The Subject-Oriented Approach to Software Design and the Abstract State Machines Method

Egon Börger

Università di Pisa, Dipartimento di Informatica, I-56125 Pisa, Italy
boerger@di.unipi.it

Abstract. In [32, Appendix] we have shown that the system which implements the Subject-oriented approach to Business Process Modeling (S-BPM) has a precise semantical foundation in terms of Abstract State Machines (ASMs). The construction of an ASM model for the basic S-BPM concepts revealed a strong relation between S-BPM and the ASM method for software design and analysis. In this paper we investigate this relation more closely. We use the analysis to evaluate S-BPM as an approach to business process modeling and to suggest some challenging practical extension of the S-BPM system.

1 Introduction

The recent book [32] on the Subject-oriented approach to Business Process Modeling (S-BPM) contains a precise high-level definition, namely in terms of Abstract State Machines (ASMs), of the semantics of business process models developed using the S-BPM tool environment.[1] The construction of an ASM which rigorously describes the basic S-BPM concepts revealed an intimate relation between on the one side S-BPM, whose conceptual origins go back to Fleischmann's software engineering book [30, Part II], and on the other side the ASM method [26], a systems engineering method which too has been developed in the 90'ies of the last century by a community effort building upon Gurevich's discovery of the notion of ASM [40] (at the time called by various names, in 1994 'evolving algebras', for the historical details see [11] or [26, Ch.9]).

In this paper we investigate the striking methodological and conceptual similarities (Sect. 2) and some differences (Sect. 3) of these two independent developments. We propose to enhance the current S-BPM system by offering the modeler tool support for the use of the full ASM-refinement method which generalizes the refinement scheme S-BPM provides the software engineer with.

[1] In the appendix, which is written in English, an ASM interpreter is defined for the behavior of such business process models. The software used to transform the pdf-file generated from latex sources into a Word document and printer-control-compatible format produced a certain number of partly annoying, partly misleading mistakes in the printed text. The interested reader can download the pdf-file for the correct text from [63].

We use this analysis to evaluate S-BPM in terms of six well-known principles for reliable software development (Sect. 4), an evaluation which shows that S-BPM provides practitioners with suitable means to precisely and faithfully *capture* business scenarios and *analyze*, *communicate* and *manage* the resulting models.[2]

What nowadays is called S-BPM is really a version tailored for the development of business processes (BPs) of a more general subject-oriented software engineering method and environment for the development of concurrent systems proposed in [30, Part II] and called there SAPP/PASS: 'Structured Analysis of Parallel Programs' with a subject-oriented modelling language named 'Parallel Activities Specification Scheme'. We use invariably the today apparently prevailing term S-BPM to refer to Fleischmann's approach.

We assume the reader to have some knowledge of the basic concepts of at least one of the S-BPM [32] or the ASM methods [26].

2 Common Features of S-BPM and the ASM Method

The S-BPM and ASM methods share their main goal, namely to reliably link the human understanding of real-life processes to their execution by machines via some implementing software. In fact the ASM method is introduced in [26, p.1] by stating that

> 'The method bridges the gap between the human understanding and formulation of real-world problems and the deployment of their algorithmic solutions by code-executing machines on changing platforms.'

Similarly, a recent presentation of the S-BPM approach states for the 'transformation process of model descriptions to executable ones' [33, Sect.2, p.3-4] that:

> 'end-to-end control is what business stakeholders need to build process-managed enterprise' and that
> 'Any mapping scheme should allow propagating the information from a value chain perspective to a software-development perspective in a coherent and consistent way'.

We explain in this section that as a consequence both methods share three major methodological concerns for descriptions of (concurrent) processes:

- the ground model concern (Sect. 2.1),
- the refinement concern (Sect. 2.2),
- the subject-orientation concern to make the executing agents and their distinct internal and external (communication) actions explicit (Sect. 2.3).

[2] In [15] we showed that the OMG standard BPMN [48], the workflow patterns of the Workflow Pattern Initiative [61] and their (academic) reference implementation YAWL [59] fail to achieve this.

Also both come with 'a *simple scientific foundation*, which adds precision to the method's practicality [26, p.1]'.

Although the two methods realize these three concerns differently, due to the more focussed BPM target of the (current incarnation of the) S-BPM method and the different definitions in the two methods of what constitutes agent behavior (described by Subject Behavior Diagrams (SBDs) resp. ASMs, see Sect. 2.3), and although their scientific foundation comes from different sources, the similarities of the two approaches to software engineering are remarkable because 'the *ground model method* for requirements capture, and the *refinement method* for turning ground models by incremental steps into executable code' form together with the concept of ASMs 'the three constituents of the ASM method for system design and analysis' [26, p.13] through which the method

> '*improves current industrial practice* in two directions:
> - On the one hand by accurate high-level modeling at the level of abstraction determined by the application domain ...
> - On the other hand by linking the descriptions at the successive stages of the system development cycle in an organic and effectively maintainable chain of rigorous and coherent system models at stepwise refined abstraction levels.' [26, p.1]

2.1 Ground Model Concern

In the S-BPM literature there is no mention of the name 'ground model' (or 'golden model' as they are called in the semiconductor industry [55]) but the ground model concern is present. The ASM ground model method [7,8,10,12,14] is about constructing prior to code development, as specification for the code, models which are

> 'blueprints that describe the required application-content of programs ... in an abstract and precise form' and are 'formulated in terms of the application domain and at a level of detailing that is determined by the application domain' [14, Sect.1].

Thus ground models satisfy needs of different stakeholders, in particular the domain experts and the software designers. First of all the domain experts (e.g. analysts or users of BPs) need ground models for a *'correct development and understanding* by humans of models and their relation to the application view of the to-be-modeled BP' [15, Sect.5].[3] *Correctness* as used here (together with its

[3] The request in [33, Sect.1,p.1] of a minimal 'semantic distance to human understanding' for S-BPM corresponds to the request for satisfactory ground model ASMs of a 'direct', coding-free relation between the basic domain elements (agents, objects, functions, properties, operations) and the corresponding ASM ground model items [8, Sect.6.2]. The ASM ground model method satisfies this request by offering 'The freedom to choose how to represent the basic objects and operations of the sytem under consideration' and by its attention to 'distinguish between concepts (*mathematical modelling*) and notation (*formalization*)' [8, Sect.5].

companion concept *completeness*) is intrinsically not a mathematical notion, but an epistemological relation between a model and the piece of reality the model is intended to capture, a relation the application experts have to understand and only they (not the software technologists) can judge.

But then also the software designers need ground models, namely as a *complete* specification, where the completeness—every behaviorally relevant feature is stated—makes a correct implementation of the specification reliable. The reliability property links these two roles of ground models. It 'means that the appropriateness of the models can be checked by the application domain experts, the persons who are responsible for the requirements, and can be used by the system developers for a stepwise detailing (by provably controllable ASM refinement steps) to executable code.' [22, p.1923]

Therefore an approach for building satisfactory (i.e. correct, complete and consistent) ground models requires to have solved before 'a *language and communication problem* between the software designers and the domain experts or customers ... the parties who prior to coding have to come to a common understanding of "what to build"' [14, Sect.2.1.1]:

> 'The language in which the ground model is formulated must be appropriate to naturally yet accurately express the relevant features of the given application domain and to be easily understandable by the two parties involved.[4] This includes the capability to calibrate the degree of precision of the language to the given problem, so as to support the concentration on domain issues instead of issues of notation.'(ibid.)(See also the 'language conditions for defining ground models' formulated ibid., Sect.2.3.)

To solve this problem S-BPM starts from two observations of language theory [33, Sect.3, p.5]:

- 'When structuring reality, humans use subjects, predicates and objects.'
- 'humans use natural language structures as primary means to ensure mutual understanding'.

Consequently S-BPM aligns BP descriptions to those three constituents of elementary sentences in natural languages and to the coordination role of communication between subjects.[5] To stay close to natural language, where domain experts formulate process requirements, BP descriptions in S-BPM express the behavior of each subject involved in the BP (read: the agents which perform the described behaviors) as a sequence of possibly guarded basic ('internal') computation or ('external') communication actions of the following form (their content is discussed in Sect. 2.3):

SbpmAction(*Condition*, *subject*, *action*, *object*) =

[4] The S-BPM literature speaks about 'duality of expressiveness' which is needed for the description language [33, Sect.2, p.4].

[5] Notably *communication* and *coordination* appear as two of the seven categories of the *Great Principles of Computing* [28].

if *Condition*(*subject*) **then** *subject* PERFORMS(*action*) **on** *object*

These basic S-BPM actions *mutatis mutandis* correspond to basic ASM transitions, even if the two methods have a different view on what is allowed, in general, to constitute an action and on their parallel resp. sequential execution (see Sect. 2.3 and 3.1). In fact in the S-BPM interpreter the ASM rule BEHAVIOR(*subj*, *state*)—which formalizes the execution by the *subj*ect of the action (called *service*(*state*)) associated with its SID-*state*—has exactly the above form, as the reader can check in [32, p.351].

In this way in S-BPM BPs are modeled using a precise language which is understandable by both parties, domain experts (analysts/managers/users) and software developers: it is constituted by elementary sentences which can be understood as (not formalized) natural language sentences, but nevertheless have a precise *operational* meaning (*modulo* a precise meaning of the constituent parts). The resulting BP ground models are as close to the intended real-world processes (read: their intuitive application-domain-views) as are the subjects, their actions and the objects which are chosen by the analyst (as BP model designers are called) to appear in the ground models. Thus the S-BPM approach offers for BPs an interesting solution to a challenge listed in [22, p.1924], namely 'supporting the extraction of ground model elements from natural language descriptions of requirements'.

The 'abstract operational' character of ASM ground models, which makes them directly executable, mentally by definition as well as mechanically by appropriate execution engines, has been recognized in [8, Sect.7] as crucial for the needed '*experimental validation* of the application-domain-based semantical correctness for *ground models*' [14, p.226]. It is a key criterion also for S-BPM, expressed as follows in [33, Sect.1, p.2]:

> 'The novelty of the approach can be summarized by two key benefits, resulting for stakeholders and organization developers:
>
> 1. Stakeholders need only to be familiar with natural language ... to express their work behavior ...[6]
> 2. Stakeholder specifications can be processed directly *without further transformations*, and thus, experienced as described'.

The *ASM ground model method* realizes the ground model concern in a similar way, but tailored for a more general system engineering setting, using the more comprehensive notion of ASM compared to S-BPM's SBDs as they are used to describe the behavior of BP subjects, see below. Not to repeat for an explanation of this difference what has been described in various articles on the theme [7,8,10,12,14] we invite the reader to read the systematic epistemological discussion of the method in [14]. We limit ourselves here to point to a typical ASM ground model 'at work' S-BPM experts may be interested in, namely the

[6] Obviously such a natural language expression of the work behavior has to be sufficiently precise, in particular to avoid misunderstandings that may arise from cultural differences among the stakeholders.

interpreter model for SBDs in [32, Ch.12 and Appendix] (see also [63]). It illustrates the characteristic properties of ASM ground models by exhibiting the direct, strikingly simple and easy to grasp correspondence between the S-BPM concepts and their mathematical, operational formalization by ASMs.

Scientific Foundation. The just mentioned ASM ground model for an SBD-interpreter constitutes the mathematical part of the scientific foundation of S-BPM. The epistemological part of its foundation is rooted in language theory. The ASM method has its simple scientific foundation directly in mathematical logic and its epistemological roots in a generalized Church-Turing thesis (see Sect. 2.3).

2.2 Refinement Concern

In S-BPM the specification of the processes which constitute a BP model is done in two steps. For each process its SBD (also called PASS graph) describes only the sequence in which the executing subject performs its basic actions. The detailed content of these actions is specified by refinements which describe 'the local variables of a process and the operations and functions defined on the local variables' [30, p.206].

Four types of operations and functions are considered, reflecting the classification of actions described in more detail in Sect. 2.3. Two types of *communication* are specified by describing a) the parameters of the communicated messages and b):

- for to-be-received messages the state change they yield, i.e. their 'effect ... on the values of the local variables, depending on the values of the message parameters and the current values of the loca variables' (ibid.)[7]
- for to-be-sent messages the definition of their content depending on the current state, i.e. 'how the values of the message parameters are obtained from the values of the local variables' (ibid.)[8]

So-called *internal operations* are specified by describing their update effect on the current state (here the values of the local variables), where one is allowed to use so-called *internal functions* (whose applications in the current version of S-BPM are not distinguished any more as separate kind of operations), that is mathematical (side-effect-free), in ASM terminology dynamic functions (i.e. functions whose result for given arguments depends on the current state).

To define these specifications and their implementation in S-BPM the approach 'is open for the integration of existing and proved development methods' [30, p.199] and in particular 'all the object oriented concepts can be applied' (ibid., p.206). These two programming-practice inspired refinement types

[7] This is described in the S-BPM interpreter model by the RecordLocally submachine of Async(*Receive*) and Sync(*Receive*) [32, p.367-368].

[8] This is described in the S-BPM interpreter model by the functions *composeMsg* and *msgData* of the PrepareMsg$_{Send}$ submachine [32, p.361].

in S-BPM (Pass graph refinement and its implementation) are instances of the concept of ASM refinement.

The ASM refinement method was conceived in the context of modelling the semantics of ISO Prolog by ASMs [4,5,6,17] (surveyed in [7]), when I was challenged by Michael Hanus to also develop an ASM for the Warren Abstract Machine (WAM)—an early virtual machine whose optimization techniques changed the performance of Prolog to a degree that made practical applications feasible—and to prove the compilation of ISO Prolog to WAM code to be correct. The challenge was solved by refining the Prolog interpreter model in 12 proven to be correct refinement steps to a WAM interpreter model [23,24,25]. The adopted refinement concept (which has been implemented in KIV for a machine verification of the WAM correctness proof [53,54,50,51,52]) is described in detail in [13]. It

- supports sequences of refinement steps whose length depends on the complexity of the to be described system, and
- links the refinement steps in a documented and precise way so that their correctness can be objectively verified.[9]

Since the ASM refinement notion is in essence more general than the programming-focussed one used in S-BPM, we discuss the details in Sect. 3.2.

2.3 Subject-Orientation Concern

In this section we elucidate for the S-BPM and ASM methods the feature which gave the name to S-BPM and is emphasized in the comparative analysis in [30, Ch.5], [32, Ch.14],[33, Sect.4] as distinctive with respect to traditional system description methods, namely the primary role of agents (called subjects) which execute step by step two distinct kinds of actions following the 'program' (behavioral description) each agent is associated with: communications ('external' actions) and 'internal' actions on corresponding objects.

Agents. *Subjects* are placed into the center of S-BPM process descriptions as the 'active elements' of a process which 'execute functions offered by the passive elements' (i.e. objects of abstract data types) [30, p.199] and have to be identified as first elements of any process description: 'start with identifying the involved subjects and after that define the behaviour specifications of acting parties' [33, Sect.3, p.8]. The ASM method shares this view: in the list of the six 'Fundamental Questions to be Asked' when during requirements capture one starts to construct an ASM ground model the first question is:

> Who are the system *agents* and what are their relations? [26, p.88]

[9] It is an important aspect for certifiability that these verifications are documented to become repeatable by mathematical 'experiment' (read: proof checking). See Sect. 3.3.

This corresponds to the fact that by its very definition an ASM is a family of pairs $(a, Pgm(a))$ of different agents, belonging to a set (that may change at runtime), and the (possibly dynamically associated) programs $Pgm(a)$ each agent executes [26, Def.6.1.1].[10] S-BPM has the same definition: 'An S-BPM *process* ... is defined by a set of subjects each equipped with a diagram, called the *subject behavior diagram* (SBD) and describing the behavior of its subject in the process.' [32, p.348] In both definitions we see multiple agents whose behavior is to execute the (sequential) program currently associated with them. Since this happens in a concurrent context, S-BPM and the ASM method both classify the basic 'actions' an agent can perform in a program step by their role for information exchange among the agents, as we are going to explain now.

Classification of Agent Actions. In S-BPM the 'actions' agents perform when executing their program are of two kinds, to 'exchange information and invoke operations' [30, p.372]. Information exchange is understood as sending or receiving messages. The information exchange actions are named 'external' because they involve besides the executing subject also other, 'external' subjects. The invoked other operations are understood as agent-'internal' (read: communication-free) computations on given objects [30, p.205].

Similarly the ASM method explicitly separates agent-internal operations from external data exchange operations (communication) with other cooperating agents, namely through the so-called classification of locations (i.e. containers of abstract data). Agent-internal operations come in the form of read/writes of so-called *controlled* locations which are performed under the complete and exclusive control of the executing agent. Data exchange (communication with cooperating agents) comes in the two forms of a) reading so-called *monitored* locations that are written by the cooperating agents (an abstract form of receiving messages sent by other agents) and b) writing so-called *output* locations to be read by the cooperating agents (an abstract form of sending messages to other agents).

In the interaction view of an S-BPM subject behavior diagram each internal or communication action counts as one step of the corresponding *subject*, namely to perform what is called the *service* associated with the subject in the given state. In the detailed (refined) interpreter view of the *subject* as defined in [32, Appendix, Sect.3] this 'abstract' interaction-view-step usually is rather complex since it is constituted by the sequence of 'detailed-view-steps' performed by the *subject* to execute the underlying internal or communication action— more precisely in the S-BPM interpreter it is the sequence of the START and all PERFORM steps made by the *subject* to execute its BEHAVIOR(*subject*, *SID_state*), otherwise stated the sequence of detailed steps *subject* performs from the moment when it enters the *SID_state* corresponding to the action (read: the associated *service*) until the moment when it exits that state to enter the *SID_state*$'$ corresponding to the next action, see [32, p.351].

[10] To name the agent can be omitted (only) in the special case where a single ASM is contemplated (which may interact with an environment that is considered as run by one other agent).

The ASM method started out to provide in full generality the means to abstract into one single-agent step an entire internal computation which may be needed to perform an action in a given state. Therefore one has to separately describe the interaction the considered agent may have with the cooperating agents in its environment to perform the action, namely receiving data from cooperating agents before it starts the abstract step and sending data to cooperating agents after (probably as a result of) the abstract step. The agent's sending interactions are collectively incorporated into its one abstract step, namely as updates of all corresponding output locations; this is without loss of generality given the parallel nature of a single ASM step which performs simultaneously an entire set of location updates. Analogously the agent's receiving interactions directly preceding (and probably influencing) its abstract step are collectively described by a separate so-called 'environment' step which precedes the agent's abstract step and is assumed to be executed by another agent representing the environment of the considered agent; this environment step performs simultaneously all the relevant updates of the corresponding monitored locations, thus completing the definition of the state in which the considered agent performs (the internal part of) its abstract step (see the formal definition in [26, Def.2.4.22, p.75]).

The difference in the technical S-BPM/ASM realization of the identical concept of distinguishing internal and external 'actions' is a result of the different origins of the two methods. The motivating target of S-BPM was to incorporate in an explicit and practically feasible way into the software engineering techniques of the time the missing high-level concept of communication between process agents, in particular for developing BPs where communication is fundamental to control the actions of the cooperating agents. Therefore it was natural to develop an orthogonal communication concept (inspired by CCS [47] and CSP [42]) which is compatible with the principal (at the time prevailingly object-oriented) programming concepts and their implementation so that it can be integrated in a modular way into any practical software engineering method. This led to the interesting input-pool-based S-BPM notion of a synchronous or asynchronous communication (send or receive) 'step' as *pendant* to and *à la pari* with any internal computation 'step'. The notion of an ASM the development of the ASM method started from grew out of an epistemological concern, namely to sharpen the Church-Turing thesis for 'an alternative computation model which explicitly recognizes finiteness of computers' [38,39] (see [11],[26, Ch.9] for the historical details). Therefore it was natural to abstract for the definition of what constitutes an ASM step from any particular form of communication mechanism and to represent a communication (receive or send) action abstractly the same way as any other basic computational action, namely as reading the value of an abstract 'memory location' resp. as updating (writing) it—clearly at the price of having to define an appropriate practical communication model where needed, a task Fleischmann accomplished for S-BPM with his input-pool concept. This concept provides an interesting contribution to the challenge listed in [22, p.1923] to develop 'practically useful patterns for communication and synchronization of multi-agent ASMs, in particular supporting omnipresent calling structures

(like RPC, RMI and related middleware constructs) and web service interaction patterns.'[11]

Behavior of Agents. In S-BPM the behavior of a single agent is represented by a graph of the Finite State Machine (FSM) flowchart type (called SBD or PASS graph) which 'describes the sequences in which a process sends messages, receives messages and executes functions and operations' [30, p.207]. This corresponds exactly to the so-called control-state ASMs [26, Sect.2.2.6] and their FSM-flowchart like graphical display[12] so that not surprisingly the high-level S-BPM interpreter in [32, Appendix, Sect.7] for the execution of SBDs is defined as a control-state ASM.

3 Differences between S-BPM and the ASM Method

In this section we discuss three major differences between the S-BPM and the ASM method. They concern the notion of *state* and state *change* (update) by actions of agents (Sect. 3.1), the notion of *refinement* of models (Sect. 3.2) and the *verification* concern which helps in the ASM method to increase the system reliability and to reduce the amount of experimental system validations (Sect. 4). Through these features the ASM method offers the practitioner additional possibilities for certifiably correct design of software-intensive systems, although we see no reason why they could not be included into S-BPM, as we are going to suggest, to increase the degree of reliability of S-BPM-designed BPs by certifiable correctness.

3.1 Notion of State and State Change

State. As we have seen in Sect. 2.2, S-BPM shares the traditional programming view of states: 'the values of all local variables define ... the local state of a process' [30, p.206]. In contrast, 'the notion of ASM *states* is the classical notion of mathematical *structures* where data come as abstract objects, i.e. as elements of sets (also called *domains* or *universes*, one for each category of data) which are equipped with basic operations (partial *functions* in the mathematical sense) and predicates (attributes or relations).'[26, p.29] In logic these structures, which have been formulated as a concept by Tarski [58] to define the semantics of first

[11] The various theoretical communication concepts surveyed in [41] appear to have been defined to suit parallel and so-called interactive forms of the ASM thesis and seem to have had no practical impact.

[12] Control-state ASMs have been introduced in [10] as 'a particularly frequent class of ASMs which represent a normal form for UML activity diagrams and allow the designer to define machines which below the main control structure of finite state machines provide synchronous parallelism and the possibility of manipulating data structures.' [26, p.44]

order logic formulae, are also called Tarski structures.[13] The relevant fact for the modelling activity is that the sets and functions which form the state of an ASM can be chosen in direct correspondence with the to-be-modelled items of the application domain, tailored with 'the greatest possible *freedom of language*' [8, Sect. 5] to the intended level of abstraction of the model and 'avoiding the formal system straitjacket' (ibid.). Thus ASM states realize an advice from a great authority: 'Data in the first instance represent abstractions of real phenomena and are preferably formulated as abstract structures not necessarily realized in common programming languages.' [62, p.10]

To provide a characteristic example we can refer to the abstract elements and functions which appear in the ASM model for S-BPM [32, Appendix] as part of the interpreter state, like all the SBD-graph structure related items, the *service*s associated with SID-states and their completion predicate *Completed*, *inputPool* with its related functions, the different sets providing *Alternative*s together with their *select*ion functions, message related functions to *composeMsg* from *msgData*, etc.

Also the object oriented slightly more complex version of the programming view of states as defined above, which comes with the suggestion to use object oriented techniques for the specification of PASS graph refinements [30, p.210], is an instance of the ASM notion of state since 'the instantiation of a relation or function to an object o can be described by the process of parameterization of, say, f to the function $o.f$, which to each x assigns the value $f(o, x)$.'[26, p.29][14]

State Change. The most general kind of a basic action to change a structure or algebra (i.e. a set of functions) appears to be that of a *function update*, i.e. change the value of a function at given arguments, which has the following form:

$$f(t_1, \ldots, t_n) := t$$

Such updates, executed by an agent (denoted by **self**) under appropriate conditions which guard the application of ASM rules:

$$\textsc{AsmRule}_{\mathbf{self}}(\mathit{Condition}, \mathit{Updates}) = \mathbf{if}\ \mathit{Condition}\ \mathbf{then}\ \mathit{Updates}$$

are exactly what constitutes the basic action of an ASM agent in a *state*, where f is an arbitrary n-ary function symbol[15] and $t_1, \ldots, t_n$ are arbitrary terms (expressions) at whose values in the current *state* the new value of the function (which will be the value of the successor state of the current *state*) is set to the value of t in the current *state* (if the indicated condition under which this action is requested to be performed is true in the current state). Given the

[13] If predicates are considered to be canonically represented by their characteristic functions, a Tarski structure becomes what is called an algebra. Viewed this way an ASM state is a set of functions or Parnas tables [49,9].

[14] Recently this parameterization facility for ASM states has been exploited to define a general ambient concept in terms of ASMs [16].

[15] 0-ary functions f, i.e. where $n = 0$, are the variables of programming.

abstract nature of the functions and objects (elements of the universe) which constitute an ASM state one can express updates at any level of abstraction, using corresponding functions f and expressions t_i, t of given complexity or level of abstraction.

This lifts variable assignment to *destructive assignment at any level of abstraction* and thus supports abstract operational modelling (providing what is nowadays often called execution semantics of a system). A typical use is illustrated by the abstract yet precise definition of the two communication actions $ComAct \in \{Send, Receive\}$ of S-BPM agents by the interpreter submachines ASYNC($ComAct$) and SYNC($ComAct$) in [32, Appendix,3.3.,3.4].

Expressivity Question. Due to its original epistemological goal the definition of ASMs had to solve an expressivity issue for the proposed simple algorithmic language, namely to guarantee that this language provides whatever may be needed to 'directly' (coding-free and thus without extraneous overhead) model any computational system. This is what the ASM thesis [38,39] was about and explains why a) the states of ASMs have to be Tarski structures and why b) differently from their static nature in mathematics and logic here these structures must be treated as updatable by basic actions of ASM agents, namely by (a set of simultaneous)[16] updates.

By its focus on modelling BPs by sets of SBDs each of which is described by constructs that are close to sentences of natural language, S-BPM derives the guarantee to be expressive enough for modelling any desired BP from the expressivity of natural language. The price paid is the focus of ground models on the level of abstraction of (sets of) SBDs which are reached by system decomposition (using data flow diagram techniques) until every communicating subject has become explicit,[17] as will become clearer in the next section where we compare the programming-oriented S-BPM refinement concept explained in Sect. 2.2 with the more general ASM refinement notion.

A positive return is the ease with which an S-BPM model can be transformed into a precise (though verbose) natural language text, essentially by paraphrasing each SBPMACTION in every SBD of the model by the obvious corresponding natural language sentence. Given the similarity between ASM rules and SBPMACTIONS, in a similar way such a transformation can also be defined

[16] The synchronous parallelism of single-agent actions in the ASM-computation model, which differs from the sequential-program view of actions of S-BPM agents, provides 'a rather useful instrument for high-level design to *locally describe a global state change*, namely as obtained in one step through executing a set of updates' and 'a convenient way to *abstract from sequentiality* where it is irrelevant for an intended design' [8, p.30].

[17] This interesting termination criterion for the 'decomposition of a system into processes'—the first of the two major system development steps in the S-BPM method—is a consequence of the communication focus (read: subject orientation): 'Finally all processes and shared objects, the messages exchanged between processes and the shared objects they use, are identified.' [30, p.204 and Ch.10]

for ASM models, as has been illustrated in [20]. There the contributing authors of the book [34] had been asked to formulate in natural language a precise and complete set of requirements for a small case study by first defining a formal specification which captures the given informal requirements and then retranslating this specification into natural language. For S-BPM a converter has been written which transforms S-BPM models into natural language texts [31] (see also [56]). Although we believe that the methodological better way to explain and document ASMs (and also S-SBM models) is to use a *literate modeling* style in the spirit of Knuth's literate programming [45], it could nevertheless be useful to write a similar *Asm2NatLang* converter to facilitate the integration of ASMs into natural language S-BPM documents for users who are not familiar with symbolic mathematical notations.

3.2 Refinement Concept

The conceptual distance between an SBD (PASS graph) to its refinement, which represents an operational specification of the communication and internal actions the subject performs in the SBD, is not very large. The next step (which we consider as another refinement step) consists in the coding of this specification where the S-BPM method adopts 'methods which are common in standard sequential programming' [30, p.296]. Therefore alltogether the 'semantic gap' between a user model (ground model PASS graph) for a BP and its code is judged not to be very large. In fact it is claimed that 'Once the interaction patterns among actors (subjects) have been refined in terms of exchange of messages, suitable program code can be generated automatically' [33, Sect.1, p.2]; this has to be understood *cum grano salis*, probably meant to hold for 'the standard part of the code' [30, p.295] resp. for code meaning method headers.

This does not solve the problem in case the distance between a ground model and the code is too large to be bridged in one or two steps in such a way that a human can understand the refinement and verify its correctness. Such a situation was at the origin of the ASM refinement method [13] and is typical for its successful applications. Mentioning a few examples should suffice here to illustrate the practical relevance of the ASM refinement notion.

The historically first example is the Prolog-to-WAM compiler verification mentioned in Sect. 2.2 where we needed 12 refinement steps to explain Warren's ideas and to prove the main theorem. The refinement correctness proofs have later been machine verified using the KIV system [53,54]. Interestingly enough to enable the KIV machine to finish its proof, for one of the optimizations in the WAM an additional refinement step had to be introduced into the handwritten proof developed to convince ourselves and our peers. The elaboration of the method for the Occam/Transputer parallel computation model (with nondeterminism) yielded 17 natural refinement steps [18] to explain the rationale and prove the correctness of the standard (INMOS) compilation scheme.

Another real-life example to be mentioned (among many others concerning architectures, control software, protocols, algorithms, etc. and surveyed in [26, Ch.9]) is the stepwise refinement of ASM interpreters for Java and the JVM,

using both horizontal and vertical refinement steps. These models have been used to verify various properties of interest for the language and its virtual machine, like type safety, compiler correctness, soundness and correctness of the bytecode verifier, soundness of thread synchronization, etc. The reader can find the details in the JBook [57]. That the method could be applied also to C# [19] and .NET CLR [35,37,36] should not come as a surprise.

A natural place to integrate into S-BPM the ASM refinement method is where one has to code complex internal actions of a subject. It is still a challenge to provide tool support for the ASM refinement method, in particular in combination with verifications of refinement correctness, e.g. building upon the implementation of the ASM refinement concept in [50] which has later been extended and been used for numerous other verification projects, see www.informatik.uni-augsburg.de/swt/projects/. Some first steps in this direction seem to appear in the area of software product lines where feature-based modeling is linked to the stepwise validation and verification of properties [60,44,3,27].

3.3 Verification Concern

The presentation of the ASM method quoted at the beginning of Sect. 2 continues as follows:

> 'It covers within a single conceptual framework both *design and analysis*, for procedural single-agent and for asynchronous multiple-agent distributed systems. The means of analysis comprise as methods to support and justify the reliability of software both *verification*, by reasoning techniques, and experimental *validation*, through simulation and testing.' [26, 1]

This shows how much the ASM method cares about both, verification by proving model properties and validation by simulation and testing of models. However it turned out to be an advantage for their use in systems engineering to pragmatically separate these two activites from the modeling (design) activity [8, Sect.4,5], differently from what do other methods (notably the conceptually very close B-method [1,2]) which link design and verification (definition and proof) to always go together.

The ASM method allows one to validate and/or verify properties of models at any level of abstraction since by their definition

- ASMs are mathematical objects so that they satisfy the rigour needed to enter a mathematical or machine supported proof,
- ASMs are conceptually executable, due to their operational character, and have been made mechanically executable by various tools.[18]

Verification cannot replace validation, but as early design-error detection technique it can considerably reduce the amount of testing and error correction after the system is built.

[18] See [26, Ch.8] for a survey of various ASM verification and validation tools and [29] for the more recent CoreASM execution engine.

The SAPP/PASS approach shares the validation and verification concern. For 'checking whether a process is correct' two aspects are distinguished [30, p.312, Sect.16.3]:

- A system must have certain properties, e.g. livelock free, deadlock free which are independent of the application. This is implicit correctness.[19]
- A specified system must do what a designer has intended. This is explicit correctness.

Both aspects are reported to have been supported by prototypical Prolog-based validation tools providing for each system modeled in PASS a sort of expert system which 'allow(s) the behavior of a process system to be analysed and can determine whether a system does what it was intended to do' (ibid., p.321).

However this verification concern seems not to be supported by the present S-BPM tool set, although the validation concern is, namely by a testing mechanism that allows one to feed concrete values for messages and function arguments and values into the system to run BP scenarios prior to coding method bodies[20].

We suggest to integrate into the current S-BPM system the possibility to

- formulate application-specific BP properties of interest to the user or manager, presumably ground model properties which go beyond the usual graph-theoretic properties like liveness, fairness, deadlock fredom, etc.,
- prove such properties for the ground model as well as their preservation through ASM refinement steps of internal actions,
- document the properties and their verifications so that they can be checked (also by third parties like certification bodies) and used to certify the correctness of the BP implementation.

This could be realized for any of the reasoning techniques the ASM method allows one to apply for the mathematical verification of system properties, at different levels of precision and under various assumptions, e.g. [14, Sect.1]

- outline of a proof idea or *proof sketch* whereby the designers communicate and document their design idea,
- *mathematical proof* in the traditional meaning of the term whereby a design idea can be justified as correct and its rationale be explained in detail,
- *formalized proof* within a particular logic calculus,
- *computer-checked* (automated or interactive) *proof*.

Each technique comes with a different amount of tool support[21] and of effort and cost to be paid for the verification and provides a different level of objective, content-based 'certification' of the professional quality of the analysed system.

---

[19] We have pointed out in [15, 4.2] that for BPs 'implicit correctness' properties are less interesting than the ones for 'explicit correctness' which typically are ground model properties to be preserved through refinement steps.

[20] This is exactly the method used in the Falko project at Siemens to validate the ASM ground model for the given scenarios, see [21]

[21] [26, 9.4.3] surveys some tool supported ASM verifications.

4 Evaluation of S-BPM

In this section we evaluate S-BPM as an approach to BPM (Sect. 4.2) using six classical evaluation criteria for practical software engineering methods (Sect. 4.1).

4.1 The Evaluation Criteria

The three major purposes of business process (BP) descriptions are the *design and analysis*, the *implementation* and the *use* of models of BPs. For each purpose pursued by the various BP stakeholders the models play a specific role, namely to serve a) as conceptual models (in particular for high-level development-for-change and management support), b) as specification of software requirements that are implemented by executable models and c) as user model for process execution, monitoring and management. This is reflected in the following six criteria (paraphrased from [15, Sect.5]) a satisfactory BPM system must satisfy:

Ground Model Support. Provide *support for a correct development and understanding* by humans of models and their relation to the application view of the to-be-modeled BP, which is informally described by the process requirements. This human-centered property is often neglected although it is the most critical one for software development systems in general[22] and in particular for BPM systems. It is crucial to support such an understanding *for both model design and use* because these models serve for the communication between

- the BP expert, who has to explain the real-world BP that is to be implemented,
- the IT expert who needs a precise specification of the coding goal,
- the BP user who applies or manages the implemented process and needs to understand for his interaction with the system that his process view corresponds to what the code does.

Refinement Support. Provide *support for faithful implementations* of models via systematic, controlled (experimentally validatable and/or mathematically verifiable) refinements. This model-centered property is methodologically speaking the simpler one to achieve because an enormous wealth of established refinement, transformation and compilation methods can be used for this—if the construction of satisfactory (precise, correct, complete and minimal) ground models is supported the implementation can start from.

Change Management. Provide *support for effective change management* of models. This involves the interaction between machines and humans who have to understand and evaluate machine executions for BP (ground or refined) models, bringing in again conceptual (ground model and refinement) concerns when it comes to adapt the system to evolutionary changes.

[22] See the discussion in [14] for the verified software challenge [43] originally proposed by Hoare.

Abstraction Provide *support for abstraction* to help the practitioner in two respects:

- in the daily challenge to develop abstract models (ground models and their stepwise refinements) out of concrete, real-life problem situations. This implies, in particular, the availability in the modeling language of a rich enough set of abstract data types (sets of objects with operations defined on them) to use so that one can
  - express the application-domain phenomena to be modeled (objects and actions) at the conceptual level without the detour of language-dependent encodings;
  - refine the abstractions in a controlled manner by more detailed operations on more specific data structures.
- to develop coherent definitions of different system views (control-flow view, data flow view, communication view, view of the actors, etc.).

Modularization. Provide *support for modularization* through rigorous abstract behavioral interfaces to support structured system compositions into easy-to-change components.[23] For BPM it is particularly important that *modeling-for-change* is supported at all three major stakeholders levels: at the *Ground Model* and *Change Management* support levels because it is the BP users and managers who drive the evolutionary adaptation of BP models, at the *Refinement* support level because the high-level model changes have to be propagated (read: compiled) faithfully to the implementing code.

Practical Foundation. Come with a *precise foundation a practitioner can work with*, i.e. understand and rely upon when questions come up about the behavioral meaning of constructs used by the tool.

4.2 Applying the Criteria to S-BPM

In this section we recapitulate what has been said showing that S-BPM [32] and its tool [46] support correct development and understanding, faithful implementation and effective management of BP models via practical abstraction and modularization mechanisms which are defined on the basis of a fundamental epistemological and mathematically stable foundation.

S-BPM satisfies the *Ground Model* criterion, as shown in Sect. 2.1.

In Sect. 2.2 we have explained to which extent S-BPM satisfies the *Refinement* criterion and in Sect. 3.2 how it can be enhanced to satisfy the full *Refinement* criterion. Modulo the same remark S-BPM satisfies the *Abstraction* criterion.

The *Change Management* criterion is satisfied by S-BPM via its technique to decompose BPs into sets of SBDs, for which in turn modeling for change is supported by two model extension schemes which allow the modeler to smoothly

[23] These two features, abstraction and modularization, also appear in the *Design* section of *Great Principles Category Narrative* in [28] listed under *simplicity* as one of the five 'driving concerns' of software design and used to 'overcome the apparent complexity of applications'.

integrate into a given SBD some new (whether normal or interrupt) behavior [32, Appendix, Sect.6].

To satisfy the *Modularization* criterion S-BPM contributes in various ways. Besides the just mentioned constructs for extending normal or interrupt behavior actions can be atomic or composed. In particular structured alternative actions are available. To accurately model alternative (whether asynchronous or synchronous) communication actions it is sufficient to use an appropriate selection function and the traditional iteration construct to loop through the offered alternatives [32, Appendix, Sect.3.1]. For alternative internal actions a structured split-join mechanism is used which allows the modeler to have the selection simply as non-deterministic choice or to condition the choice by static or dynamic possibly data-related criteria (ibid., Sect.4). Further modular composition constructs include the rigorously defined use of macros, of a normalization to interaction views of SBDs and support for process hierarchies (networks) (ibid., Sect.5).

Notably the model itself which defines the semantics of these features is formulated in a modular way using stepwise ASM refinement (ibid.).

Last but not least S-BPM has a *Practical Foundation* via the accurate definition of its semantics using the language of ASMs—a mathematically precise, wide-spectrum action description language which uses rules of the the same form as guarded basic SBD actions (see Sect. 2.1) and thus is familiar to all BP stakeholders.

References

1. J.-R. Abrial. *The B-Book*. Cambridge University Press, Cambridge, 1996.
2. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, Cambridge, 2010.
3. D. Batory and E. Börger. Modularizing theorems for software product lines: The Jbook case study. *J. Universal Computer Science*, 14(12):2059–2082, 2008.
4. E. Börger. A logical operational semantics for full Prolog. Part I: Selection core and control. In E. Börger, H. Kleine Büning, M. M. Richter, and W. Schönfeld, editors, *CSL'89. 3rd Workshop on Computer Science Logic*, volume 440 of *Lecture Notes in Computer Science*, pages 36–64. Springer-Verlag, 1990.
5. E. Börger. A logical operational semantics of full Prolog. Part II: Built-in predicates for database manipulation. In B. Rovan, editor, *Mathematical Foundations of Computer Science*, volume 452 of *Lecture Notes in Computer Science*, pages 1–14. Springer-Verlag, 1990.
6. E. Börger. A logical operational semantics for full Prolog. Part III: Built-in predicates for files, terms, arithmetic and input-output. In Y. N. Moschovakis, editor, *Logic From Computer Science*, volume 21 of *Berkeley Mathematical Sciences Research Institute Publications*, pages 17–50. Springer-Verlag, 1992.
7. E. Börger. Logic programming: The Evolving Algebra approach. In B. Pehrson and I. Simon, editors, *IFIP 13th World Computer Congress*, volume I: Technology/Foundations, pages 391–395, Elsevier, Amsterdam, 1994.
8. E. Börger. Why use Evolving Algebras for hardware and software engineering? In M. Bartosek, J. Staudek, and J. Wiederman, editors, *Proc. SOFSEM'95, 22nd*

Seminar on Current Trends in Theory and Practice of Informatics, volume 1012 of *Lecture Notes in Computer Science*, pages 236–271. Springer-Verlag, 1995.
9. E. Börger. Evolving Algebras and Parnas tables. In H. Ehrig, F. von Henke, J. Meseguer, and M. Wirsing, editors, *Specification and Semantics*. Dagstuhl Seminar No. 9626, Schloss Dagstuhl, Int. Conf. and Research Center for Computer Science, July 1996.
10. E. Börger. High-level system design and analysis using Abstract State Machines. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Current Trends in Applied Formal Methods (FM-Trends 98)*, volume 1641 of *Lecture Notes in Computer Science*, pages 1–43. Springer-Verlag, 1999.
11. E. Börger. The origins and the development of the ASM method for high-level system design and analysis. *J. Universal Computer Science*, 8(1):2–74, 2002.
12. E. Börger. The ASM ground model method as a foundation of requirements engineering. In N.Dershowitz, editor, *Verification: Theory and Practice*, volume 2772 of *LNCS*, pages 145–160. Springer-Verlag, 2003.
13. E. Börger. The ASM refinement method. *Formal Aspects of Computing*, 15:237–257, 2003.
14. E. Börger. Construction and analysis of ground models and their refinements as a foundation for validating computer based systems. *Formal Aspects of Computing*, 19:225–241, 2007.
15. E. Börger. Approaches to modeling business processes.
A critical analysis of BPMN, workflow patterns and YAWL. *J. Software and Systems Modeling*, 2011. DOI: 10.1007/s10270-011-0214-z.
16. E. Börger, A. Cisternino, and V. Gervasi. Ambient Abstract State Machines with applications. *J. Computer and System Sciences*, 2011. Special Issue in honor of Amir Pnueli, see http://dx.doi.org/10.1016/j.jcss.2011.08.004.
17. E. Börger and K. Dässler. Prolog: DIN papers for discussion. ISO/IEC JTCI SC22 WG17 Prolog Standardization Document 58, National Physical Laboratory, Middlesex, England, 1990.
18. E. Börger and I. Durdanović. Correctness of compiling Occam to Transputer code. *Computer Journal*, 39(1):52–92, 1996.
19. E. Börger, G. Fruja, V. Gervasi, and R. Stärk. A high-level modular definition of the semantics of C#. *Theoretical Computer Science*, 336(2–3):235–284, 2005.
20. E. Börger, A. Gargantini, and E. Riccobene. Abstract State Machines. A method for system specification and analysis. In M. Frappier and H. Habrias, editors, *Software Specification Methods: An Overview Using a Case Study*, pages 103–119. HERMES Sc. Publ., 2006.
21. E. Börger, P. Päppinghaus, and J. Schmid. Report on a practical application of ASMs in software design. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory and Applications*, volume 1912 of *Lecture Notes in Computer Science*, pages 361–366. Springer-Verlag, 2000.
22. E. Börger and A. Prinz. Quo Vadis Abstract State Machines? *J. Universal Computer Science*, 14(12):1921–1928, 2008.
23. E. Börger and D. Rosenzweig. From Prolog algebras towards WAM – a mathematical study of implementation. In E. Börger, H. Kleine Büning, M. M. Richter, and W. Schönfeld, editors, *CSL'90, 4th Workshop on Computer Science Logic*, volume 533 of *Lecture Notes in Computer Science*, pages 31–66. Springer-Verlag, 1991.
24. E. Börger and D. Rosenzweig. WAM algebras – a mathematical study of implementation, Part 2. In A. Voronkov, editor, *Logic Programming*, volume 592 of *Lecture Notes in Artificial Intelligence*, pages 35–54. Springer-Verlag, 1992.

25. E. Börger and D. Rosenzweig. The WAM – definition and compiler correctness. In C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, volume 11 of *Studies in Computer Science and Artificial Intelligence*, chapter 2, pages 20–90. North-Holland, 1995.
26. E. Börger and R. F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.
27. B. Delaware, W. Cook, and D. Batory. Product lines of theorems. In *Proc. OOPSLA 2011*, Portland, October 2011.
28. P. J. Denning and C. Martell. Great principles of computing. http://cs.gmu.edu/cne/pjd/GP/GP-site/welcome.html (consulted July 26, 2011), 2007.
29. R. Farahbod et al. *The CoreASM Project*. http://www.coreasm.org.
30. A. Fleischmann. *Distributed Systems: Software Design and Implementation*. Springer-Verlag, 1994.
31. A. Fleischmann. Sbpm2NatLang converter. e-mail of September 8 to Egon Börger, 2011.
32. A. Fleischmann, W. Schmidt, C. Stary, S. Obermeier, and E. Börger. *Subjektorientiertes Prozessmanagement*. Hanser-Verlag, München, 2011. See [63] for a correct version of the appendix.
33. A. Fleischmann and C. Stary. Whom to talk to? A stakeholder perspective on business process development. *Universal Access in the Information Society*, pages 1–26, June 2011. DOI 10.1007/s10209-011-0236-x.
34. M. Frappier and H. Habrias, editors. *Software Specification Methods: An Overview Using a Case Study*. HERMES Sc. Publ., Paris, 2006.
35. N. G. Fruja. *Type Safety of C# and .NET CLR*. PhD thesis, ETH Zürich, 2006.
36. N. G. Fruja. Towards proving type safety of .net cil. *Science of Computer Programming*, 72(3):176–219, 2008.
37. N. G. Fruja and E. Börger. Modeling the .NET CLR Exception Handling Mechanism for a Mathematical Analysis. *Journal of Object Technology*, 5(3):5–34, 2006. Available at http://www.jot.fm/issues/issue_2006_04/article1.
38. Y. Gurevich. Reconsidering Turing's Thesis: Toward more realistic semantics of programs. Technical Report CRL-TR-36-84, EECS Department, University of Michigan, September 1984.
39. Y. Gurevich. A new thesis. *Abstracts, American Mathematical Society*, 6(4):317, August 1985.
40. Y. Gurevich. Evolving algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
41. Y. Gurevich. Interactive algorithms 2005 with added appendix. In P. W. D. Goldin, S. A. Smolka, editor, *Interactive Computation: The New Paradigm*, pages 165–182. Springer, 2006.
42. C. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
43. C. A. R. Hoare, J. Misra, G. T. Leavens, and N. Shankar. The verified software initiative: a manifesto. *ACM Computing Surveys*, 2009.
44. C. H. P. Kim, D. Batory, and S. Khurshid. Reducing combinatorics in testing product lines. In *Proc. Aspect Oriented Software Development Conference*. ACM, 2011.
45. D. Knuth. *Literate Programming*, volume 27 of *CSLI Lecture Notes*. Center for the Study of Language and Information, Stanford, California, 1992.
46. Metasonic. Metasonic-suite. www.metasonic.de/metasonic-suite.
47. R. Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Wpringer, 1980.

48. OMG. Business Process Model and Notation (BPMN). http://www.omg.org/spec/BPMN/2.0, 2011. formal/2011-01-03.
49. D. L. Parnas and J. Madey. Functional documents for computer systems. *Sci. of Comput. Program.*, 25:41–62, 1995.
50. G. Schellhorn. *Verifikation abstrakter Zustandsmaschinen*. PhD thesis, Universität Ulm, Germany, 1999.
51. G. Schellhorn. Verification of ASM refinements using generalized forward simulation. *J. Universal Computer Science*, 7(11):952–979, 2001.
52. G. Schellhorn. ASM refinement and generalizations of forward simulation in data refinement: A comparison. *Theoretical Computer Science*, 336(2-3):403–436, 2005.
53. G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM case study. *J. Universal Computer Science*, 3(4):377–413, 1997.
54. G. Schellhorn and W. Ahrendt. The WAM case study: Verifying compiler correctness for Prolog with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction – A Basis for Applications*, volume III: Applications, pages 165–194. Kluwer Academic Publishers, 1998.
55. Semiconductor Industry Assoc. International technologoy roadmap for semiconductors. Design. http://www.itrs.net/Links/2005ITRS/Design2005.pdf, 2005.
56. S. H. Sneed. Exporting natural language: Generating NL sentences out of S-BPM process models. In A. Fleischmann, W. Schmidt, R. Singer, and D. Seese, editors, *Subject-Oriented Business Process Management*, volume 138 of *Communications in Computer and Information Science*, pages 163–179. Springer, 2011. Second International Conference, S-BPM ONE 2010, Karlsruhe October 2010.
57. R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001.
58. A. Tarski. Der Wahrheitsbegriff in den formalisierten Sprachen. *Studia Philosophica*, 1(261-405), 1936.
59. A. ter Hofstede, W. van der Aalst, M. Adams, and N. Russell, editors. *Modern Business Process Automation*. Springer, 2010.
60. E. Uzuncaova, S. Khurshid, and D. Batory. Incremental test generation for software product lines. *IEEE Transactions on Software Engineering*, 36(3):309322, 2011.
61. W. van der Aalst and A. ter Hofstede. Workflow patterns home page. http://www.workflowpatterns.com/, created and maintained since 1999.
62. N. Wirth. *Algorithms & Data Structures*. Prentice-Hall, 1975.
63. Here the file for the correct text of the appendix of [32] can be downloaded.

    `http://www.hanser.de/buch.asp?isbn=978-3-446-42707-5&area=Wirtschaft,`
    `http://www.di.unipi.it/~boerger/Papers/SbpmBookAppendix.pdf`

    .