "layers" of project leaders and management. It may lead to replanning, see Item 4.

6. **Informative Document Assessment and Improvement:**

   We refer to Page 23.

   Informative documents are usually directed at client and software house management and not at software house software engineers. As such they are often the result of the combined labour of client and software house management. Assessments take place while the planned project is being discussed between these partners. Improvements may then be suggested at such mutual project planning meetings.

7. **Staff and Tool Qualification Assessment**

   We refer to Page 23.

   This form of assessment is probably the most crucial aspect of SPA (and hence of SPI). It strikes at the core of software development. The resources spent in what is being assessed conventionally represents a very large, a dominating percentage of resource expenditures.

   Thus this complex of "myriads" of process step, stage and phase (document) assessment must be subject to utmost care.

7. **Staff Qualification Improvement:**

   We refer to Page 23.

   The implications of even minor staff improvement actions may be serious: staff well-being, inavailability of staff, serious delays are just a few. Thus improvement planning must be subject to utmost care, both technically and socio-economically, but also as concerne human relations.

8. **Tool Qualification Improvement:**

   We refer to Page 24.

   The implications of even minor tool improvement actions may be serious: serious re-training or restaffing, serious time delays, and serious hence cost overruns.

4.6 Hindrances to Process Assessment and Improvement

What could be "standard" hindrances to assessment and improvement? And what could be similar hindrances to actually carrying out projects according to the triptych process model?

4.6.1 Lack of Knowledge of Methodology

Both management and development staff must be intimately familiar with the triptych process model and its syntactic, semantic and pragmatic implications, its need for from systematic via rigorous to formal development, its need for the creation, use, maintenance and correlation of myriads of documents, and its need for assessment and possible improvement. Lack of knowledge of the methodology, ever so sporadically, is a hindrance to proper software development processes.

4.6.2 Generation Gaps

Classically we see that young candidates join software houses as software engineers, fluent in the kind of methods: principles, techniques and tools inherent in the triptych approach. They are eager to use these. But they are usually stifled: their slightly older colleagues as well as their project leaders and managers do not possess the same skills, or are outright illiterate wrt. the tripych methods: principles, techniques and tools. Lack of knowledge of the methodology, across generations of staff, is a hindrance to proper software development processes — and even a few years (say ten) count as a generation today.

4.6.3 Lack of Tools

Above we pointed out that there we intellectual tools and there were software tools that support the use of the intellectual tools. Here we mean both.

On one hand, the problem being tackled in a particular software development project may be such that there are, as of today, year 2006, no obvious or no good intellectual tools (and a methodological approach, i.e., a process model) for the properly assessable and improvable pursuit of such a project. On the other hand, even when appropriate intellectual tools are (and a process model is) available there may not be good manifest, that is, software support tools available.

Lack of tools is a serious hindrance to proper software development processes.

4.6.4 Lack of Acceptance

By far the most common hindrance to proper software development processes — such as suggested by the triptych process model — processes that can be properly assessed and for which a continuum of improvement possibilties exists — is (1) the lack of acceptance of what is referred to as "formal methods", and (2) the lack of acceptance of the necessity to do proper domain modelling before tackling requirements.

This is not the time and place to lament on those "facts".

5 Conclusion

It is time to conclude.

5.1 Summary

In Sects. 2. and 3 we have overviewed a rather comprehensive process model, the triptych model which prescribes three development phases: domain engineering, requirements engineering and software design, and which, within these prescribes a number of stages and within these again a number os steps. Phases, stages and steps may be iterated, and phases, stages and steps, as well as the transition between them results in documents. We have modelled process models as acyclic graphs which denote possibly infinite sets of indefinite length traces of waves, where a wave is a set of nodes and edges of the graph not on the same path from an input node (of in-degree 0) to an output node (of out-degree 0), but where subsequences of traces may be repeated (due to process iterations: redoing "previous" tasks).

In Sect. 4 we have then identified a class of seven software process assessment categories and eight software process improvement categories, all in relation to the syntax and semantics of the triptych process model. Finally we briefly touched upon hindrances to process assessment and improvement.

5.2 Future

This is the first time the author has related the triptych model of [7, 8, 9] to SPA and SPI: software process assessment and software process improvement, and hence to CMM, Watts Humphrey's Capability Maturity Model. It has been instructive to do so. Clearly, for actual projects to apply the triptych approach and to carry out the assessments and improvements suggested in this paper, more clarifying directions must be given. And support tools developed.

5.3 Software Procurement

5.3.1 Software

By software we shall here mean not just the executable code and some manuals on how to install, use and possibly repair this code, but also all the documents that emanates from a full project developing this code. That is, all the documents listed in Fig. 3 on page 6, Figs. 5 on page 9, 6 on page 10 and 7 on page 11, and in Fig. 10 on page 13.

5.3.2 Procurement

In software procurement it is therefore natural that the procurement includes as large a set of the documents mentioned in those figures, and that all these documents have passed an assessment with some positive, CMM level-relatable degree of acceptance.

6 Laudatio

Congratulations and many happy returns to Kouichi Kishida.

References

[1] Ian Sommerville. *Software Engineering*. Addison-Wesley, 6th edition, 1982–2001.

[2] Roger S. Pressman. *Software Engineering, A Practitioner's Approach*. International Edition, Computer Science Series. McGraw–Hill, 5th edition, 1981–2001.

[3] Shari Lawrence Pfleeger. *Software Engineering, Theory and Practice*. Prentice–Hall, 2nd edition, 2001.

[4] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, 2002. 2nd Edition.

[5] Watts Humphrey. *Managing The Software Process*. Addison-Wesley, 1989. ISBN 0201180952.

[6] Hans van Vliet. *Software Engineering: Principles and Practice*. John Wiley & Sons, Ltd., Baffins Lane, Chichester, West Sussex PO19 1UD, England, 2000. 2nd Edition.

[7] Dines Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.

[8] Dines Bjørner. *Software Engineering, Vol. 2: Specification of Systems and Languages*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. Chapters 12–14 are primarily authored by Christian Krog Madsen.

[9] Dines Bjørner. *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.

[10] Michael A. Jackson. *Problem Frames — Analyzing and Structuring Software Development Problems*. ACM Press, Pearson Education. Addison–Wesley, Edinburgh Gate, Harlow CM20 2JE, England, 2001.

[11] D. Bjørner and O. Oest. The DDC Ada Compiler Development Project. *[51]*, pages 1–19, 1980.

[12] G.B. Clemmensen and O. Oest. Formal specification and development of an Ada compiler – a VDM case study. In *Proc. 7th International Conf. on Software Engineering, 26.-29. March 1984, Orlando, Florida*, pages 430–440. IEEE, 1984.

[13] Daniel Jackson. *Software Abstractions Logic, Language, and Analysis*. The MIT Press, Cambridge, Mass., USA, April 2006. ISBN 0-262-10114-9.

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

[15] K. Futatsugi, A.T. Nakagawa, and T. Tamai, editors. *CAFE: An Industrial–Strength Algebraic Formal Method*, Sara Burgerhartstraat 25, P.O. Box 211, NL–1000 AE Amsterdam, The Netherlands, 2000. Elsevier. Proceedings from an April 1998 Symposium, Numazu, Japan.

[16] Kokichi Futatsugi and Razvan Diaconescu. *CafeOBJ Report The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*. AMAST Series in Computing – Vol. 6. World Scientific Publishing Co. Pte. Ltd., 5 Toh Tuck Link, SINGAPORE 596224. Tel: 65-6466-5775, Fax: 65-6467-7667, E-mail: wspc@wspc.com.sg, 1998.

[17] Ražvan Diaconescu, Kokichi Futatsugi, and Kazuhiro Ogata. CafeOBJ: Logical Foundations and Methodology. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [52, 53, 20, 54, 43, 50] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.

[18] Michel Bidoit and Peter D. Mosses. CASL *User Manual*. LNCS 2900 (IFIP Series). Springer, 2004. With chapters by T. Mossakowski, D. Sannella, and A. Tarlecki.

[19] CoFI (The Common Framework Initiative). CASL *Reference Manual*, volume 2960 of *Lecture Notes in Computer Science (IFIP Series)*. Springer–Verlag, 2004.

[20] Till Mossakowski, Anne E. Haxthausen, Don Sanella, and Andzrej Tarlecki. CASL — The Common Algebraic Specification Language: Semantics and Proof Theory. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [52, 53, 17, 54, 43, 50] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.

[21] Chao Chen Zhou and Michael R. Hansen. *Duration Calculus: A Formal Approach to Real––time Systems*. Monographs in Theoretical Computer Science. An EATCS Series. Springer––Verlag, 2004.

[22] Chao Chen Zhou, Charles Anthony Richar Hoare, and Anders P. Ravn. A Calculus of Durations. *Information Proc. Letters*, 40(5), 1992.

[23] Werner Damm and David Harel. LSCs: Breathing life into Message Sequence Charts. *Formal Methods in System Design*, 19:45–80, 2001. Early version appeared as Weizmann Institute Tech. Report CS98-09, April 1998. An abridged version appeared in *Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-based Distributed Systems* (FMOODS'99), Kluwer, 1999, pp. 293–312.

[24] David Harel and Rami Marelly. *Come, Let's Play – Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.

[25] Jochen Klose and Hartmut Wittke. An automata based interpretation of Live Sequence Charts. In T. Margaria and W. Yi, editors, *TACAS 2001*, LNCS 2031, pages 512–527. Springer-Verlag, 2001.

[26] ITU-T. CCITT Recommendation Z.120: Message Sequence Chart (MSC), 1992.

[27] ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC), 1996.

[28] ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC), 1999.

[29] Kurt Jensen. *Coloured Petri Nets*, volume 1: Basic Concepts (234 pages + xii), Vol. 2: Analysis Methods (174 pages + x), Vol. 3: Practical Use (265 pages + xi) of *EATCS Monographs in Theoretical Computer Science*. Springer–Verlag, Heidelberg, 1985, revised and corrected second version: 1997.

[30] Carl Adam Petri. *Kommunikation mit Automaten*. Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.

[31] Wolfgang Reisig. *Petri Nets: An Introduction*, volume 4 of *EATCS Monographs in Theoretical Computer Science*. Springer Verlag, May 1985.

[32] Wolfang Reisig. *A Primer in Petri Net Design*. Springer Verlag, March 1992. 120 pages.

[33] Wolfgang Reisig. *Elements of Distributed Algorithms: Modelling and Analysis with Petri Nets*. Springer Verlag, December 1998. xi + 302 pages.

[34] Chris W. George, Peter Haff, Klaus Havelund, Anne Elisabeth 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.

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

[36] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[37] David Harel. On visual formalisms. *Communications of the ACM*, 33(5), 514–530 1988.

[38] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark B. Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *Software Engineering*, 16(4):403–414, 1990.

[39] David Harel and Amnon Naamad. The STATEMATE semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(4):293–333, 1996.

[40] David Harel and Eran Gery. Executable object modeling with Statecharts. *IEEE Computer*, 30(7):31–42, 1997.

[41] Leslie Lamport. The Temporal Logic of Actions. *Transactions on Programming Languages and Systems*, 16(3):872–923, 1995.

[42] Leslie Lamport. *Specifying Systems*. Addison–Wesley, Boston, Mass., USA, 2002.

[43] Stephan Merz. On the Logic of TLA+. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [52, 53, 17, 20, 54, 50] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.

[44] Dines Bjørner and C.B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer–Verlag, 1978. This was the first monograph on *Meta-IV*. .

[45] Dines Bjørner and C.B. Jones, editors. *Formal Specification and Software Development*. Prentice-Hall, 1982.

[46] John S. Fitzgerald and Peter Gorm Larsen. *Developing Software using VDM-SL*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 1RU, England, 1997.

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

[48] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[49] J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.

[50] Martin C. Henson, Steve Reeves, and Jonathan P. Bowen. Z Logic and its Consequences. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [52, 53, 17, 20, 54, 43] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.

[51] Dines Bjørner and O. Oest, editors. *Towards a Formal Description of Ada*, volume 98 of *LNCS*. Springer–Verlag, 1980.

[52] Wolfgang Reisig. The Expressive Power of Abstract State Machines. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [53, 17, 20, 54, 43, 50] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.

[53] Dominique Cansell and Dominique Méry. Logical Foundations of the B Method. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [52, 17, 20, 54, 43, 50] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.

[54] Chris W. George and Anne E. Haxthausen. The Logic of the RAISE Specification Language. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [52, 53, 17, 20, 43, 50] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.