High Level System Design and Analysis using Abstract State Machines

Egon Börger

Università di Pisa, Dipartimento di Informatica, Corso Italia 40, I-56125 Pisa, Italy, boerger@di.unipi.it

Abstract. We provide an introduction to a practical method for rigorous system development which has been used successfully, under industrial constraints, for design and analysis of complex hardware/software systems. The method allows one to start system development with a trustworthy high level system specification and to link such a "ground model" in a well documented and inspectable way through intermediate design steps to its implementation. The method enhances traditional operational modelling and analysis techniques by incorporating the most general abstraction, decomposition and refinement mechanisms which have become available through Gurevich's Abstract State Machines. Through its versatility the ASM approach is non-monolithic and integratable at any development level into current design and analysis environments. We also collect experimental evidence for the ASM thesis, a generalization of Turing's thesis.

1 Introduction

In [21] the methodological and epistemological reasons are explained why Gurevich's concept of Abstract State Machines (ASMs) allowed us to develop a mathematically well founded approach to software and hardware design and analysis which is nevertheless practical, improving system design practice by a piece of useful theory. Due to the progress the method and its industrial applications have made since then, some of the predictions and hopes expressed in [21] have become reality. It is therefore with pleasure that I accept the invitation to write a general introduction to the approach.

The first section contains the definition of ASMs and an illustration of the main characteristics of the method, namely the most general *abstraction mechanism* it offers together with the corresponding *refinement* technique for vertical structuring of systems and the *decomposition* technique for horizontal structuring. We define the *function classification* which the applications have led us to introduce into ASMs as practical support for abstraction, refinement and (de)composition in concise high-level modelling of large systems. We show how ASMs combine abstraction with rigour—circumventing the omnipresent "formal system straitjacket"—and how their versatility permits one to integrate

[0] Part of the material appearing here has been presented to the ASM Workshop held in Magdeburg, September 21-22, 1998, as part of the Annual GI-Meeting.

them into existing development environments and to apply to them established analysis (validation and verification) techniques. Since this holds also at high hardware or software design levels, ASMs help to avoid and to detect errors in the early stages of development, where reasoning, verification, simulation and testing are less expensive than in the later stages. In the second section we show that ASMs encompass in a natural way other well known approaches to system design and collect experimental evidence for Gurevich's ASM Thesis [76] which strengthens the Church-Turing thesis. Concluding we point to some directions for future research and applications.

2 Characteristics of the ASM Method

Gurevich [71–74] discovered the notion of ASM in an attempt to sharpen the Church-Turing thesis by considerations from complexity theory (bounded resources) and abstract data structures. I tried out ASMs in an attempt to model in a rigorous but transparent way the dynamic semantics of the programming language PROLOG (see [18, 19]). This led to the ISO standard definition of the entire language [22, 39, 23] and became the starting point for the definition of a proven to be correct scheme for the implementation of Prolog on the Warren Abstract Machine [40]. Through this work and its various extensions [15, 16, 41, 34, 37, 42] we realized that on the basis of the most general abstraction mechanism inherent in ASMs, one can build a practical design method which can solve the fundamental triple problem of system development, namely

- to elaborate the informally presented requirements of the desired system turning them into a satisfactory *ground model*, i.e. a functionally complete but abstract description of sufficient but not more than necessary rigor which a) can be read and understood by and justified to the customer as solving his problem, b) defines every system feature as far as this is semantically relevant for the work the user expects the system to achieve, c) contains only what the logic of the problem requires for the system behavior, i.e. does not rely upon any further design decision belonging to the system implementation,
- to implement the ground model in a reliable manner by *refining* it step by step, in a controlled and well documented way, *through a hierarchy of intermediate models* each of which reflects some major design decision(s),
- to *structure the system horizontally* (from the ground model specification through the entire design process) by building it from components with abstract but sufficiently detailed definitions of the behavior of each single component and of the interaction of the components through interfaces.

This section contains the definition of ASMs and an illustration of this method through modelling examples taken from the literature. (For a detailed analysis of the ASM literature see [33] and [2].)

2.1 Abstraction Mechanism and Definition of ASMs

It has become a common place among computer science theoreticians and practitioners that most general abstraction principles are needed for coping with the complexity of present day hardware and software systems (for an illuminating early survey see [119]). The disagreement and the problems with concrete realizations of this maxime begin with the definition of abstraction.

On the one hand the available theories for composing abstract data or actions (e.g. the sophisticated algebraic specification [133] and action semantics theories [99]) are not practical for descriptions of complex dynamic real-life systems. They make it difficult to follow the guidance of a given problem for adequately choosing and formulating the data and the actions, either in combination or independently of each other, at the abstraction level which is appropriate for the application under study. On the other hand even the most advanced among the currently available practical approaches suffer from serious limitations. They tend to restrict the abstractions provided for data and operations (typically offering basic manipulations of sets, sequences, mappings, etc., as for example in the VDM approach [60]) or to impose particular forms for defining action abstractions (like the syntactic definition of atomic actions in the B-Method [3] by means of substitutions). Such restrictions typically reflect design decisions which have been made a priori for a fixed (conceptual or tool based) framework for refining specifications to executable implementations. By letting such implementation concerns creep into the fundamentals of specification, a heavy tribute is paid to the particular solution, in the underlying framework or tool environment, of the representation problem for abstract data or actions.

Abstract States. Using ASMs one can (and is advised to) separate the high level specification concerns from concerns related to further design, without creating a gap between specification and design. ASMs allow us to produce rigorous high-level definitions which reflect in a direct (i.e. coding free) way the application domain knowledge and the connotations of the problem as it appears to the application domain expert. One can tune the formulation of the data and operations in terms of the concepts (objects and actions) which appear in the problem domain, freed from having to think about how to represent them through a priori fixed schemes for defining data types and actions. Technically this *freedom of abstraction*—meaning freedom of listening to the subject, thinking semantically in terms of objects and effects of actions and not in terms of their representations or schemes of definition—is realized by the concept of machine state and of state transforming machine instructions.

An ASM state is not an indivisible entity like the control state of a finite automaton, it is not a mere set (a collection of values) and also not a function (an association of values to variables); it is an arbitrarily complex (or simple) structure in the general sense the term is used in mathematics and has been defined by Tarski [127], i.e. a collection of domains (sets, also called universes, each of which stands for a particular type of objects) with arbitrary functions and relations defined on them. What a machine "knows" about the elements of its domains, and about the functions and predicates it uses for manipulating them, depends on what is there in the particular application for which the machine has been defined. A universe can be completely abstract, meaning that no restriction is imposed on (and therefore no knowledge is available yet about) its elements, in particular not on their possible representation in our language or system. If the domain elements are assumed to have certain properties or to be in certain relations with other objects or to be subject to certain manipulations, we have to formulate these by corresponding operations, predicates or by conditions (integrity constraints) which the elements are required to satisfy.

As will become clear below, the ASM approach does not prescribe any particular notation to formulate constraints on machine states, whether verbally or formally, in programming, algorithmic, mathematical or any other rigorous or if necessary formal language. This has a pragmatically important effect on the integration potential of the approach (see section 2.5). Obviously the overall rigor of a description depends among other things on the degree of precision which is chosen for expressing such integrity constraints.

Before we illustrate the use of abstract machine states by four examples, let us say a word to avoid a possible misunderstanding which may come from the different use of the term "function" in mathematics and in programming. What we refer to is the mathematical meaning of the word function as a set of $n + 1$-tuples with a uniqueness property (the functional dependence of the $n + 1$-th element of a tuple from the first n elements, its arguments). Such a function corresponds to an n-dimensional array of programming. In particular nullary functions are the usual "programming variables", in the object-oriented spirit "class variables" which do not depend on the class members, whereas a monadic function $f : A \to B$ can be thought of as "instance variable" belonging to the class A, to be instantiated for each class member.

Computers hosting PVM Processes. Imagine you want to analyse the PVM [63] communication mechanism between application programs which run concurrently on physically interconnected heterogeneous machines of various architectures (serial, parallel, vector computers). Without going into irrelevant representation details we can describe the role of the constituting member computers by declaring them to be elements of a dynamically changing set *HOST* which comes with three informations. A 0-ary function (class variable)

$$master : HOST$$

describes a designated host machine on which PVM is started and which plays a special role for maintaining the control over the dynamically changing PVM configuration. Another function

$$arch : HOST \to ARCH$$

defines for each host machine its architecture which belongs to an abstract domain *ARCH* (definable as the set of possible architectures to be used with PVM 3 as listed in [63]). A third function

$$daemon : HOST \to DAEMON$$

dynamically provides each host machine with an abstract daemon process which acts as a PVM supervisor for the local management of application programs (which are enrolled as processes into PVM) or for the communication between such processes (which may reside on different host machines). For the initialization a 0-ary function

$$demiurge : DAEMON$$

provides a distinguished daemon who resides on the master host (expressed by $daemon(master) = demiurge$) and is responsible in particular for creation and deletion of hosts and maintenance of the corresponding communication structure. These abstract domains and functions are the basis for the ASM model of PVM in [28, 29]. No further information is needed about what hosts, daemons, architectures and processes "are" or how they could be "represented".

Avoiding premature object representations. Working with abstract domains and functions allows us to avoid premature object representation decisions. In a practical modelling approach these *should* be avoided[1] because they typically enforce dealing with mere formalism (or performance) problems which carry no conceptual content for the problem under investigation and deviate the attention from it. This can be illustrated by the modelling example presented in the chapter on "Constructing a Model" in [60] "to provide some initial guidance on how one can start developing formal models using VDM-SL". A call-out mechanism has to be designed for a chemical plant alarm system where experts with different kinds of qualification must be called for coping with different kinds of alarm. The VDM-SL model represents experts by a record type with an expert identifier and a qualification field. It is explained (op.cit.p.18) that "the representation of ExpertID is important in the final implementation, but in this model none of the functions need to be concerned with its final representation. All that is required at this level of abstraction is for the identifiers to be values that can be compared for equality". As a consequence the authors propose *ExpertId=token* as "the completed type definition in VDM-SL", adding that "In VDM-SL, the special type representation called *token* is used to indicate that we are not concerned with the detailed representation of values, and that we only require a type which consists of some collection of values." All this does not prevent the authors from having to introduce, see op.cit. page 24, yet another condition in order "to ensure that ... one could not erroneously have two experts with different qualifications having the same expert identification". In an ASM model for the alarm system these complications disappear by treating experts abstractly as elements of a set $EXPERT$ which comes with a function

$$quali : EXPERT \to QUALIFICATION$$

associating qualifications to experts. The doubling of experts and of expert identifiers (by the way also the introduction of a special type representation *token*) are avoided and the expert identification problem (which does interest our customer) is solved free of charge by the underlying classical logic of equality applied

[1] "Data in the first instance represent abstractions from real phenomena and are preferably formulated as abstract structures not necessarily realized in common programming languages."[135, p.10]

to elements of sets. Certainly one has to guarantee an appropriate internal representation of experts in the implementation (why not by records), but this is a problem which does not concern the customer any more and which is solved when the specification is refined to excutable code (see the discussion of refinement techniques in section 2.3).

Annotated program syntax trees. Typical usage of abstract domains and functions can be illustrated also through the ASM modelling of the dynamic semantics of various real-life programming languages like PROLOG [39], OCCAM [27], VHDL [30, 31], JAVA [43] and many others [33]. Here the problem consists in expressing the static language concepts (objects/operations) and the dynamic program actions (effect of basic instructions) faithfully, the way they appear in the standard or language reference manual, without introducing any extraneous encoding due to an a priori fixed abstraction level and avoiding in particular any a priori imposed static representation of actions-in-time. In [27] we expressed the canonical (syntactically defined) part of the sequential control by a graph[2], i.e. a statically defined successor function on an abstract domain *NODE*. The nodes are decorated by a function

$$cmd : NODE \to INSTRUCTION$$

with atomic commands to be executed. This circumvents the typical encodings with terms (generated by the underlying context free grammar) or with continuations (typically appearing in denotational descriptions) or tables and the like and allows one to formulate the meaning of instructions locally (providing a modular definition of semantics) and such that it directly reflects the relevant language design decisions. In this way the ASM model for Occam [27] lets the dynamics of the distributed language features stand out explicitly in rules which govern the local execution of program instructions by daemons. The daemons are elements of an abstract dynamic set *DAEMON*; they are created and deleted and, carrying along their private environment, walk through the graph (updating a positioning function *loc*). They move in a truly concurrent way, independently from each other (unless they synchronize explicitly by communication), each at its own pace, with its own notion of time. This model lent itself to an implementation by Transputer code [26] where the abstract daemons are implemented as workspace addresses. The workspace encodes, reflecting the Transputer layout for optimized use of memory, all the abstract functions which we introduced for a transparent definition of the semantics of Occam programs. We have used a similar approach for dealing with threads in a platform independent high-level definition of the dynamic semantics of Java [43]. In [78] this locality principle is pushed further to obtain a provably correct and reusable compilation scheme, from source to intermediate code, by orthogonal specification and verification of the semantics and of the translation of independent language concepts. As

[2] Later I discovered that a similar idea has been used before in numerous places in the literature, the earliest reference I found is [51] where expression evaluation is described by a processor which "moves through the ... expression tree making local transformations in it".

with attribute grammars, abstract syntax tree nodes, representing a syntactical category, are used to describe the semantics and the compilation of the involved language construct, together with the relevant context (including the data and control flow); this is enhanced by introducing generic (parameterized) ASMs to define frequently occurring and reusable transformations.

ISO PROLOG state definition. This example is taken from the ASM definition for the ISO standard of the dynamic semantics of Prolog [22, 23, 39]. The members of the ISO Prolog standardization committee wanted to see Prolog computations as depth-first left-to-right tree traversal, in search of possible solutions to the initially given query using the SLD resolution mechanism. We described therefore Prolog computation states as elements of an abstract set *NODE* coming with a static 0-ary function *root*, a 0-ary dynamic function (i.e. a function whose value will change during the computation) *currnode* for positioning the current node and a function

$$father : NODE - \{root\} \ \rightarrow \ NODE$$

which keeps track of the originator of the given step. We had to enrich this tree structure by the information each element of *NODE* has to carry for the Prolog computation state it represents, namely the sequence of goals still to be executed, the substitution computed so far, and the sequence of alternative states still to be tried. This was accomplished by the introduction of abstract domains of Prolog literals and goals (sequences of literals), of substitutions and (for a high-level description of the effect of executing the Prolog cut operator) of a set of goals "decorated" by their cutpoint information:

$$LIT,\ GOAL = LIT^*,\ SUBST,\ DECGOAL = GOAL \times NODE.$$

The goal and substitution information is associated to nodes by dynamic functions *decglseq*, *s* defined on *NODE*. For keeping track of the sequence of alternative states still to be tried, it sufficed to associate to each node an element of an abstract domain *CODE* of instruction lines containing each an occurrence of a *clause* which can be retrieved by an abstract *clause-line* function:

$$cll : NODE \ \rightarrow \ CODE, \quad clause : CODE \ \rightarrow \ CLAUSE.$$

For setting *cll* dynamically (see Figure 1), an auxiliary abstract function

$$procdef : LIT \times PROGRAM \ \rightarrow \ CODE^*$$

sufficed of which we assumed to yield the properly ordered list of code, in logic programming vernacular the candidate clause lines for the given literal in the given program. (In object-oriented terminology *procdef* could be interpreted as an array variable of the class *PROGRAM*.) Be aware that a considerable part of the standard virtual machine for implementing Prolog, Warren's Abstract

Machine, is devoted to implementing this function and the representation of terms and substitutions efficiently.

This is all one needs to know about the state (the structure defined by the abstract sets and functions) of the ASM which defines the ISO standard for the user-defined kernel of Prolog (see the details at the end of 2.1).

Abstract Instructions for Changing States. Up to now we have explained half of the freedom of abstraction offered by ASMs, namely the freedom to choose the states and thereby their signature, i.e. the number of the universes and of the operations together with their arity, domains and ranges, and the integrity constraints. The other half is the freedom to choose the machine instructions in order to reflect explicitly the actions which in the desired abstraction level are to be designed as basic dynamic actions. Through the signature we can determine which operations will be considered as given and atomic (either static or belonging to the environment), whereas the instructions allow us to determine which operations are decomposed into (multiple) basic machine steps.

What is the most general form of machine instructions to transform structures? For the sake of simplicity of exposition assume that predicates (properties and relations) are treated as characteristic (i.e. Boolean valued) functions. If the signature remains fixed, there is only one thing one can do to change such a structure[3], namely change under certain conditions the value of some functions for some arguments (usually not for all arguments and not for all functions, see the discussion of the locality principle below). Therefore the most general structure transforming machine instructions (called ASM *rules*) are guarded destructive assignments to functions at given arguments, expressable in the following form:

$$\textbf{if } \mathit{Cond} \textbf{ then } \mathit{Updates}.$$

Cond is an arbitrary condition (statement) formulated in the given signature, *Updates* consists of finitely many *function updates*:

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

which are executed simultaneously. f is an arbitrary n–ary function, $t_1, \ldots, t_n$ are arguments at which the value of the function is set to t. We thus arrive at Gurevich's definition.

Definition of ASMs. An ASM M is a finite set of rules for guarded multiple function updates, as defined above. Applying one step of M to a state (algebra) $\mathcal{A}$ produces as next state another algebra $\mathcal{A}'$, of the same signature, obtained as follows. First evaluate in $\mathcal{A}$, using the standard interpretation of classical logic, all the guards of all the rules of M. Then compute in $\mathcal{A}$, for each of the rules of M whose guard evaluates to true, all the arguments and all the values appearing in the updates of this rule. Finally replace, simultaneously for each rule and for all the locations in question, the previous $\mathcal{A}$-function value by the newly

[3] structures without predicates, i.e. consisting only of functions, are often called algebras

computed value (if this is possible, i.e. if no two required updates contradict each other). The algebra $\mathcal{A}'$ thus obtained differs from $\mathcal{A}$ by the new values for those functions at those arguments where the values are updated by a rule of M which could fire in $\mathcal{A}$. The effect of an ASM M started in an arbitrary state $\mathcal{A}$ is to apply one step of M as long as possible (until no M-rule can fire any more because all the guards of M-rules have become false).

It may sound unbelievable, but the preceding definition is all the practitioner has to know about the semantics of ASMs in order to be able to work with them. The definition uses only basic mathematical concepts, no further theory is needed. It is adequate to use ASMs as abstract (conceptual) code; we avoid here the word pseudo-code because through Gurevich's more detailed definition [75] the abstract code offered by these machines is semantically well founded and rigorous enough to be applicable by practitioners and to be implementable, as is, by various support tools, unambiguously, without leaving room for faithful interpretations to differ between the tools or between the practitioners, whereas pseudo-code has never enjoyed this property. The at first sight astonishing simplicity of the semantics of ASMs is in part due to the fact that the only programming construct offered for free is the guarding of function updates. This explains why also application domain experts with only basic understanding of algorithmic processes can read and understand such specifications without any problem. When the system design specialist has an advantage to use more complex programming constructs (sequencing, loop, recursion, procedures, etc.), he can certainly do so, defining them in the expected way for ASMs but at the price of further semantical complications inherent in these concepts. ASMs provide not a programming, but a design language which supports linking specifications to code in a controlled (well understood and well documented) manner.

Certainly there is something more which can be said about how to use ASMs, belonging however less to their theoretical foundation than to integrating useful techniques from current system design practice. We obviously make free use of all kinds of standard programming notation (nested if-then-else, case, pattern matching, let, where, etc.) which can be reduced in a canonical way to the above basic definition. But unless there is an important reason which pays for it, we avoid the use of any more complex or non-standard concept or notation in order to keep the machines transparent and easy to follow. Usually one is only interested in states which are reachable, in the machine in question, from a given collection of initial states; the ASM approach leaves it completely open how rigorously they are defined and also imposes no particular definition language or method. Concentrating on explaining the abstraction mechanism, we have phrased the discussion in terms of one-agent or *sequential ASMs* which suffice for the purpose and by the way cover already much of practical system design (see for example how far one can go with Abrial's B-Method [3] or with the Vienna Development Method [60] which both were developed for modelling, analysing and designing exclusively sequential software systems). We discuss the extension to multi-agent (distributed) ASMs—very much needed for present-day applications—in see section 2.4., the detailed definition appears in [75].

The parallel execution model for sequential ASMs, realized by the simultaneous execution of all updates in all rules which can fire, turned out to be particularly useful in many applications. An example par excellence is provided by specifications of synchronous systems where each global clock tick is directly reflected by the application of a single machine step (which may consist in the firing of many rules); this has been used with adavantage for modelling pipelining in RISC architectures [35, 85]. The parallelism incorporated into sequential ASMs also helps to specify macro steps where one wants to hide the intended implementation by multiple lower level micro steps. It also helps to avoid premature sequentialization in system design. This is useful when the sequential execution is either not required by the logic of the problem under study or even hinders an easy to analyse high-level specification of the problem solution; in the design of a metal-processing plant [36] the abstraction, in the ground model, from a certain sequential use of the seven components of the system not only simplified the system description, but also allowed us to establish by a simple argument the strongest form of liveness which the problem formulation required to guarantee; only in the transformation of the final model to C++ code [95] became it necessary (and easy) to sequentialize the rules of the different agents. Abstraction from the sequentialization can also be crucial for achieving a modular system design when the sequential scheduling problem is difficult enough in itself to be better treated separately from the rest of the system behavior. An example is the sequential implementation of the parallelism of Occam programs in Transputer code [26]: the sequence of ASM models which links, in a proven to be correct way, Occam programs as understood by the programmer to their compilation to Transputer code, deals with the sequentialization problem separately from the Transputer details; in this case at a rather high system level where it is easy to link, in a correct and controlled way, parallel executing processes to abstract priority queues of processes managed on the basis of a time-slicing procedure.

Locality Principle. ASMs typically describe local changes (of a few functions for selected arguments appearing in the rules), the way it is needed to naturally reflect the overall behaviour of large systems as determined by local actions of some components. The computation model guarantees that in every step, everything not affected by the updates in the rules fired in this step remains unchanged. This frees the designer from having to worry about the well known frame problem; often the frame axioms are responsible for the combinatorial explosion of formal descriptions relying upon a logical or axiomatic system (or otherwise global transformation) view, as for example in Z [48], temporal logic or denotational [98] approaches. Exploiting a locality property often leads to modular design and supports stepwise refinement by offering a safe way to concentrate on the parameters which do change while neglecting the others. In contrast to a widely held view not only the system design, but also the mathematical reasoning about the system behaviour can become considerably simpler if the locality principle is used.

Consistency of updates is a problem the system designer has to care about. Obviously an execution tool should (be able to) tell, at the latest at runtime, if and when an inconsistency does occur, as is the case in [54]. Since in general the consistency problem is undecidable, as long as we work in the high-level design area where a truly general specification method is needed, there is no other choice than to be conscious about and to try to avoid the problem by transparent design. More we move towards lower levels, more chances we have to trade expressability for decidability and for error detection by tools.

Nondeterminism can be reflected in ASMs in various ways. One can modify the basic semantical definition by firing in each step one (or some) of all firable updates or rules, which yields also a standard way to hide the consistency problem; this was the solution chosen in [74]. If one prefers to let the nondeterministic choice stand out explicitly as appearing in the signature, it can be phrased using appropriately specified selection functions. One can also express the nondeterminism using the so called qualified choose construct, introduced into ASMs by rules of the following form (with the obvious meaning) where *Rule* is an arbitrary ASM rule, D any domain of the given signature and α an arbitrary (the qualifying) condition on the choice:

$$\begin{array}{l}\textbf{Choose}\ \ x\ \textbf{in}\ D\ \ \textbf{s.t.}\ \alpha(x)\\ \qquad\qquad \textit{Rule}\end{array}$$

Given an ASM M, i.e. a set of rules, we can express its nondeterministic variant $N(M)$ by the following ASM:

$$\begin{array}{l}\textbf{Choose}\ \ \textit{Rule}\ \textbf{in}\ M\ \ \textbf{s.t.}\ \textit{firable(Rule)}\\ \qquad\quad \textit{Rule}\end{array}$$

Fine tuning without formal overkill. The following real-life example is taken from an ISO standardization endeavour and illustrates fine tuning of models to the level of abstraction adequate for their intended use, meeting the desired rigor and conciseness without having to pay a tribute to formal overkill.

The example defines the kernel of the ISO standard of Prolog [22, 23, 39] dealing with the so called user-defined predicates P of a given program. Predicates of logic programming represent procedures which are invoqued by basic Prolog statements $P(s_1, \ldots, s_m)$, called *literals*. They are computed (in logic programming vernacular: checked to be satisfiable for the arguments $s_1, \ldots, s_n$) by trying to execute at least one of the "instructions" (called *clauses*) $c_1, \ldots, c_n$ which define the procedure code, in the order they appear in the current program *prog*. The fundamental abstraction we had to provide for the ISO definition was a mechanism to extract from the current program, for the currently examined literal (the activator *act*), this properly ordered list of candidate clauses without mentioning any detail for the implementation of this mechanism (and of the related representation of terms, substitutions, literals and clauses in the WAM). The problem is not as innocuous as it appears because Prolog programs are dynamic, modifiable at runtime. Nevertheless it is easily solved by the dynamic function *procdef* introduced above whose behavior is determined referring only to the current activator *act* and the current program (and indirectly also to the dynamic abstract association *clause* of *clauses* to lines of *code*).

Using *procdef*, the entire Prolog kernel for user-defined predicates is easily defined by adaptating, to the Prolog environment, the depth-first left-to-right tree creation and traversal, see Figure 1. The *Success* rule terminates the computation when the current goal sequence has been emptied. The *Continuation* rule makes the computation *proceed*, upon successful computation of the current procedure body (read: satisfaction of all the literals of the leading *goal* attached to the current node), by updating the goal sequence. The *Call* rule calls the procedure for the currently computed literal *act* by attaching to the current node the sequence of candidate nodes for the possible procedure body subcomputation trees, one tree root t_i for each alternative clause c_i found for the activator in the current program, and passes control to the *Select* rule for trying these children in the indicated order for execution (read: logic programming resolution).

Success

```
if $decglseq(currnode) = [\,]$
then $stop := Success$
```

Continuation

```
if $goal(currnode) = [\,]$
then $proceed$
```

Call

```
if $is\_user\_defined(act)$
  & $mode = Call$
then extend $NODE$ by
$t_1, \ldots, t_n$ with
 $father(t_i) := currnode$
 $cll(t_i) := c_i$
 $cands(currnode) := [t_1, \ldots, t_n]$
 $mode := Select$
where $[c_1, \ldots, c_n]$
   $= procdef(act, prog)$
```

Select

```
if $is\_user\_defined(act)$
  & $mode = Select$
thenif $cands(currnode) = [\,]$
then $backtrack$
else $resolve$
where $backtrack \equiv$
  if $father(currnode) = root$
  then $stop := Failure$
  else $currnode := father(currnode)$
       $mode := Select$
```

Fig. 1. ISO Prolog kernel (for user-defined predicates)

The function *goal* yielding the leading goal of a Prolog state is a derived function, i.e. a function which can be defined in terms of other functions, here explicitly in terms of *decglseq* and of list functions. Similarly for *act* where the definition incorporates the scheduling optimization for determining which one of the literals in the leading goal is selected for the next execution step. Even at the level of abstraction we are considering here, namely of the semantics of the language, the definition of the abstract action *resolve* is a matter of further refinement. It provides the really logical ingredients of Prolog computation steps, involving the logical structure of terms and the logical concepts of substitution and unification. Similarly the definition of *proceed* provides further insight into the tree pruning and the error handling mechanisms which enter ISO Prolog through the extra-logical built-in predicates *cut* and *throw* (see[22, 39] for details).

The four rules of Figure 1 define the abstractions which turned out to be appropriate to clarify the database update problem which has intrigued for a long time the ISO standardization effort [25, 38].

2.2 Building Ground Models

In the introduction to [3] Jean-Raymond Abrial states "that the precise mathematical definition of what a program does must be present at the *origin* of its construction" with the understanding that the concept of program incorporates that of specification. For epistemological reasons the most difficult and by statistical evidence[4] the most error prone part of the thus understood program construction is at the origin because it is there that we have to relate the part of the real-world under study to the models we have for it in our language (representing the models in our mind). Since there is no infinite reduction chain between models, as discussed already by Aristotle[5] criticising Plato [8], this link itself cannot be justified theoretically, by mere logico-mathematical means of definition or proof. But still, in order to gain confidence in the models which are the basis for the entire program construction leading to executable code, we must justify their appropriateness by connecting somehow their basic objects and operations to the basic entities and actions we observe in the real world.

Pragmatic foundation is the best we can achieve, exploiting conceptual and experimental features of the models. A conceptual justification can be given by grasping the adequacy of the *ground* model [6] with respect to the informal description of the world, through direct comparison. This requires that the ground model is tailored at the level of abstraction at which we conceive the part of the real world to be modelled, i.e. in such a way that its mathematical objects, predicates, functions, transformations correspond in a simple way, possibly one-to-one, to the entities, properties, relations, processes appearing in the informal "system requirements" to be captured. An experimental justification can be provided for the ground model by accompanying it with clearly stated system test and verification conditions (system acceptance plan), i.e. falsifiability criteria in the Popperian sense [109] which lay the ground for objectively analyzable and repeatable experiments. This too requires an easily inspectable, comprehensible and transparent link between the formal ground model and the informal problem description, the outcome of the experiments having to be confronted

[4] 80% of the errors in system programming do occur at the level of requirement specifications.

[5] "Every theory and in general every deductive knowledge has a certain wisdom as premise."[6]

[6] In [20, 39] they were called *primary models* to stress that they are not in any way absolute or unique but simply starting points for a series of mathematical transformations.

with the informally described real-world situation (see below the discussion of the oracle problem of testing and more generally of validating high-level models by simulation). In comparison to the conceptual and experimental justification of a ground model, the mathematical justification of its internal consistency is the smaller although not negligible problem, essentially a problem of high-level reasoning and (where possible machine assisted) proof checking.

Finding the right abstraction level is the main problem the definition of appropriate ground models has to face. One must discern the aspects of the desired system which have to be included in the mathematical model, for its being semantically complete, satisfying both the customer and the designer, and relegate the ones not relevant for the logic of the problem to further design decisions, as belonging to the implementation. This choice can be made and justified appropriately only on the basis of the related application domain knowledge, typically in close cooperation between the system designer and the application domain expert who brings in the informal requirements. Usually these initial requirements are neither complete nor minimal and it needs engineering and conceptual skill to distill the relevant features, finding those which are lacking in the informal description (because implicitly assumed by the domain expert) and hiding the dispensable ones, reserving them for further refinement steps. The need for cooperation between the user and the designer to construct the ground model reveals yet another difficulty, namely to have a common language for sufficiently unambiguous communication between the two parties, as is confirmed by statistical data: two thirds of the software development time are spent for communication between user and designer; one quarter of software project failures is due to user/designer communication problems; mismatch in system requirements understanding between user and designer is recognized as the most frequent cause of user dissatisfaction with the final product.

ASMs solve the language problem, for communication between customer and designer, by using only common mathematical and algorithmic concepts and notation for the description of real world phenomena. They also contribute to satisfactory and practically viable solutions of the formalization problem. The freedom of abstraction allows the system designer to model the informal requirements by expressing them directly[7], in application domain terms, without having to think about formal encoding matters which are extraneous to the problem. In the VDM approach, before starting the modelling work, one first has to learn "the most basic kinds of data value availabe to the modeller and ... how values can be manipulated through operators and functions" [60, p.72] and is then forced to formalize everything in terms of these fixed data abstraction possibilities, inventing encodings if they don't fit directly. With ASMs the designer can freely choose the basic data values (objects of abstract domains) and their manipulations (through functions or updates), looking only at what is showing up in the application domain and in the problem explanation given by the cus-

[7] "A significant engineering project begins with a specification describing as directly as possible the observable properties and behaviour of the desired product".[81, p. 4]

tomer. Similarly the freedom to separate modelling from justification concerns and, when it comes to proving, to choose the appropriate level of proof, solves also the justification problem. Mathematical experience as well as experience with machine assisted proof systems show that for the purpose of proving, one often has to strengthen the claim and to provide more details than what appears in the property to be proved. Specification *tout court*, or coming with a high-level instead of a formalized and machine checked reasoning, also represents a form of abstraction. True, "when the proof is cumbersome, there are serious chances that the program will be too" [3, p.XI]; but equally well too many details imposed by the proof system may unnecessarily complicate the design.

Thus an ASM ground model has the chance to serve equally well the customer and the designer. It can be understandable for the customer who typically is not thinking in terms of data representation and usually is not trained to understand proofs. The designer can show to his team of software experts that it is consistent and satisfies the required high-level system properties. In addition it can be sufficiently rigorous and complete for the design team to proceed with mathematical analysis and transformation into an implementation.

Supporting practical design activity. ASMs enable the designer to make his intuitive ideas explicit by turning them into a rigorous definition which, far from being an "add-on", documents the result of his "normal" activity and thereby provides the possibility of an objective assessment of the properties of his design. The abstraction freedom does not provide the intuitions needed for building a good ground model, but it permits to smoothly accompany their natural formulation, turning them into a concise definition, with an amount of writing and notation proportional to the complexity of the task to be specified; differently from most other formal methods, ASMs enable us not to complicate matters by formal overhead, simply because no (linguistic or computational) modelling restriction forces us to include (encoding or scheduling) details which belong only to the specification framework and not to the problem to be solved.

As with every freedom, the abstraction freedom puts on the designer the full responsibility for justifying, to his team of programmers and to the application domain expert, the choices made for the basic data (domains with predicates and functions) and actions (rules). This leads him naturally to state and collect along the way all the assumptions made either by the customer, for the system to work as desired, or by the designer, for laying out the software architecture in the ground model. This helps not to depend, at least not inadvertedly, upon any bias to particular implementation schemes and has the important practical effect to make relevant application domain knowledge explicitly available in the ground model documentation.

These features of ground model development can be illustrated by two examples in the literature [36, 17] where an informal requirement specification has been turned into a ground model and implemented, going through stepwise refined models, by C++ code which has been validated through extensive experimentation with the provided simulators. The robot control example [36] exploits the ground model abstractions for guaranteeing, in a straightforward manner, all the required safety properties, under explicitly stated high-level assumptions which could easily be established to be true for the subsequent refinements; similarly the ground model abstractions allowed us to establish the strongest form of the required liveness (that "every blank inserted into the system will eventually have been forged") and the maximal performance property for the system. All these properties could be verified by a standard analysis of the ASM runs, carried out in terms of traditional mathematical arguments and later also mechanically confirmed by a model checking translation in [132]. In both examples the ground model refinements have been developed up to a point from where the executable C++ code could be obtained through an almost mechanical translation of ASM rules into C++ procedures which are executed in a context of basic routines implementing the ASM semantics. This illustrates the role of intermediate models for documenting the structure of the executable code as well as the design decisions which have led to it; these formally inspectable intermediate models can serve as starting point for possible modifications coming up in the code maintenance process (optimizations, extensions, etc.).

The freedom of linguistic and computational abstraction makes ASMs also adaptable to different application areas and yields easily modifiable and in particular extendable models. See for example the ease and naturalness with which the models for Prolog and its implementation on the Warren Abstract Machine [39, 40] could be extended post festam for including polymorphic types or constraints and their implementation on corresponding WAM extensions [15, 16, 41, 42], see also the WAM extension by parallel execution with distributed memory in [5] and the adaptation to the implementation of scoping of procedure definitions in a sublanguage of Lambda-Prolog where implications are allowed in the goals [90]. Some students in my specification methods course in Pisa in the Fall of 1998 have produced an almost straightforward after the fact extension of the production cell ASM in [36] to the fault-tolerant and to the flexible real time production cells proposed in [93, 94], another group has adapted the high-level steam-boiler ASM [17] for a modelling of the emergency closure system for the storm surge barrier in the Eastern Scheldt in the Netherlands [97].

2.3 Refinement Technique

It is nowadays a common place that software design has to be hierarchical and has to be based on techniques for crossing the abstraction levels encountered on the long way from the understanding of the problem to the validation of its final solution. There are numerous proposals for defining and relating these levels as support for the separation of different software development concerns: separating program design from its implementation, from its verification, from its domain dependence[11], from hardware/software-partitioning [91], system design from software design from coding and similarly for testing (Sommerville's V-model [121]), functionality from communication, etc. Numerous vertical structuring principles have been defined in terms of abstraction and refinement.

Semantical refinement is what ASMs add to this, due to their freedom of abstraction, refinement being the reverse of abstraction. Therefore the designer can follow the needs for vertical structuring and separation of concerns (including modularization and information hiding) as suggested by the semantics of the system to be developed, without being directed by any a priori fixed syntactical or computational boundary condition one would have to comply with. He is also not forced to first learn one of the many complicated but typically rather specialized and not easily applicable refinement theories in the literature. Since the abstraction mechanism incorporated into the definition of ASMs is as general as the present day scientific knowledge enables us to conceive (see section 3.), the corresponding ASM refinement method permits to push Dijkstra's [57] and Wirth's [134] refinement program to its most general consequences. ASMs allow us to realize it independently from the restrictions which necessarily come with every concrete programming language for executable code, producing maximal practical effect for hierarchical and ideally provably correct system design.

Various practical refinement notions have been developed by us, through real-life case studies, in an attempt to close, in a controllable manner, the gap between the design levels involved. They can all be put into the form of the well known commuting diagram of Figure 2, for a given machine A which is refined to a machine B, where a usually partial *abstraction function*[8] $\mathcal{F}$ serves as proof map, mapping certain refined states $\mathcal{B}$ of B to abstract states $\mathcal{F}(\mathcal{B})$ of A, and certain sequences R of B-rules to sequences $\mathcal{F}(R)$ of abstract A-rules (in cases where the proof map is used as *refinement function*, $\mathcal{F}$ goes from A to B).

$$\begin{tikzcd}
A \arrow[r, "\mathcal{F}(R)"] & A' \\
B \arrow[u, "\mathcal{F}"] \arrow[r, "R"'] & B' \arrow[u, "\mathcal{F}"']
\end{tikzcd}$$

Fig. 2. ASM refinement scheme

In order to establish the desired *equivalence* of the two machines, before proving the commutativity of the diagram, one can (and first of all has to) *define* the appropriate notions of *correctness* and/or *completeness* between refined runs $(\mathcal{B}, \mathcal{S})$ and abstract runs $(\mathcal{A}, \mathcal{R})$. This definition is in terms of the locations (the "observables") one wants to compare in the related states of the two machines. The observables could be, for example, the operations the user sees in the ab-

[8] Schellhorn [115] generalizes this to relations and provides examples where it is convenient to relax the abstraction and refinement functions to relations.

stract machine, which are implemented through the refinement step. This case is characteristic for the refinement concept of the B-method: "During each such step, the initial abstract machine is entirely reconstructed. It keeps, however, the same operations, as viewed by its users, although the corresponding pseudo-code is certainly modified" [3, p.XVI]. ASMs offer an a priori not restricted spectrum for instantiating the refinement notion, for fine tuning it each time to the underlying implementation idea; this allows one to concentrate on the main problem, namely to find and to describe in an understandable way the appropriate design idea, i.e. "the right $\mathcal{F}$", the function (or relation) which incorporates and documents the software engineering experience and implementation skill. The difference of the refinement concepts for B and ASMs is probably a consequence of the fact that in B, "in the intermediate refinement steps, we have a hybrid construct, which is not a mathematical model any more, but certainly not yet a programming module" [3, p.XVI], whereas the refined ASMs are certainly mathematical models, the same way as the abstract ones.

We call $\mathcal{F}$ a proof map because it guides the justification the designer provides for the semantical correctness of the refinement step. Proofwise the ASM refinement notions are supported, in their full generality, in PVS [58] and KIV [115]; Schellhorn defines sufficient conditions for correctness and/or completeness preserving composition of commutative refinement diagrams out of subdiagrams, which yield a modular scheme for provably correct refinements of finite or infinite runs. This confirms our experience that it helps if one provides many and easily controllable refinement steps instead of only a few but difficult ones; as a by-product of such refinement sequences one obtains a detailed documentation for each single design decision, a feature which supports design-for-reuse, easy extendability and reliable maintenance of the final product.

Numerous case studies illustrate the far reaching practical use of semantically oriented refinement chains, in so different domains as implementation of programming languages, architecture design, protocol verification and development of control software. The 12 intermediate models created in [40] to link in an easily justifiable way the ISO Prolog model to its implemementation on the WAM served to implement the following features: the backtracking structure (in a stack model with reuse of choicepoints), the predicate structure (introducing determinacy detection—a look-ahead optimization—and compiling the selection of alternatives by (re)try/trust code with switching), the clause structure (implementing continuations by environment (de)allocation and the clause compilation by unify/call code), the term and substitution structure (introducing the heap, implementing the unification, compiling clause heads/bodies by getting/putting instructions and variable binding) and WAM optimizations (environment trimming, last call optimization, local/unsafe variables, on-the-fly initialization). There is not a single step which has been suggested or guided by the syntax of Prolog programs. It turned out later that these intermediate models make it possible to extend without difficulty both the specification and the proof steps, from Prolog to Prolog with polymorphic types or constraints and their implementation on the corresponding WAM extensions PAM [15,16] and CLAM [41].

The refinement hierarchy in [27,26] led to a provably correct compilation scheme for Occam programs to Transputer code. It reflects standard compilation techniques, some peculiarities of the communication and parallelism concept of the language and some characteristic Transputer features. There was only a small contribution from program syntax to the following chain of 15 models for the implementation of the following features: channels, sequentialization of parallel processes (by two priority queues with time-slicing), program control structure, environment (mapped to memory blocks addressed by daemons), transputer datapath and workspace (stack and registers, workspace and environment size), relocatable code (relative instruction addressing and resolving labels), everything with a high-level notion of expression and expression evaluation which abstracts from the peculiarities of the Transputer expression evaluation.

The chain of over 15 models leading from Java through its compilation on the JVM to the full JVM [43,44,46] reflects two goals. One is related to the language and its implementation on the JVM, namely to orthogonalize sequential imperative (while program) features, static class features (procedural abstraction and module variables represented by class methods/initializers and class fields), object oriented features, the error handling mechanism and concurrency (threads). The other goal is related to the Java Virtual Machine as a Java independent platform, namely to separate trustful bytecode execution, bytecode verification and dynamic loading concerns in order to make their interaction transparent.

The equivalence notions defined for the commutative diagrams in such compilation scheme correctness proofs prove much more than the traditional equation

$$[P]_{source} \;=\; [compile(P)]_{target}$$

where the square brackets denote the denotational input/output program meaning. Typically this meaning is defined by a fixpoint solution to certain equations over abstract domains, following the syntactical structure of P. A bitter consequence is that applications of such domain based methods are usually restricted to relatively simple properties for small classes of programs; the literature is full of examples: *pure* versions of various functional programs (pure instead of common LISP programs [108]), Horn clauses or slight extensions thereof [123] instead of Prolog programs, structured WHILE programs [100] instead of imperative programs appearing in practice (for example Java programs with not at all harmful, restricted forms of go to). A remarkable recent exception is the use of denotational semantics for the functional-imperative language ComLisp in the Verifix project [131]. The add on of ASMs with respect to denotational methods is that properties and proofs can be phrased in terms of abstract runs, thus providing a mathematical framework for analyzing also runtime properties, e.g. the initialization of programs, optimizations, communication and concurrency (in particular scheduling) aspects, conditions which are imposed by implementation needs (for example concerning resource bounds), exception handling, etc.

The refinement step in [32] is completely unrelated to program structure. It is determined by the intention to separate the considerations one has to make for durative actions from the correctness concern for the mutual exlusion protocol with atomic actions. This algorithm design in two steps simplified considerably the analysis of Lamport's protocol in the literature. The five design levels introduced in [35] serve to separate and justify the major techniques involved in pipelining a typical RISC microprocessor, namely parallelization and elimination of structural, data and control hazards. The refinement steps in [36, 17] are guided by the intent to turn the given informal problem description into a well documented and justified to be correct solution of the problem by C++ code; each of the intermediate models reflects some important design decision.

2.4 Decomposition Technique and Function Classification

(De)Composition provides horizontal structuring of systems as a tissue of separate subsystems which interact with each other through well defined interfaces. For decomposing a system one has to recognize, within the global system behavior, separate roles and to encapsulate them into the definition of subsystems. Thereby the strength of a decomposition method is related to its abstraction capabilities for defining each component's behavior and its interface for the interaction with other components, with the necessary transparency and precision, conciseness and completeness, abstracting from internal details. The freedom of abstraction the designer enjoys with ASMs offers the corresponding freedom to device rigorous system (de)composition techniques which are not limited to notational (signature) issues, as large parts of UML [130] are, but can handle problem oriented modularization by reflecting abstractly semantical component content (functionality) and interface behavior (interaction content). We explain in this section that ASMs not only support best modularization practice, but also enhance it by lifting it from programming to rigorous high-level design and extend it to the case of multi-agent systems.

Best modularization practice needs a most flexible abstraction mechanism[9]. The **ASM function classification** realizes a rigorous high-level concept of modularity which leaves more design freedom than (but can be specialized to) the modularization and compositionality principles in current programming languages. In an ASM M we distinguish *basic* functions from *derived* functions (which are defined in terms of basic ones). Within derived or basic functions we separate *static* functions (which remain constant during M-computations) from *dynamic* ones; among the dynamic functions we distinguish—using a terminology appearing in Parnas' Four Variable Model [107]—the *controlled* ones, which are subject to change by an update appearing in a rule of M, from the *monitored* ones which can change only due to the environment or more generally due to actions of other agents. Last but not least there are *shared* functions which can

[9] "A module achieves program simplification by providing an abstraction. That is, its function can be understood through its interface definition without any need to understand the internal details."[96]

be written by M and by some other agent and for whose consistency typically a protocol has to be devised. Shared functions help to naturally reflect multi-agent computations and combined read/write use of locations (like ports in chip design which are used for both input and output).

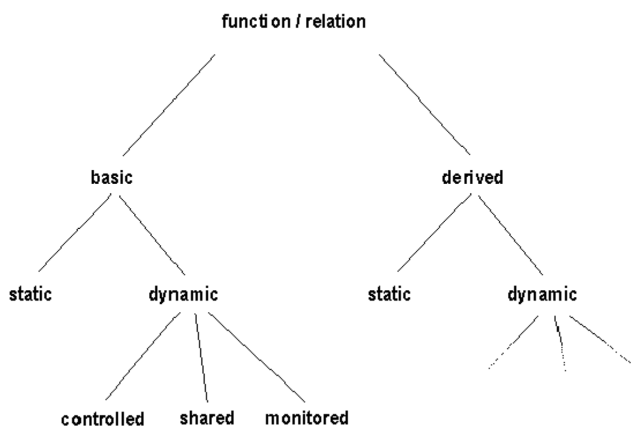

Fig. 3. ASM Function Classification

Distinguishing between basic and derived, static and dynamic or controlled and monitored functions constitutes a rigorous high-level realization of Parnas' **information hiding** principle achieving "designer control of the distribution of information" [103, p.344] through the extent to which the description of abstract non-controlled functions is given to the programmer. A function may be described by its signature only (conveying only syntactic or type information), by an implicit axiomatic definition (through possibly semantical constraints describing "the what without the how"), by an explicit or recursive definition (describing the mathematical law which determines the semantical meaning of the function modulo the occurring basic functions), by a module interface definition, by an algorithm (fixing also the procedure by which the function values are computed), by another ASM or by a detailed program (giving away all the implementation details). The programmer has full control (read and write access) only upon the controlled functions, the ones which he is asked to program, but for accomplishing this task he is freed from the obligation to also care about how to define the non-controlled functions which he can read without any restriction when determining the arguments and the new values for updates of controlled functions, and upon whose effect he can rely as much as he has been informed.

Non controlled functions allow the designer to specify and reason about his system on the basis of what is given through such functions. This is not a phenomenon of underspecification of the system nor does the system become fuzzy through the possible lack of detailed information on the non controlled functions. Whenever, in a run of an ASM M, a non controlled function e is invoked in a state S, it appears in S as a given function and thus can be freely used, although we may not know (or may want to abstract from) how the current interpretation of e, in S, has become available to us. For example in an update $a := f(a, g(b))$ (with say g declared as a unary monitored function and b as a static 0-ary function), the term $f(a, g(b))$ is used to denote an object in state S, obtainable through a computation (the standard logical interpretation) which follows the term structure and uses the interpretation of a, b, g in S which is *known* once the state is given, never mind how these interpretations have been defined. It is indeed only conceptually that we need to (de)compose our systems, therefore for determining the value of the complex construct $f(a, g(b))$ in S we do abstract from the way the monitored function g and the static function b have been given in S but nevertheless have the functions available for use.

Numerous **examples** illustrate the power of information hiding and modularization through non-controlled functions in ASMs. In the model for the IEEE VHDL'93 standard [30, 31] the details of the propagation of signal values in zero time are hidden from the definition of the VHDL kernel by two derived functions for the so–called driving and effective values (which we define by a recursion on the signal sources resp. on port association elements from ports to signals, abstracted from the complex algorithms in the language reference manual [86]); similarly the availability of values in zero time, at certain points in a circuit, can be described by a static (or derived) function, usually expressing a combinatorial network. The details of error propagation and handling in the Java Virtual Machine are separated in [44] from the main machine definition by a derived, recursively definable *catcher* function. The details of the unification procedure have been encapsulated in [39] into a static function *unify*. The reactive behaviour of (concurrently operating) PVM daemon processes, in response to requests coming from local tasks to carry out some PVM instruction or to the reception of a message from another daemon, is modelled in [28, 29] using a monitored function *event* whose values *event(daemon)* are assumed to remain stable until *daemon* has read the function destructively; this abstracts from the details of a communication scheme. In [43] compilation features are separated from the dynamic semantical aspects by encapsulating them into static functions; the implementation defined scheduling algorithm, which resides on top of the synchronization of Java threads as required by the language reference manual, is encapsulated into a monitored function. Derived and monitored functions can also been used for modelling the interface between the discrete and the continuous world in hybrid systems. An example is the real-valued speed function appearing in the high-level Falko-ASM, developed at Siemens within a project for building a tool for train schedule development and validation. Train speed has been encapsulated there into a derived function which is computed from monitored data, using the laws of physics and continuous mathematics.

Also syntactical composition principles can be put to fruitful use in ASMs. An example is the composition of a VLSI implemented microprocessor out of basic architectural components (sub-ASMs) in [24]; it provided the possibility to analyse processor properties by reducing them to properties of the components and has been the key for accomplishing a reverse engineering project for the microprocessor control of the APE100 massively parallel machine. In [55] this approach is exploited further for supporting hardware/software-partitioning and instrumenting of building blocks. Another example is the submachine concept, illustrated by the replacement in [40] of the above mentioned function *unify* by a sub-ASM which implements a unification algorithm. A promising scheme for composition of ASMs out of generic (parameterized) ASMs, supporting library building and reuse of components, has recently been defined analysing transformations which occur typically in programming language compilation [78].

The flexibility ASMs offer for rigorous high-level definitions of system components and their interaction facilitates a transparent component reuse and helps to master the compatibility problem for the design of heterogeneous systems; a successful industrial use of this property is reported in this volume [89]. The abstraction possibilities also help keeping the global understanding of a system during its entire development, from the ground model and through the refinement definitions down to the implementation. Statistical evidence and practical experience reported in [10] confirm the importance, for a software project to succeed, of maintaining such a global system view and of the possibility to make it accessible through an appropriate documentation. We made the same experience when using ASMs for documenting the basic functionality of a large C++ software package, developed and used successfully at Siemens for the simulation of railway systems. The problem consisted in specifying the components of the system at a level of abstraction appropriate to make the constraints about the component interaction explicit [102].

Distributed (multi-agent) ASMs, where multiple sequential ASMs operate concurrently, support the decomposition in a particularly important way. Gurevich [75] has defined a notion of multi-agent ASM run which avoids any commitment to particular forms of concurrency, distilling the bare minimum needed for guaranteeing the global consistency of local states. It says that such a run is a partially ordered set M of "moves" x of a finite number of sequential ASMs (agents) $A(x)$ in which

1. each move has only finitely many predecessors,
2. the moves of every agent are linearly ordered,
3. each initial segment X corresponds to a state $\sigma(X)$—the result of executing all moves in X—which for every maximal element $x \in X$ is obtainable by applying move x in state $\sigma(X - \{x\})$.

This definition guarantees what is needed in applications, namely that the scheduling of moves which are independent of each other in the given run does not influence the (global view of the) state resulting from the run, more formally expressed: given any finite initial segment of a run, each linearization has the same final (global view of the) state. This provides the freedom needed to faithfully reflect distributed systems as they occur in practical life without being committed a priori to special synchronization concepts, in particular not to framework dependent timing conditions.

The most important (large) example one can point to is the forthcoming distributed real-time model for the to be defined SDL-2000 standard [64, 66] which provides a rigorous but concise and extensible definition for the practitioner's intuitive understanding of the functional and timing behavior of SDL. In addition to the model of Basic SDL-92 in [65] (where the behavior of channels, processes and timers with respect to signal transfer operations is formalized following faithfully the International Telecommunication Union *T Recommendation Z.100*), it provides structural system decomposition in terms of concurrent processes (ASM agents) which interact communicating asynchronously through gates for signal exchange (their interfaces). Another (small) example is the above mentioned robot controller [36] where the concurrently operating components (sequential ASMs) are small finite automata with some additional state structure and interface conditions. We use there shared functions as interfaces which, where necessary, make the sequentialization conditions for certain actions (of the otherwise independent components) explicit and at the same time describe abstractly the effect of these actions. These interfaces simplified considerably the specification and the amount of work necessary to establish the required safety and liveness conditions. Our interfaces can be naturally mapped to the CSP-style synchronization mechanism in [114], a particular communication scheme which has been encoded in Concurrent ML.

2.5 Analysis and Integration Techniques

In this section we survey the practicability of the method which is due to the simplicity of ASMs (making them easy to read and to write for the practitioner) and to the support and enhancement they provide for existing system design and analysis techniques.

Naturaleness. ASM system design comes as a natural thing for the practitioner. Differently from what happens with most academic approaches, the system programmer is not asked to be converted from the common process and state oriented model based reasoning. In the contrary the operational view is enhanced by adding fine tuning, to whatever abstraction level is appropriate for the application under hand, and by putting all this on a simple and rigorous foundation. The reader has seen above the definition—which starts from scratch and is all one needs to understand for applying the method. It is what I had (see [73]) when, as a logician without any practical experience, I joined Gurevich to explore his bold ASM thesis [76]. A frequent critique of formal methods complains about the need for extensive specific training in these methods before they can be put to fruitful industrial use, if they are applicable at all: "formal methods cannot be ... a distinct "add-on" that goes beyond the "normal" things that software developers already do" [105, p.195]. The ASM approach supports directly, without need for special training, the software developers' daily work and improves its quality by enabling the engineer to "produce documents that are easier to read than the programs that they describe" [105, p.195]. Admittedly there is one difficulty, something one has to learn, namely to find the right abstractions. But this is a problem *in re*, of system design, not of the adopted method. ASMs are different from most other methods in that to this unavoidable difficulty they do not add any formalistic complications concerning the design language, the model of computation, the analysis and reasoning schemes, etc. The combination of conceptual programming and rigorous reasoning, characteristic for ASM modelling, directly supports engineering skill and experience, leading from ground models via refinements to implementations.

Integratability. The ASM approach is not monolithic but provides a well defined semantical basis for integrating other description techniques or synthesis methods (see section 3.2) and can itself be integrated, at any development level, into established design flows. The integration potential is technically achieved through committing to standard mathematical and programming language, concepts and notation and through the use of abstract functions; more specifically through the freedom to choose how and to which degree of precision derived, static and monitored functions and integrity constraints are determined (axiomatically, algebraically, functionally, algorithmically, etc.). Due to such functions ASMs represent "code with holes", the important point being that the looseness (desired implementation freedom) of the model is circumscribed by the description of the holes.

This versatility also facilitates tailoring the general ASM specification mechanism to typical data structures and computational patterns which appear again and again in a given application domain. The semantical simplicity and openness of ASMs result in their flexibility for incorporating special application domain knowledge, namely by adapting the general method to the particular structures of that domain. The specialization helps to elaborate the application domain knowledge in a tool based way, thus making the ASM specifications and transformations reusable. An excellent illustration is provided by the Verifix project [131] where the use of ASMs and their refinements—for the semantical definition of source, intermediate and target languages [136]—is coupled with numerous well established or advanced compiler generation and verification techniques (see for an example [68] in this volume) in order to achieve the overall project goal, namely to build a correct compiler for a real-life high-level language on a real-life machine (DEC-Alpha processor).

Verification Techniques. As a consequence of this conceptual openness and of separating design from verification concerns, the method combines problem oriented modelling and simple design with the applicability of state-of-the-art analysis techniques for validation and verification (by "head" or by machine, interactively or fully automated). Most importantly for the practitioner's daily work, ASMs support on the fly justification by the designer because the intuitions and reasoning about the system behavior, which indeed guide and accompany the design, can be turned into rigorous statements about runs and can therefore be checked objectively (refuting or justifying them exploiting traditional mathematical methods). In this way ASMs not only help to debug the designer's work from its very beginning, but have also the important practical effect of producing an intersubjective documentation which makes available to the rest of the world what the designer had in mind when building the system. The key for the success of this program, which leads much further than purely static analysis methods, is that ASM models allow us to abstract, to rigorously formulate and then to verify or validate runtime properties through analysis or simulation.

In certain (theoretical and industrial) circles it has become fashionable to claim that mathematical proofs are not reliable and have to be replaced by machine checked proofs. Without reentering this discussion here (see [21]), it should be pointed out that nothing prevents from verifying standard mathematical reasoning about ASMs by interactive or fully automated proof tools which force us to make explicit, for the checking machine, all the details which good mathematical proofs are geared to hide in order to convey to the human reader a transparent picture of the proof. The additional effort needed for proof checking ASM properties by machines is similar to the additional effort needed to implement ASMs: the abstract domains have to be represented by standard data structures of the prover, the static functions have to be defined completely by equations, axioms, etc.. Delivering our work to the prover forces us to cash the freedom of the high-level reasoning; for this endeavor it is helpful to find and exploit similarities between the proof structures and the design structures. During the last years various such investigations have been started and some encouraging results are already reported in the literature. The correctness proof in [40], for the compilation scheme from Prolog to WAM code, has been verified in KIV [116] (and for some of the proof steps also in Isabelle [111]). The correctness proof for the first refinement step in the model for pipelining a RISC machine in [35] has been checked in KIV [67] and in PVS [124]; the proof checker applied in [79] has discovered an omission of a hazard case in the last refinement step. The properties proved for the production cell ASM [36] have been successfully model checked in [132]; this work is part of an effort to exploit the abstraction possibilities of ASMs to contain the state explosion problem when designing FSMs for model checking. The ASM proofs used in [136]—showing the correctness of bottom-up rewriting specifications for back-end compilers from intermediate languages into binary RISC processor code—have been checked using PVS [58].

Validation Techniques. The method enhances current industrial system design through the possibilities for early and high-level validation of ASMs, whether of system components, of their interaction through interfaces, of particular requirements in the ground model, etc.. We do not want to enter the discussion in the literature [77, 61] whether specifications should be executable or not. There are basically two complementary possibilities to simulate high-level ASMs. One is through a general purpose simulator which can execute any given ASM once its non-controlled functions are defined (interactively, by inputting scenarios, by ad hoc implementations, by using library functions, etc.). An example is the workbench presented in this volume [54] which has been used successfully for extensive simulation of various versions of the above mentioned high-level Falko-ASM. An ASM M can also be made executable by a special purpose interpretation or compilation which exploits the programming environment of the application domain of M for implementing the non-controlled functions. The Prolog kernel ASM in Fig.1 has been implemented at Quintus and used there for various experiments with Quintus Prolog [47]; it took a couple of hours to write this simulator where the two non-trivial non-controlled functions, *unify* and *procdef*, have been linked to available Quintus code. During our work on the model for the JVM in [44], Wolfram Schulte has realized and tested successfully an executable version of it by implementing in Haskell the recursive definitions for the static functions. On the basis of a recent KIV implementation of the Java-ASM in [43], all the Java programs appearing in [69] have been tested successfully [125], except those which lead to compilation errors (due to type check problems, etc.) or deal with arrays, two features we did not cover in our Java model. Another group of examples uses a direct encoding of (refined) ASMs into C++, done ad hoc [36] or using a compilation method (see [17] and Joachim Schmid's recent transformation of the Falko-ASM into C++ code). In between these two approaches we find the special-purpose simulator Gem-Mex developed by Matthias Anlauff to make ASM interpreters for a class of programming languages executable [4]. The tool exploits the ASM specification techniques for the typical structures occurring in the definition and compilation of imperative programming language constructs. This idea has been used and further developed in the Verifix project [136, 78] and represents an instructive example for combining the advantages of the general ASM approach with those coming from the specifics of application domain structures.

3 Universality of ASMs for System Development

In [76] Gurevich gives a penetrating epistemological justification of his strengthening of Turing's thesis which stood at the origin of the discovery of the ASM concept [71, 72], namely that every sequential computational device, on any level of abstraction, can be simulated in lock-step by a sequential ASM of approximately the same size. We collect here some empirical evidence that this thesis is the practical system design analogue of Turing's thesis.

The ASM thesis is confirmed by the following four experiences. First to be mentioned is the multitude and diversity of successful complex system design and analysis projects, carried out using ASMs and covering the definition of the semantics of real-life programming languages and their implementation, real-time algorithms, protocols, control software, virtual machines and architectures, see [33, 2]. Second to be mentioned is the adaptability of ASMs to whatever application domain for solving the ground model problem. Thirdly the ASM method completes, for practical system design, the ambitious structured programming project, synthesizing in a remarkably simple way two fundamental lines of thought in the history of ideas in computer science; see the historical details in section 3.1. Last but not least other well known design and computation models are naturally embedded into ASMs where they can be recognized by specializing the signature, the rules, the constraints, the runs. We show this below for VDM [60], Abrial's Abstract Machine Notation [3], Parnas tables [107], stream X-machines (an important subclass of Eilenberg's X-machines [59]), Petri nets [113] and Codesign FSMs [91]. ASMs appear to be a truly encompassing framework for successfully comparing and combining major current design and analysis techniques, on a rigorous semantical basis which allows one to exploit their respective advantages by appropriate specializations and restrictions.

3.1 Abstract Machines + Abstract State = ASM

The term *abstract machine* has been coined in 1968 by Dijkstra [56], in the context of defining the operating system T.H.E., preceded by the class concept of Simula67 which has been interpreted as a form of abstract machine [80, 52]). Numerous variations appear in the literature, like hierarchical systems, layered architectures, data spaces, virtual machines [106, 137, 84, 126, 51], with corresponding program development methods like top-down design, multi-level programming, stepwise program refinement and data abstraction, etc., which characterize the structured programming and abstract data type method [134, 53] and have prepared the ground for VDM [60] and for Abrial's sophisticated combination of Abstract Machine Notation with proof controlled stepwise refinements [3].

Nevertheless nobody has addressed the epistemological and practically important question of how far the proposed concept of abstract machine can lead us or whether there is at all a rigorous most general notion of abstraction determining the underlying states and operations of such machines. This is even more surprising given that the development of the data abstraction idea in the theory of abstract data types has led to understand that a most general notion of state does exist [101, 62], namely Tarski's logical concept of structure [127]. However in the theory of abstract data types, structures have been encoded into terms so that equational or axiomatic algebraic specification methods could be applied—as a bitter consequence state changes, instead of being related to the dynamics of machine instruction execution, remained static, solutions of fixpoint equations. This approach has its mathematical beauty but turned out to be impractical.

Although the two ingredients of a satisfactory most general and rigorous but practical notion of abstract machines—instruction set machines (abstract dynamics) and structures as states (abstract statics)—were in the literature for more than 15 years, ready to complete the longstanding structural programming endeavour (see [53]) by lifting it from particular machine or programming notation to truly abstract programming on arbitrary structures, nobody performed the natural step to simply combine these two notions and obtain ASMs. Probably this has to do with the fact that in theoretical computer science (and apparently not only there [117]), the value of declarative and of compositional (usually syntax driven) instead of procedural methods has been largely overestimated. In theory circles it still is widely regarded as scientifically not qualifying or rewarding to study the operational, process and machine oriented view of computation[10], with the consequence that the proposed *pure* (functional, algebraic, axiomatic, logical) methods, impractical as they are for dealing with system dynamics, up to now did not really influence or support the way practitioners work. It needed a new start, from scratch (in an attempt to sharpen Turing's thesis) and free from *ideo*-logical a prioris, which eventually led to the right notion, Abstract State Machines, where static and dynamic methods instead of excluding each other can be fruitfully combined.

There is an analogy to the discovery of the notion of Turing machine. During the first third of this century numerous definitions were proposed to formalize the intuitive concept of algorithm and computable function, without really capturing in a convincing way the intuitive notion. The discovery of universal Turing machines [129] provided a justifiably general basis for the theory of algorithms and complexity and laid the conceptual ground for the construction of von Neumann computers. Gurevich's notion of ASM clarifies the relation between different virtual machines and computation models and has laid the ground upon which a practical well founded method could be built for design and analysis of complex real-life hardware/software systems. By the way it has also led to interesting new developments in logic and complexity theory [13, 70, 14].

It would not come as a surprise to see the historian confirm in the retrospective that it was a thorny way which led to ASMs. Büchi, explaining "an approach he developed with Jesse B. Wright in the 1950s in the Logic of Computers Group at the University of Michigan, Ann Arbor"(Sic) (D. Siefkes, op.cit.p.5) speaks about "an intriguing interpretation … of an algebra as a machine" where "the mathematical structures … in our theory are to represent the real objects"[49, p.76]—to exploit it for an (elegant) algebraic interpretation of finite automata as finite algebras with unary functions. Scott [118] lifted finite automata to abstract machines by adding, to the change of the internal control state, the inspection and transformation of a memory state, but he formulated these memory state transformations by static functions $f : M \to M$ whereby they remain global and inherit the annoying frame problem, as is the case in numerous followers of this approach [59, 51, 82, 92, 83]. The ASM machine instructions (guarded function updates) enable us to distinguish between (and to combine the advantages of) a global state view and local state transformations, a particularly important feature for distributed systems, see below the discussion of the concept of Globally Asynchronous Locally Synchronous (GALS) machines.

3.2 Encompassing Sequential Models of Computation

The VDM approach to system modelling is very carefully explained in [60]. VDM is restricted to sequential runs. The abstraction level is fixed, for sets by the VDM-SL types (which have to be built from basic types by constructors),

[10] Michel Sintzoff, in an e-mail discussion in June 1996, commented: "If the operational style of dynamical systems is "dirty", then Poincaré should be removed from the history of mathematics". See the discussion in [120, section 2.4] of the fundamental role operational methods play for expert knowledge.

for functions it permits explicit and implicit definitions, for operations one is allowed to use procedures (with possible side effects). The notion of state is restricted to records of variables (sets of 0-ary functions) which are classified into read/write variables only. The tutorial [60] is biased towards functional modelling[11] although assignments and state based specifications are supported by VDM. It would be interesting to know whether and to what extent the IFAD tool support for VDM could be enriched to cover the greater semantical freedom offered by ASM modelling, even if restricted to the sequential case.

Abrial's B-Method is the method which in spirit and conceptually comes closest to the ASM method. The B-method is model oriented, the way also its predecessors Z and VDM are, but in addition it is based on Abstract Machine (instead of purely axiomatic or functional) Notation (AMN). A difference in spirit comes out in the way abstract machine programs are related to the justification that the program does what it is supposed to do. In B "the idea is to accompany the technical process of *program* construction by a similar process of *proof* construction, which guarantees that the proposed program agrees with its intended meaning" and as a consequence the "simultaneous concerns about the architecture of a program and that of its proof" [3, p.XI] characterize the B-method. They determine its compositional constructs (guards, sequentialization, parallelism, non-deterministic choice of actions or objects, inclusion/import, using/seeing) as well as its refinement concept and thereby guide the specification and design process. This is enforced by the supporting proof tools. The ASM designer has a greater freedom to choose, from case to case and depending on the design level, the most appropriate and convenient way to proceed; the method avoids any a priori fixed combination of design language and proof system, inviting the software engineer to use whatever form of mathematical language, programming notation and rigorous reasoning may be useful for defining the desired system, for implementing it through further detailing (see above the comparison of the ASM and B refinement concepts) and for making one "convinced that the software system in question is indeed correct" [3, p.XV]. ASMs adhere to the idea that there are many layers not only for the design, but also for justifying a design (i.e. of correctness "proofs").

A conceptual difference between the B-method and the ASM method stems from the different epistemological explanation of what "is" an abstract machine. For Abrial it "is a concept that is very close to certain notions well-known in programming, under the names of modules, classes or abstract data types" [3, p.XV] and as a matter of fact is defined on the basis of pre- and post-conditions and specifically of Dijkstra's weakest pre-condition theory. ASMs are the practical hardware/software design analogue of Turing machines, result of a modern analysis of Turing's thesis [76]; they are based only on standard mathematical

[11] because this allows to go, as the authors say, "without the distraction of operation syntax, side-effects and access restrictions to external variables" [60, p.XII]. The ASM philosophy is as follows. If there is an effect which plays a role, it should stand out explicitly; if it does not play a role, it should be hidden (abstracted away) so it does not appear not even as side-effect.

logic (which includes Thue's [128] nowadays omnipresent notion of transition system). The natural semantical basis for ASMs protects the notion from certain complications the B-method has to face due to its conceptual heritage. For example the fundamental assignment operation, which *is* used throughout, is not introduced as semantically basic but has to be defined syntactically through the not completely elementary concept of substitution. The proof method is tailored for termination proofs, using specific refinement techniques which in certain situations lead to problems (see [110, 7]). The axiomatic foundation by Dijkstra's weakest precondition theory is not easy to understand; furthermore it seems to lead to the tendency to specify machine operations using logical descriptions instead of expressing the dynamics directly through local state updates. AMN deals only with sequential machines (although Abrial is working on an extension of AMN to distributed computations).

There are also some technical differences. The state notion in Abrial's AMN is that of a set of finitely many variables, instead of arbitrary Tarski structures. Sets and functions are supposed to be finite (whereas with ASMs dealing with the finiteness problem can be postponed to the moment when it comes to implement the abstract machine into specific control and data structures and to worry about garbage collection). Anchoring ASMs in classical logic avoids the well definedness problem without having to rely upon neither three-valued logic nor any special proof calculus to circumvent the logic of partial functions [12].

Eilenberg's X-machines [59] are state machines which, in their current *control state*, modify their current *memory state* as indicated by their *input*, produce an *output* and pass to the next control state; differently from a finite automaton, an X-machine has as input σ not a letter, but a partial memory function $\sigma : X \to X$ which is consumed by applying it to the current memory state. X-machines are a special form of the abstract machines defined by Scott in 1967 [118]. In 1988 Holcombe used X-machines for describing Turing machines, Petri nets, finite automata and a small storage system and proposed "to integrate the machine ideas central to the modelling of the control of the processing together with the data type descriptions of the processing operations in a unified methodology ... for dynamic system specification"[82, p.72]. Starting with [92] a practically important subclass of these machines, called stream X-machines, has been used to generalize Chow's finite state machine testing method [50, 83].

X-machines are specializations of what we call *control state ASMs*, i.e. ASMs where all the rules have the (usually graphically displayed) form

$$s \xrightarrow[\mathit{Rule}]{} s'$$

with elements s, s' of a set of control states (the "internal states" in finite automata) and standing for

$$\textbf{if } \mathit{currstate} = s \textbf{ then } \begin{array}{l} \mathit{currstate} := s' \\ \mathit{Rule} \end{array}$$

X-machines are sequential control state ASMs where *Rule* is a single update by a global memory function f, i.e. of form $\mathit{currmemory} := f(\mathit{currmemory})$.

Stream X-machines are a specialization of what we call *Mealy-ASMs*, i.e. ASMs with a monitored input function *in* where all the rules have the (usually graphically displayed) form

$$s \xrightarrow[Rule]{Cond(in)} s'$$

standing for

$$\textbf{if } currstate = s \wedge Cond(in) \textbf{ then } \begin{array}{l} currstate := s' \\ Rule \end{array}$$

Rule may and usually does include an output update *out* := .... Mealy-ASMs provide a uniform semantical basis for event driven machine computation models, like Message Sequence Charts, Discrete Event Systems [112], etc.. Stream X-machines are Mealy-ASMs where *Rule* is restricted as in X-machines, the output is defined as concatenation of the current with the previous output, and *Cond(in)* is restricted to $\sigma \in A$ for the underlying function alphabet A.

The (stream) X-machines, in the form they are defined and used in [82, 92, 83], are exposed to the difficulties of the frame problem. They inherit from their algebraic origin a global (Cartesian product) memory view, showing up in the frequent "don't care" and "no change" entries in function definitions. This is different from the combination of global memory view with local update view in control state ASMs, Mealy-ASMs and ASMs in general. Also in ASMs there is a global set X, the so called *superuniverse* [75], which together with the functions defined on X forms the global state (memory and control). But memory changes can nevertheless be thought of as happening locally, by formulating the intended updates $f(t_1, \ldots, t_n) := t$ in rules. The semantical definition of ASMs states once and for all that controlled functions change only due to rule execution, freeing the designer from having to think about this frame condition and having to present particular instances of it at each state change. It would be interesting to know how much of the decomposition, structuring and refinement principles from algebraic automata theory can be extended to capture some of the semantical structuring possibilities for ASM. The same question applies to extensions of Chow's finite automata based testing method [50, 83].

Parnas tables [107] exploit the mathematical matrix notation for a concise geometrical layout of multiple case distinctions in predicate and function definitions. Various semantical interpretations of such tables have been introduced for software documentation purposes, see [1, 88] for a survey and [87] for a formal semantics. We illustrate three frequently used types of Parnas tables by ASM definitions which provide a simple and uniform semantical basis for such 2-dimensional notations.

Normal tables are used to assign a value $t_{i,j}$ to $f(x, y)$ in case the row condition $r_i(x)$ and the column condition $c_j(y)$ are both true. The consistency of such definitions usually results from the disjointness of the row conditions and of the column conditions.

$$\begin{array}{l|lcl} N(f) & c_1 & \dots & c_n \\ \hline r_1 & t_{1,1} & \dots & t_{1,n} \\ \vdots & & \vdots & \\ r_m & t_{m,1} & \dots & t_{m,n} \end{array} \quad \text{meaning} \quad \begin{array}{lll} \textbf{if } r_i(x) \wedge c_j(y) & \textbf{where} & 1 \leq i \leq m \\ \textbf{then } f(x,y) := t_{i,j} & & 1 \leq j \leq n \end{array}$$

Inverted tables are used to assign a value t_j to $f(x, y)$ in case at least one of the leading conditions $r_i(x, y)$ (which usually depend only on one of the variables) and the corresponding side condition $c_{i,j}(x, y)$ are both true.

$$\begin{array}{l|lcl} I(f) & t_1 & \dots & t_n \\ \hline r_1 & c_{1,1} & \dots & c_{1,n} \\ \vdots & & \vdots & \\ r_m & c_{m,1} & \dots & c_{m,n} \end{array} \quad \text{meaning} \quad \begin{array}{lll} \textbf{if } r_i(x,y) \wedge c_{i,j}(x,y) & \textbf{where} & 1 \leq i \leq m \\ \textbf{then } f(x,y) := t_j & & 1 \leq j \leq n \end{array}$$

Decision tables are used to trigger an action t_j (read: to assign a value t_j to $f(x_1, \dots, x_k)$) in case all the conditions $r_{i,j}(x_1, \dots, x_k)$ appearing in column j are satisfied, where usually each condition $r_{i,j}(x_1, \dots, x_k)$ concerns one term sequence s_i which guides the case distinction and is displayed in the leading column.

$$\begin{array}{l|lcl} D(f) & t_1 & \dots & t_n \\ \hline s_1 & r_{1,1} & \dots & r_{1,n} \\ \vdots & & \vdots & \\ s_m & r_{m,1} & \dots & r_{m,n} \end{array} \quad \text{meaning} \quad \begin{array}{ll} \textbf{if } \bigwedge_{1 \leq i \leq m} r_{i,j}(s_i) & \textbf{where } 1 \leq j \leq n \\ \textbf{then } f(x_1, \dots, x_k) := t_j & \end{array}$$

The theory for Parnas tables and the underlying Four Variable Model [107] is focussed on sequential systems with states consisting of finitely many variables, classified into monitored and controlled; recently also shared variables have been included [104]. The state variables are treated as function of time and thereby capture system dynamics; this does not work however for distributed features which are not only a function of time. In applications of Parnas tables we observe a frequent use of "auxiliary" functions, predicates, data structures, etc. whose semantical integration is taken for granted. The ASM definition of tables provides a simple rigorous foundation for this natural form of semantical table modularization and decomposition, in particular through the fine grained function classification explained above; it also makes the ASM refinement techniques applicable to tables and thus helps to document the system decomposition. The ASM definition of tables also overcomes the limitations of the 2-dimensional geometrical layout due to which the tables typically define functions and predicates which are viewed as monadic or binary (although in another analysis layer, each argument may appear as a tuple). Despite their state based dynamic orientation, Parnas tables are reminiscent of a "declarative" view of computation: the 1-step transition relation is expressed not by transition rules using destructive updates (assignments), but by mathematical formulae which use the well known x/x'-notation of temporal logic and thereby inherit the frame problem—appearing in the so-called NC-conditions ("No Change") of tables.

3.3 Encompassing Distributed Models of Computation

Petri nets are among the most popular models for distributed computation. We do not want to go here into the details of the numerous different versions of Petri nets. Instead we give a definition of naturally *generalized* Petri nets which covers all cases in the literature. Any first-order condition (on markings, colours, whatever you want) is allowed as precondition, involving not only single but arbitrary combinations of places; any function update (to change the marking, the colours, whatever) is allowed for any combination of places affected by a rule. This generalization supports the integration of Petri nets into the ASM framework and vice versa. Here is the definition.

Let P be a set (of so called *places*). A Petri-ASM (over P) is a multi-agent ASM where the rules of each agent are of the following (usually graphically displayed) form:

$$\textbf{if } \alpha_1(p_1) \wedge \ldots \wedge \alpha_n(p_n) \textbf{ then } \begin{array}{l} f_1(q_1) := t_1 \\ \vdots \\ f_m(q_m) := t_m \end{array}$$

where p_i, q_j are finite sequences of elements of P, α_i are arbitrary first-order formulae, f_j are arbitrary functions on places and t_j are arbitrary terms used to determine values to be associated to functions at places. In Petri nets typically the p_i, q_j are sequences of length 1 and some p_i are idential to some q_j. In the case of standard marked Petri nets for example, $\alpha_i \equiv L_i \subseteq m(p_i)$ where $m : P \rightarrow M$ is the dynamic marking function and L_i is the static marking precondition (so called label) attached by the rule to p_i; f_j describes the intended update of $m(q_j)$ (deleting the precondition marks from q_j, if there are any, and adding the postcondition marks R_j attached by the rule to q_j, say $m(q_j) := m(q_j) - L_j + R_j$). By deciding which sequential agent gets which rules to execute, the intended distributed nature of Petri net computations is realized faithfully. Attributing transitions to agents can simplify the analysis of runs and the corresponding proofs. Specializations of Petri nets by particular synchronization mechanisms are reflected by corresponding (order) restrictions on the runs of the Petri-ASM.

Last but not least we want to mention the very interesting concept of **Codesign Finite State Machines** (CFSMs), defined with the goal to "combine the advantages of verifiability in synchrony and flexibility in asynchrony in a globally asynchronous, locally synchronous (GALS) model" [91, section 4.1] and synthesizing a detailed analysis of the advantages and drawbacks of the major models of computation in use for system design. CFSMs can be defined as distributed ASMs whose components are Mealy-ASMs, possibly coming together with a global scheduler controlling the interaction of the components and/or with timing conditions in case of durative instead of atomic component actions. The (in ASM terminology sequential) Mealy-ASM components have a locally synchronous behavior, whereas the partial order of the distributed ASM reflects the globally asynchronous system character. It should be stressed that also the data structure capabilities of Mealy-ASMs are exploited for the definition of

CFSMs; namely the transitions are allowed to refer to arbitrary combinational (external and instantaneous) functions. See [91] for further details.

4 Conclusion and Future Research

Various prototypical tools have been developed for designing and executing ASMs; for an overview see [54] in this volume. More advanced and industrially satisfactory tool support is needed for defining, simulating and visualizing, debugging, transforming (refining, implementing, where possible through code generation), analysing (testing and verifying) ASMs. The tool environment should support to capture design knowledge in a rigorous and electronically available way. It should be linked as much as possible to established design flows and exploit their achievements; for an encouraging successful project in this context see the description [89] in this volume. Together with an advanced tool environment a model and a proof theory of ASMs are needed which in particular define practical refinement principles, ideally together with corresponding proof schemes; see [115] for a good start. The proof theory we need should alleviate the formalization effort encountered in practical applications, instead of making things more difficult; it should support abstraction and refinement, structuring and layering in proofs instead of focussing on a single (worse if only machine oriented) level of detailing. The ASM theory should build upon and extend what has been achieved in more specific design approaches which are supported by a rich tool environment, like CFSMs, B, UML, VDM, Petri nets, etc. This is a ridge walk between freedom and discipline, creativity and pattern oriented design, generality and specialization, expressability and limitations by tool support.

The "codeless form of programming" offered by ASMs helps porting application programs from one platform or language to another and could lead to fruitful applications for plug-and-play software technology [9]. Paradigmatic and parameterized ASM components and (de)composition techniques for constructing them have to be defined and to be made available in libraries. ASMs should also be put to use to enhance current (mostly signature oriented) software architecture description techniques by adding semantical content to the structural definitions. This is particularly promising at the levels of what in [122] is called *conceptual architecture* and *module interconnection architecture*. Conceptual architecture refers to the ground model level where domain specifics play a major role; the module interconnection architecture level reflects implementation decisions which are independent of any particular progamming language, such as the logical/functional decomposition, layers with allowable import/export relations, interface constraints, etc.. The integration potential of ASMs, as a universal model of computation, is crucial in achieving the goal to capture the overall behavior of a complex system in terms of the behavior of the interacting components, by whatever rigorous descriptions are appropriate (static, dynamic, functional, state-based, etc.). This will not only improve the reuse and the reconfiguration possibilities for system descriptions, but also contribute to a satisfactory comparative analysis of different architectural models.

It would be interesting to investigate the possibilities ASMs offer for a theoretical foundation of good testing methods for high-level design. ASMs can help to solve the crucial and essentially creative part of test case selection, given that this selection is driven by appplication domain expert knowledge and thus can be formulated using the ASM ground model. The ground model may allow one to even generate a test scheme. Furthermore the ground model supports solving the oracle problem of testing: the expected output, which has to be compared with the execution output, can be defined using the ground model specification (which is independent of the programming language where the system will be encoded). A similar remark applies to static testing (code inspection) where one has to formulate the properties to be checked.

No doubt in this presentation ASMs have received much praise. To figure out whether this is only adulation or whether there is a *fundamentum in re*, there is only one thing one can do: put ASMs to the test.

Acknowledgment. Thanks for the critical remarks received on drafts of this paper from Donatella Barnocchi, Don Batory, Jonathan Bowen, Axel Dold, Mike Holcombe, Jim Huggins, Hans Langmaack, Hanno Nickau, Peter Päppinghaus, Gerhard Schellhorn, Peter Schmitt, Wolfgang Schönfeld, Dilip Soni, Lothar Thiele, Kirsten Winter, Jeff Zucker. To Michel Sintzoff thanks also for the stimulating discussion on the history of the concept of abstract machines. Last but not least thanks to my students in the Fall 1998 *Specification Methods* Course, in particular Marenco Yari, Fabrizio Sanna, Cristian Gozzi, Tommaso Mangini, Emanuele Lupi, Andrea Pieroni and Cinzia Ridolfo, for the ASM reuse experiments mentioned above.

In: Hutter, D., Stephan, W., Traverso, P., Ullmann, M. (eds): Current Trends in Applied Formal Methods (FM-Trends 98). Lecture Notes in Computer Science, Vol. 1641, pp. 1-43. Springer-Verlag, Berlin Heidelberg New York (1999)

References

1. Abraham, R.: Evaluating Generalized Tabular Expressions in Software Documentation. M. Eng. Thesis, CRL Report 346, McMaster University, Hamilton, Ontario, Canada (1997)
2. http://www.eecs.umich.edu/gasm/, http://www.uni-paderborn.de/cs/asm.html
3. Abrial, J.-R.: The B-Book. Assigning Programs to Meanings. Cambridge University Press (1996)
4. Anlauff, M., Kutter, P., Pierantonio, A.: Formal Aspects of and Development Environments for Montages. In: Sellink, M. (ed): 2nd International Workshop on the Theory and Practice of Algebraic Specifications. Springer Workshops in Computing (1997)
5. Araujo, L.: Correctness Proof of a Distributed Implementation of Prolog by Means of ASMs. J. of Universal Computer Science. Special ASM Issue 3(5) (1997)
6. Aristotle. Analytica Posteriora I,1, 71a,1 sq.
7. Banach, R., Poppleton, M.: Retrenchment: An Engineering Variation on Refinement. In: Bert, D. (ed): B'98: Recent Advances in the Development and Use of the

B Method. Lecture Notes in Computer Science, Vol. 1393. Springer-Verlag, Berlin Heidelberg New York (1998) 129-147

8. Barnocchi, D.: L"Evidenza" nell'assiomatica aristotelica. Proteus II,5 (1971) 133–144
9. Batory, D., Singhai, V., Sirkin, M., Thomas, J.: Scalable Software Libraries. ACM SIGSOFT'93: Symposium on the Foundations of Software Engineering. Los Angeles/California (1993)
10. Batory, D., Coglianese, L., Goodwin, M., Shafer, S.: Creating Reference Architectures: An Example from Avionics. Symposium on Software Reusability. Seattle/ Washigton (1995)
11. Batory, D., O'Malley, S. : The Design and Implementation of Hierarchical Software Systems with Reusable Components. ACM Transactions on Software Eng. and Methodology (October 1992)
12. Behm, P. Burdy, L., Meynadier, J.-M.: Well Defined B. In: Bert, D. (ed): B'98: Recent Advances in the Development and Use of the B Method. Lecture Notes in Computer Science, Vol. 1393. Springer-Verlag, Berlin Heidelberg New York (1998) 29-45
13. Blass, A., Gurevich, Y.: The Linear Time Hierarchy Theorems for Abstract State Machines. J. of Universal Computer Science. Special ASM Issue, 3(4) (1997) 247-278
14. Blass, A., Gurevich, Y., Shelah, S. : Choiceless Polynomial Time. EECS Dept. University of Michigan, Technical Report CSE-TR-338-97 (1997)
15. Beierle, Ch., Börger, E.: Specification and Correctness Proof of a WAM Extension with Abstract Type Constraints. Formal Aspects of Computing 8(4) (1996) 428-462
16. Beierle, Ch., Börger, E.: Refinement of a Typed WAM Extension by Polymorphic Order-Sorted Types. Formal Aspects of Computing 8(5) (1996) 539-564
17. Beierle, Ch., Börger, E., Đurđanović I., Glässer, U., Riccobene,E.: Refining Abstract Machine Specifications of the Steam Boiler Control to Well Documented Executable Code. In: Abrial, J.-R., Börger, E., Langmaack, H. (eds.): Formal Methods for Industrial Applications. Specifying and Programming the Steam-Boiler Control. Lecture Notes in Computer Science, State–of–the–Art Survey, Vol. 1165. Springer-Verlag, Berlin Heidelberg New York (1996) 52–78
18. Börger, E.: A Logical Operational Semantics for Full Prolog. Part I: Selection Core and Control. CSL'89. Lecture Notes in Computer Science, Vol. 440. Springer-Verlag, Berlin Heidelberg New York (1990) 36–64
19. Börger, E.: A Logical Operational Semantics for Full Prolog. Part II: Built-in Predicates for Database Manipulations. In: Rovan, B. (ed): MFCS'90. Mathematical Foundations of Computer Science. Lecture Notes in Computer Science, Vol. 452. Springer-Verlag, Berlin Heidelberg New York (1990) 1–14
20. Börger, E.: Logic Programming: The Evolving Algebra Approach. In: Pehrson, B., Simon, I. (eds): IFIP 13th World Computer Congress 1994. Volume I: Technology and Foundations. Elsevier, Amsterdam (1994) 391–395
21. Börger, E.: Why Use Evolving Algebras for Hardware and Software Engineering. In: Bartosek, M., Staudek, J., Wiedermann, J.(eds): SOFSEM'95. 22nd Seminar on Current Trends in Theory and Practice of Informatics. Lecture Notes in Computer Science, Vol. 1012. Springer-Verlag, Berlin Heidelberg New York (1995)236–271
22. Börger, E., Dässler, K.: Prolog: DIN Papers for Discussion. ISO/IEC JTCI SC22 WG17 Prolog standardization document, no. 58,.NPl, Middlesex (1990) 92–114
23. ISO/IEC 13211-1 Information Technology-Programming Languages-Prolog-Part 1: General Core (1995)

24. Börger, E., Del Castillo, G.: A Formal Method for Provably Correct Composition of a Real-Life Processor out of Basic Components (The APE100 Reverse Engineering Project). Proc. of the First IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'95). Extended version in: Gurevich, Y., Börger, E. (eds): Evolving Algebras. Mini–Course. University of Aarhus. BRICS NS-95-4 (1995) 195–222
25. Börger, E., Demoen, B.: A Framework to Specify Database Update Views for Prolog. In: Maluszynski, M. J. (ed): PLILP'91. Lecture Notes in Computer Science, Vol. 528. Springer-Verlag, Berlin Heidelberg New York (1991) 147–158. See also: The View on Database Updates in Standard Prolog: a Proposal and a Rationale. In: ISO/IEC JTC1 SC22 WG17 Prolog Standardization Report no.74 (February 1991) pp. 3-10
26. Börger, E., Đurđanović, I.: Correctness of Compiling Occam to Transputer Code. Computer Journal 39(1) (1996) 52–92
27. Börger, E., Đurđanović, I., Rosenzweig, D.: Occam: Specification and Compiler Correctness. Part I: The Primary Model. In: Olderog, E.-R. (ed): Proc. of PRO-COMET'94 (IFIP Working Conference on Programming Concepts, Methods and Calculi). North-Holland (1994) 489–508
28. Börger, E., Glässer, U.: A Formal Specification of the PVM Architecture. In: Pehrson, B., Simon, I. (eds): IFIP 13th World Computer Congress 1994. Volume I: Technology and Foundations. Elsevier, Amsterdam (1994) 402–409
29. Börger, E., Glässer, U.: Modelling and Analysis of Distributed and Reactive Systems Using Evolving Algebras. In: Gurevich, Y., Börger, E. (eds): Evolving Algebras. Mini–Course. University of Aarhus. BRICS NS-95-4 (1995) 128–153
30. Börger, E., Glässer, U., Mueller, W.: The Semantics of Behavioral VHDL'93 Descriptions. EURO-DAC'94 European Design Automation Conference with EURO-VHDL'94. Proc. IEEE CS Press, Los Alamitos/CA (1994) 500–505
31. Börger, E., Glässer, U., Mueller, W.: Formal Definition of an Abstract VHDL'93 Simulator by EA–Machines. In:Delgado Kloos, C., Breuer, P.T. (eds): Semantics of VHDL. Kluwer (1995) 107–139
32. Börger, E., Gurevich, E., Y.., Rosenzweig, D.: The Bakery Algorithm: Yet Another Specification and Verification. In: Börger, E. (ed): Specification and Validation Methods. Oxford University Press, (1995) 231–243
33. Börger, E., Huggins, J.: Annotated Bibliography on Abstract State Machines (ASMs). EATCS Bulletin (February 1998)
34. Börger, E., Lopez-Fraguas, F.J., Rodrigues-Artalejo, M.: A Model for Mathematical Analysis of Functional Logic Programs and their Implementations. In: Pehrson, B., Simon, I. (eds): IFIP 13th World Computer Congress 1994. Volume I: Technology and Foundations. Elsevier, Amsterdam (1994) 410–415. Full version: Towards a Mathematical Specification of Narrowing Machines. Report DIA 94/5, Dep. Informática y Automática. Universidad Complutense, Madrid (March 1994) 1–30
35. Börger, E., Mazzanti. , S.: A Practical Method for Rigorously Controllable Hardware Design. In: Bowen, J.P., Hinchey, M.B., Till, D. (eds): ZUM'97: The Z Formal Specification Notation. Lecture Notes in Computer Science, Vol. 1212. Springer-Verlag, Berlin Heidelberg New York (1997) 151–187
36. Börger, E., Mearelli, L.: Integrating ASMs into the Software Development Life Cycle. J. of Universal Computer Science, Special ASM Issue, 3 (5) (1997) 603-665
37. Börger, E., Riccobene, E.: A Formal Specification of Parlog. In: Droste, M., Gurevich, Y. (eds): Semantics of Programming Languages and Model Theory. Gordon and Breach (1993) 1–42

38. Börger, E., Rosenzweig, D.: An Analysis of Prolog Database Views and their Uniform Implementation. In: Prolog. Paris Papers–2. ISO/IEC JTC1 SC22 WG17 Prolog Standardization Report no.80 (July 1991) 87–130
39. Börger, E., Rosenzweig, D.: A Mathematical Definition of Full Prolog. Science of Computer Programming. 24 (1995) 249–286
40. Börger, E., Rosenzweig, D.: The WAM–Definition and Compiler Correctness. In: Beierle, Ch.,Plümer, L. (eds): Logic Programming: Formal Methods and Practical Applications. Elsevier Science B.V./North–Holland (1995) 20–90
41. Börger, E., Salamone, R.: CLAM Specification for Provably Correct Compilation of CLP($\mathcal{R}$) Programs. In: Börger, E. (ed): Specification and Validation Methods. Oxford University Press, (1995) 97–130
42. Börger, E., Schmitt, P.: A Formal Operational Semantics for Languages of Type Prolog III. Lecture Notes in Computer Science, Vol. 533. Springer-Verlag, Berlin Heidelberg New York (199) 67–79
43. Börger, E., Schulte, W.: Programmer Friendly Modular Definition of the Semantics of Java. In: Alves-Foss, J. (ed): Formal Syntax and Semantics of Java. Lecture Notes in Computer Science, Vol. 1523. Springer-Verlag, Berlin Heidelberg New York (1999) 353 – 404. Extended Abstract in: Berghammer, R., Simon, F. (eds): Programming Languages and Fundamentals of Programming. University of Kiel (Germany) TR 9717 (1997) 175–181.
44. Börger, E., Schulte, W.: Defining the Java Virtual Machine as Platform for Provably Correct Java Compilation. In: Brim, L., Gruska, J., Zlatuska, J. (eds): Proc. MFCS'98. Lecture Notes in Computer Science, Vol. 1450. Springer-Verlag, Berlin Heidelberg New York (1998) 17–35
45. Börger, E., Schulte, W.: Initialization Problems for Java. Software—Concepts and Tools 20 (4) (1999)
46. Börger, E., Schulte, W.: Modular Design for the Java Virtual Machine Architecture. In: Börger, E. (ed): Architecture Design and Validation Methods. Springer Verlag, Berlin Heidelberg New York 1999
47. Bowen, D. Personal communication. Palo Alto (5.11.1991)
48. Bowen, J.P.: Formal Specification and Documentation Using Z: A Case Study Approach. Int. Thomson Computer Press (1996)
49. Büchi, J.R.: Finite Automata, their Algebras and Grammars. Siefkes, D. (ed). Springer-Verlag (1988)
50. Chow, T.S.: Testing Software Design Modeled by Finite State Machines. IEEE Trans.Softw.Engineering 4(3) (1978) 178–187
51. Cremers, A.B.C., Hibbard, T.N.: Formal Modeling of Virtual Machines. IEEE Transactions on Software Engineering SE-4(5) (1987) 426–436
52. Dahl, O.: Discrete Event Simulation Languages. In: F. Genuys (ed): Programming Languages. Academic Press (1968) 349–395
53. Dahl, O., Dijkstra, E.,Hoare, C.: Structured Programming. Academic Press (1972)
54. Del Castillo, G.: Towards Comprehensive Tool Support for Abstract State Machines: The ASM Workbench Tool Environment and Architecture. This volume
55. Del Castillo, G., Hardt, W.: Fast Dynamic Analysis of Complex HW/SW-Systems based on Abstract State Machines. IEEE Proc. 6th. International Workshop on HW/SW Co-Design (CODES/CASHE'98). Washington (March 1998)
56. Dijkstra, E.W.: Structure of the T.H.E. Multiprogrammming System. Communications of ACM 11 (1968) 341–346
57. Dijkstra, E.W.: Notes on Structured Programming. In: Structured Programming. Academic Press, New York (1972) 1–82

58. Dold, A.: A Formal Representation of Abstract State Machines using PVS. Verifix Report Ulm/6.2 (July 1998)1–25
59. Eilenberg, S.: Automata, Languages and Machines. Vol.A. Academic Press (1974)
60. Fitzgerald, J., Gorm Larsen, P.: Modelling Systems. Practical Tools and Techniques in Software Development. Cambridge University Press (1998)
61. Fuchs, N.E.: Specifications are (Preferably) Executable.IEE/BCS Software Engineering Journal 7(5) (1992) 323–334
62. Gaudel, M.C.: Génération et Preuve de Compilateurs Basées sur une Sémantique Formelle des Langages de Programmation. Thèse, L'Institut National Polytechnique de Lorraine (1980)
63. Geist, A., Beguelin, A., Dongarra, J.,Jiang, W., Manchek, B., Sunderam, V.: PVM 3 User's Guide and Reference Manual. Technical Report ORNL/TM-12187. Oak Ridge National Laboratory, Oak Ridge/Tennessee 37831 (September 1994)
64. Glässer, U., Gotzhein, R.: Towards a New Formal SDL Semantics - Outline of an ASM Behavior Model. Submitted to 9th SDL Forum, Montreal/Quebec (21-25 June 1999) http://www.iro.umontreal.ca/SDL/
65. Glässer, U., Karges, R.: Abstract State Machines Semantics of SDL. J. of Universal Computer Science 3 (12) (1997) 1382–1414
66. Glässer, U., Prinz, A.: Abstract State Machines Semantics of SDL. Submitted (1999)
67. Giese, M., Kempe, D., Schönegge, A.: KIV zur Verifikation von ASM-Spezifikationen am Beispiel der DLX-Pipelining Architektur. University of Karlsruhe, Institute for Logic, Complexityy and Deduction Systems. Int. Rep. 16/97 (1997) 1–37
68. Goerigk, W., Hoffmann, U.: Rigorous Compiler Implementation Correctness: How to Prove the Real Thing Correct. This volume
69. Gosling, J., Joy, B., Steele, G.: The Java(tm) Language Specification. Addison Welsley (1996)
70. Grädel, E., Gurevich, Y.: Metafinite Model Theory. Lecture Notes in Computer Science, Vol. 960. Springer-Verlag, Berlin Heidelberg New York (1995) 313–366
71. Gurevich, Y: A New Thesis. Abstracts. American Mathematical Society (August 1985) 85T-68-203.
72. Gurevich, Y.: Algorithms in the World of Bounded Resources. In: Herken, R. (ed): The Universal Turing Machine–A Half-Century Story. Oxford University Press(1988) 407–416.
73. Gurevich, Y.: Logic and the Challenge of Computer Science. In: Börger, E. (ed): Current Trends in Theoretical Computer Science. Computer Science Press (1988) 1–57
74. Gurevich, Y.: Evolving Algebras: An Attempt to Discover Semantics. Bulletin EATCS 43 (1991) 264–284. Slightly revised in: Rozenberg, G., Salomaa, A. (eds): Current Trends in Theoretical Computer Science. World Scientific (1993) 274–308
75. Gurevich, Y.: Evolving Algebra 1993: Lipari Guide. In: Börger, E. (ed): Specification and Validation Methods. Oxford University Press (1995) 9–36
76. Gurevich, Y.: The Sequential ASM Thesis. Bulletin of the EATCS (February 1999)
77. Hayes, I.J., Jones, C.B.: Specifications are not (Necessarily) Executable. IEE/BCS Software Engineering Journal 4(6) (1989) 330–338
78. Heberle, A., Löwe, W., Trapp, M.: Safe Reuse of Source to Intermediate Language Compilations. In: Chillarege, R. (ed): Proc. 9th. Int. Symp. on Software Reliability Engineering (1998) http://www.chillarege.com/issre/fastabstracts/98417.html

79. Hinrichsen, H.: Formally Correct Construction of a Pipelined DLX Architecture. Darmstad University of Technology, Dept. of Electrical and Computer Engineering. TR 98-5-1 (1998)
80. Hoare, C.A.R.: The Structure of an Operating System. Lecture Notes in Computer Science, Vol. 46. Springer-Verlag, Berlin Heidelberg New York (1976) 242–265
81. Hoare, C.A.R.: Mathematical Models for Computing Science. Manuscript (August 1994) 1–65
82. Holcombe, M.: X-Machines as a Basis for Dynamic System Specification. Software Engineering Journal 3(2) (1988) 69–76
83. Holcombe, M., Ipate, F.: Correct Systems. Springer-Verlag, Berlin Heidelberg New York (1998)
84. Horning, J., Randell, B.: Process Structuring. Computing Surveys 5 (1973) 5–30
85. Huggins, J. K., Van Campenhout, D.: Specification and Verification of Pipelining in the ARM2 RISC Microprocessor. ACM Transactions on Design Automation of Electronic Systems 3 (4) (October 1998)
86. IEEE Standard VHDL Language Reference Manual—IEEE Std 1076–1993. The Institute of Electrical and Electronics Engineering. New York/NY (1994)
87. Janicki, R.: On a Formal Semantics of Tabular Expressions. Faculty of Engineering, McMaster University. Hamilton/Ontario. CRL Report 355 (1997). Short version in: Towards a Formal Semantics of Parnas Tables. ACM/IEEE Proceedings of the 17th International Conference on Software Engineering, Seattle (April 1995) 231–240
88. Janicki, R., Parnas, D.L., Zucker, J.I.: Tabular Representations in Relational Documents. In: Brink, C., Kahl, W., Schmidt, G.(eds): Relational Methods in Computer Science. Springer-Verlag, Berlin Heidelberg New York (1997) 184–196
89. Kutter, P.W., Schweitzer, D., Thiele, L.: Integrating Formal Domain Specific Language Design in the Software Life Cycle. This volume
90. Kwon, K.: A Structured Presentation of a Closure-Based Compilation Method for a Scoping Notion in Logic Programming. J. of Universal Computer Science, Special ASM Issue, 3(4)(1997) 341–376
91. Lavagno, L., Sangiovanni-Vincentelli, A., Sentovich, E.M.: Models of Computation for System Design. In: Börger, E. (ed): Architecture Design and Validation Methods. Springer- Verlag, Berlin Heidelberg New York (1999)
92. Laycock, G.T.: The Theory and Practice of Specification Based Testing. PH.D. Thesis, University of Sheffield, UK (1992)
93. Lötzbeyer, A.: Task Description of a Fault-Tolerant Production Cell. FZI, University of Karlsruhe/Germany. Version 1.6 (June 17, 1996)
94. Lötzbeyer, A., Mühlfeld, R.: Task Description of a Flexible Production Cell with Real Time Properties. FZI, University of Karlsruhe and Siemens ZT SE 1/Germany. Version 2.1 (July 30, 1996)
95. Mearelli,L.: Refining an ASM Specification of the Production Cell to C^{++} Code. J. of Universal Computer Science, Special ASM Issue 3(5) (1997) 666–688
96. Marcotty, M., Ledgard, H.F.: The World of Programming Languages. Springer-Verlag, Berlin Heidelberg New York (1986)
97. Van der Meulen, M., Clement, T.: Formal Methods in the Specification of the Emergency Closing System of the Eastern Scheldt Storm Surge Barrier. This volume
98. Mosses, P.D.: Denotational Semantics. In: Van Leeuwen, J.(ed.): Handbook of TCS. Elsevier (1990)
99. Mosses, P.D.: Action Semantics. Cambridge University Press (1992)

100. Nielson, H.R., Nielson, F.: Semantics with Applications. Wiley (1992)
101. Pair, C.: Types Abstraits et Sémantique Algébrique des Langages de Programmation. Centre de Recherche en Informatique de Nancy. TR 80-R-011 (February/July 1980) 1–46
102. Päppinghaus, P., Börger, E.: Industrial Use of ASMs for System Documentation. In: Jähnichen, S., Loeckx, J., Wirsing, M. (eds): Logic for System Engineering. Dagstuhl Seminar Report 171 (9710) 26
103. Parnas, D.L.: Information Distribution Aspects of Design Methodology. In: Freiman, C.V. (ed): Proc. of IFIP Congress 1971. Volume 1: Foundations and Systems. North-Holland (1972) 339–344
104. Parnas, D.L.: Personal communication (1997)
105. Parnas, D.L.: "Formal Methods" Technology Transfer Will Fail. J. Systems Software 40 (1998) 195–198
106. Parnas, D.L., Darringer, J.: SODAS and a Methodology for System Design. Proc. AFIPS Fall Joint Conf. Vol.31. Academic Press (1967) 449–474
107. Parnas, D.L., Madey, J.: Functional Documents for Computer Systems. Science of Computer Programming 25 (1995) 41–62
108. Pippenger, N.: Pure versus Impure Lisp. ASM Transactions on Programming Languages and Systems 19 (1997) 223–238
109. Popper, K.: Logik der Forschung (1935)
110. Potet, M.-L., Rouzaud, Y.: Composition and Refinement in the B-Method. In: Bert, D. (ed): B'98: Recent Advances in the Development and Use of the B Method. Lecture Notes in Computer Science, Vol. 1393. Springer-Verlag, Berlin Heidelberg New York (1998) 46-65
111. Pusch, C.: Verification of Compiler Correctness for the WAM. In: Von Wright, J., Grundy, J., Harrison, J.(eds): Theorem Proving in Higher Order Logics (TPHOLs'96). Lecture Notes in Computer Science, Vol. 1125. Springer-Verlag, Berlin Heidelberg New York (1996) 347–362
112. Ramadge, P.J.G., Wonham, W.M.: The Control of Discrete Event Systems. Proc. of the IEEE 77(1) (1989) 81–98
113. Reisig, W.: Elements of Distributed Algorithms. Modeling and Analysis with Petri Nets. Springer-Verlag, Berlin Heidelberg New York (1998)
114. Rischel, H., Sun, H.: Design and Prototyping of Real-Time Systems using CSP and CML. 9th Euromicro Workshop on Teal-Time Systems. Toledo (June 11-13, 1997)
115. Schellhorn, G.: Verifikation abstrakter Zustandsmaschinen. PhD Thesis. University of Ulm (1999)
116. Schellhorn, G., Ahrendt, W.: Reasoning about Abstract State Machines: The WAM Case Study. J. of Universal Computer Science. Special ASM Issue, 3(4) (1997) 377–413
117. Schwank, I.: Zur Konzeption prädikativer versus funktionaler kognitiver Strukturen und ihrer Anwendung. Zentralblatt für Didaktik der Mathematik 6 (1996) 168–183
118. Scott, D.: Some Definitional Suggestions for Automata Theory. J. of Computer and System Sciences 1 (1967) 187–212
119. Shaw, M.: The Impact of Abstraction Concerns on Modern Programming Languages. Proc. IEEE 68 (9) (1980)
120. Shaw, M., Garlan, D.: Formulations and Formalisms in Software Architecture. In: van Leeuwen, J. (ed): Computer Science Today: Recent Trends and Developments. Springer-Verlag, Berlin Heidelberg New York (1995) 307–323

121. Sommerville, I.: Software Engineering. Addison-Wesley (1992)
122. Soni, D., North, R.L., Hofmeister, C.: Software Architecture in Industrial Applications. Proc. 17th ACM Conf. Sw. Engg.. Seattle (1995)
123. Stärk, R.: The Theoretical Foundations of LPTP (A Logic Program Theorem Prover). The Journal of Logic Programming 36 (1998) 241–269
124. Stegmüller, M.M.: Formale Verifikation des DLX RISC-Prozessors: Eine Fallstudie basierend auf abstrakten Zustandsmaschinen. Diploma Thesis, University of Ulm (1998)
125. Stenzel, K., Haneberg, D.: Personal communication. KIV Group, University of Ulm/Germany
126. Tanenbaum, A.S.: Structured Computer Organization (1976)
127. Tarski, A.: Der Wahrheitsbegriff in den formalisierten Sprachen. Studia Philosophica 1 (1936) 261–405
128. Thue, A.: Probleme über Veränderungen von Zeichenreihen nach gegebenen Regeln. Skr.Videnks.Sels I (10)1–34
129. Turing, A.: On computable numbers, with an application to the Entscheidungsproblem. Proc. London Math. Soc. (2) 42 (1937) 230–265
130. UML—Unified Modeling Language. Rational Software Corporation. Updates via the worldwide web (http://www.rational.com)
131. Goerigk, W., Dold, A., Gaul, T., Goos, G., Heberle, A., von Henke, F. W., Hoffmann, U., Langmaack, H., Pfeifer, H., Ruess, H., Zimmermann, W.: Compiler Correctness and Implementation Verification: The *Verifix* Approach. In: Fritzson, P.(ed): Proceedings of the Poster Session of CC'96— International Conference on Compiler Construction. IDA Technical Report LiTH-IDA-R-96-12. Linköping/Sweden (1996)
132. Winter, K.: Model Checking for Abstract State Machines. J. of Universal Computer Science. Special ASM Issue, 3(5) (1997)
133. Wirsing, M.: Handbook of Algebraic Specifications. In: van Leeuwen, J. (ed): Handbook of Theoretical Computer Science B. Elsevier (1990) 675–788
134. Wirth, N.: Program Development by Stepwise Refinement. Comm. ACM 14 (4) (1971)
135. Wirth, N.: Algorithms & Data Structures. Prentice-Hall (1975)
136. Zimmermann, W., Gaul, T.: On the Construction of Correct Compiler Back-Ends: An ASM Approach. J. of Universal Computer Science. Special ASM Issue, 3(5) (1997) 504–567
137. Zurcher, F.W., Randell, B.: Iterative Multi-Level Modelling–A Methdology for Computer System Design. Proc. IFIP Congress 1968. North-Holland, Amsterdam (1968) 867–871

PS. The paper is going to be published in: Hutter, D., Stephan, W., Traverso, P., Ullmann, M. (eds): Current Trends in Applied Formal Methods (FM-Trends 98). Lecture Notes in Computer Science, Vol. 1641, pp. 1-43. Springer-Verlag, Berlin Heidelberg New York (1999)