Under consideration for publication in Formal Aspects of Computing

The ASM Refinement Method

Egon Börger (Università di Pisa, Italy, boerger@di.unipi.it)

Abstract. In this paper the Abstract State Machine refinement method is presented. Its characteristics compared to other refinement approaches in the literature are explained. Some frequently occurring forms of ASM refinements are identified and illustrated by examples from the design and verification of architectures and protocols, from the semantics and the implementation of programming languages and from requirements engineering.

1. Introduction

Refinement is a general methodological principle which is present wherever a complex system or problem is described piecemeal, decomposing it into constituent parts which are detailed in steps to become manageable. Refinement goes together with the inverse process of abstraction which characterizes mathematics since ancient times. In software engineering stepwise refinement in one way or the other underlies all top-down approaches. Not surprisingly it also played a central role for incremental program development and verification in the structural programming endeavor (see [82, 48, 45]) and since then has been studied intensively for specification and verification approaches which are based on algebraic, set-theoretic or logical methods, as is well documented in numerous books, e.g. [83, 74, 46, 53, 47]. Exploiting the generality of abstraction offered by the notion of Abstract State Machines (ASMs) [57] and guided by problems of practical system design, the ASM ground model method has been developed in [10, 11, 12, 13] together with a rather general refinement scheme for ASMs. Through further development in [36, 26, 7, 6, 22, 28, 29, 40, 79], ASM *ground model construction* and *stepwise refinement* became together with the *definition of ASMs* in [58] the three constituents of the ASM method for practical system design and analysis (see the AsmBook [42] for a systematic introduction and [16] for historical details). In the ASM method, stepwise refinement appears as a practical method for crossing levels of abstraction to link ASM models through well-documented incremental development steps, starting from ground models and turning them piecemeal into executable code.

In this paper we explain the ASM refinement method and characterize it with respect to other refinement approaches in the literature. We show that the ASM refinement method provides a kind of meta-framework which integrates well-known more specific notions of refinement (see [73, 64, 67, 74, 46, 47]), similarly to the way the notion of ASMs covers well-known models of computation and approaches to system design [17]. For the readers who are not familiar with ASMs we provide in Section 2 a summary of the few basic definitions which are needed for a technical understanding. In Section 3 we define the general notion of ASM refinement. In Section 4 we illustrate the ASM refinement concept by some practically useful specializations, namely conservative extension, procedural (submachine) refinement, pure data refinement, and instantiation. In Section 5 we present a general scheme for proving the correctness of refinements which can be used to establish complex system properties by proving them in an abstract model which is correctly refined to the considered system. In the Conclusion we briefly review some main usages of ASM refinements in practical

Correspondence and offprint requests to: **Egon Börger** (Università di Pisa, Italy, boerger@di.unipi.it) system design and analysis. The focus in this survey paper is on an illustration of principles and usage of ASM refinements, capturing various forms of refinement within one conceptual framework, rather than on a formalized technical development. For the illustrations we use a variety of examples from different application domains to document the wide range spanned by the refinement approach.

2. The Notion of Abstract State Machines

In this section we recall the basic definitions concerning ASMs. For a more detailed definition of these terms we refer the interested reader to Section 2.4 of the AsmBook [42].

2.1. Basic ASMs and their Runs

An ASM is a finite set of so called *transition rules* of form

if $Condition$ **then** $Updates$

which transform abstract states. Two more forms are explained below. The $Condition$ (also called *guard*) under which a rule is applied is an arbitrary predicate logic formula without free variables, whose interpretation evaluates to $true$ or $false$. $Updates$ is a finite set of assignments of form

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

whose execution is to be understood as *changing* (or defining, if there was none) in parallel the value of the occurring functions f at the indicated arguments to the indicated value. More precisely, in the given state first all parameters t_i, t are evaluated to their values, say v_i, v, then the value of $f(v_1, \dots, v_n)$ is updated to v which represents the value of $f(v_1, \dots, v_n)$ in the next state. Such pairs of a function name f, which is fixed by the signature, and an optional argument $(v_1, \dots, v_n)$, which is formed by a list of dynamic parameter values v_i of whatever type, are called *locations*. Location-value pairs (loc, v) are called *updates*.

The notion of ASM *states* is the classical notion of mathematical *structures* where data come as abstract objects, i.e., as elements of sets (also called *domains* or *universes*, one for each category of data) which are equipped with basic operations (partial *functions* in the mathematical sense) and predicates (attributes or relations). For the evaluation of terms and formulae in an ASM state, the standard interpretation of function symbols by the corresponding functions in that state is used. Without loss of generality we usually treat predicates as characteristic functions and constants as 0-ary functions. Partial functions are turned into total functions by interpreting $f(x) = undef$ with a fixed special value $undef$ as $f(x)$ being undefined.

The notion of ASM *run* is an instance of the classical notion of computation of transition systems. An ASM computation step in a given state consists in executing *simultaneously* all updates of all transition rules whose guard is true in the state, if these updates are consistent, in which case the result of their execution yields the *next* state. In the case of inconsistency the computation does not yield a next state, a situation which typically is reported by executing engines with an error message. A set of updates is called *consistent* if it contains no pair of updates with the same location, i.e. no two elements $(loc, v), (loc, v')$ with $v \neq v'$. An ASM step is performed as an atomic action with no side effects.

Simultaneous execution provides means to locally describe a global state change, namely as obtained in one step through executing a set of updates. The only limitation—imposed by the need of uniquely identifying objects residing in locations—is the consistency of the set of to be executed updates. The local description of global state changes also implies that by definition the next state differs from the previous state only at locations which appear in the update set. Simultaneous execution also provides a convenient way to abstract from sequentiality where it is irrelevant for the investigation. This synchronous parallelism in the ASM execution model is enhanced by the following notation to express the simultaneous execution of a rule R for each x satisfying a given condition φ (where typically x will have some free occurrences in R which are bound by the quantifier):

```
forall $x$ with $\varphi$
  $R$
```

Similarly non-determinism as a convenient way to abstract from details of scheduling of rule executions can be expressed by rules of the form

```
choose $x$ with $\varphi$
    $R$
```

where φ is a Boolean valued expression and R a rule. The meaning of such an ASM rule is to execute rule R with an arbitrary x chosen among those satisfying the selection property φ. If there exists no such x, nothing is done

Common notations like **where**, **let**, **if-then-else** are used without further explanation since they are easily reducible to the above basic definitions. Sometimes we also use the table-like **case** notation with pattern matching, in which case we try out the cases in the order of writing, from top to bottom. We also use rule schemes, namely rules with variables, and named parameterized rules, mainly as an abbreviational device to enhance the readability or as macro allowing us to reuse machines and to display a global machine structure. For example

if $\ldots a = (x, y) \ldots$ **then** $\ldots x \ldots y \ldots$

abbreviates

if $\ldots ispair(a) \ldots$ **then** $\ldots fst(a) \ldots snd(a) \ldots$

sparing us the need to write explicitly the recognizers and the selectors. Similarly, an occurrence of

$r(x_1, \ldots, x_n)$

where a rule is expected stands for the corresponding rule R (which is supposed to be defined somewhere else, with $r(x_1, \ldots, x_n) = R$ appearing in the declaration part of the ASM where $r(x_1, \ldots, x_n)$ is used). When such a “rule call” $r(x_1, \ldots, x_n)$ is used, the parameters have to be instantiated by legal values (objects, functions, rules, whatever) so that the resulting rule has a well defined semantical meaning on the basis of the explanations given above. A precise semantical definition of such ASM submachine calls has been defined in [38].

For purposes of separation of concerns it is often convenient to impose for a given ASM additional constraints on its runs to circumscribe those one wants to consider as *legal.* Logically speaking this means to restrict the class of models satisfying the given specification. Such restrictions are particularly useful if the constraints express reasoning assumptions for a high-level machine which are easily shown to hold in a refined target machine. In general ASMs are reactive systems which iterate their computation step, but for the special case of terminating runs one can choose among various natural termination criteria to constrain runs, namely that no rule is applicable any more or that the machine yields an empty update set or that the state does not change any more.

2.2. Classification of Locations and Functions

In an ASM, a priori no restriction is imposed neither on the abstraction level nor on the complexity nor on the means of definition of the functions used to compute the arguments and the new value denoted by t_i, t in function updates. In support of the principles of separation of concerns, information hiding, data abstraction, modularization and stepwise refinement, the ASM method exploits however the following distinctions reflecting the different roles these functions (and more generally locations) can assume in a given machine.

The major distinction for a given ASM M is between its *static* functions—which never change during any run of M so that their values for given arguments do not depend on states of M—and *dynamic* ones which may change as a consequence of updates by M or by the environment (read: by some other—say an unknown—agent representing the context in which M computes), so that their values for given arguments may depend on states of M. By definition static functions can be thought of as given by the initial state, so that where appropriate, handling them can be clearly separated from the description of the system dynamics. Whether the meaning of these functions is determined by a mere signature (“interface”) description, or by axiomatic constraints, or by an abstract specification, or by an explicit or recursive definition, or by a program module, depends on the degree of information hiding the specifier wants to realize. Static 0-ary functions represent constants, whereas with dynamic 0-ary functions one can model variables of programming (not to confuse with logical variables). Dynamic functions can be thought of as a generalization of array variables or hash tables.

The dynamic functions are further divided into four subclasses. *Controlled* functions (for M) are dynamic functions which are directly updatable by and only by the rules of M, i.e., functions f which appear in at least one rule of M as leftmost function (namely in an update $f(s) := t$ for some s, t) and are not updatable by the environment (or more generally by another agent in the case of a multi-agent machine). These functions are the ones which constitute the internally controlled part of the dynamic state of M.

Monitored functions, also called *in* functions, are dynamic functions which are read but not updated by M and directly updatable only by the environment (or more generally by other agents). They appear in updates of M, but not as leftmost function of an update. These monitored functions constitute the externally controlled part of the dynamic state of M. To describe combinations of internal and external control of functions, one can use *interaction* functions, also called *shared* functions, defined as dynamic functions which are directly updatable by rules of M and by the environment and can be read by both (so that typically a protocol is needed to guarantee consistency of updates). The concepts of monitored and shared functions allow one to separate in a specification computation from communication concerns. In fact the definition does not commit to any particular mechanism (e.g. message passing via channels) to describe the exchange of information between an agent and its environment (and similarly between arbitrary agents in the case of a multi-agent machine). As with static functions the specification of monitored functions is open to any appropriate method. The only assumption made is that in a given state, the values of all monitored functions are determined.

Out functions are dynamic functions which are updated but not read by M and are monitored (read but not updated) by the environment or in general by other agents. Formally, such output functions do appear in some rules of M, but only as leftmost function of an assignment.

Functions are called *external* for M if for M they are either static or monitored.

An orthogonal, pragmatically important classification comes through the distinction of *basic* and of *derived* functions. Basic functions are functions which are taken for granted (declared as “given”, typically those forming the basic signature); derived functions are functions which even if dynamic are not updatable neither by M nor by the environment but may be read by both and yield values which are defined by a fixed scheme in terms of other (static or dynamic) functions (and as a consequence may sometimes not be counted as part of the basic signature). Thus derived functions are sort of auxiliary functions coming with a specification or computation mechanism which is given separately from the main machine; they may be thought of as a global method with read-only variables.

The same classification principle is applied to (sets of) locations or updates.

2.3. Multi-Agent ASMs

A *multi-agent ASM* is defined as a set of agents which execute each its own basic ASM. This may happen in a synchronous or in an asynchronous manner. In a *synchronous ASM* the agents execute their basic ASM in parallel, synchronized using an implicit global system clock. Semantically a synchronous ASM is equivalent to the set of all its constituent single-agent ASMs, operating in the global states over the union of the signatures of each component, though each agent is equipped with its own set of states and rules. This concept allows one to define and analyze the interaction between components using precise interfaces over common locations.

A problem one has to solve for runs of asynchronously cooperating agents originates in the possible incomparability of their moves which may come with different data, clocks, moments and duration of execution, making it difficult to uniquely define a global state where moves are executed to locate changes of monitored functions in an ordering of moves. A coherence condition in the definition of asynchronous multi-agent ASM runs given in [58] postulates well-definedness for a relevant portion of state in which an agent is supposed to perform a step, thus providing a notion of ‘local’ stable view of ‘the’ state in which an agent makes a move. The underlying synchronization scheme is described using partial orders for moves of different agents which reflect causal dependencies, determining which move depends upon (and thus must come ‘before’) which other move. This synchronization scheme is as liberal as it can be, restricted only by the consistency condition for the updates which is logically indispensable, and thus can be instantiated by any consistent synchronization mechanism.

Formally a run of an asynchronous ASM, also called *partial order run*, is defined as a partially ordered set $(M, <)$ of *moves* m (read: rule applications) of its agents satisfying the following conditions:

finite history: each move has only finitely many predecessors, i.e. for each $m \in M$ the set $\{m' | m' < m\}$ is finite,

sequentiality of agents: the set of moves $\{m | m \in M, a \text{ performs } m\}$ of every agent $a \in Agent$ is linearly ordered by $<$,

coherence: each finite initial segment (downward closed subset) X of $(M, <)$ has an associated state $\sigma(X)$—think of it as the result of all moves in X with m executed before m' if $m < m'$—which for every maximal element $m \in X$ is the result of applying move m in state $\sigma(X - \{m\})$.

The coherence condition immediately implies for every finite initial segment X of a run of an async ASM, that all linearizations of X yield runs with the same final state. The definition provides no clue to construct partial order runs for an async ASM, but it makes the freedom explicit one has in implementing the described causal dependencies of certain local actions of otherwise independent agents. Notably the definition imposes no fairness condition on runs.

3. Definition of ASM Refinements

In this section we define the notion of ASM refinement and its frequently used and practically important specializations to conservative extension, procedural (submachine) refinement, pure data refinement and instantiation.

Most established refinement notions are based upon some a priori principle. An example is the substitutivity principle which is usually expressed as follows, quoted from [47, pg. 47]:
"*Principle of substitutivity*: it is acceptable to replace one program by another, *provided* it is impossible for a user of the programs to observe that the substitution has taken place."

Refinement notions which are tailored to match such an a priori epistemological principle as a result may be restricted in various ways, limiting their range of applicability. Among the restrictions of this sort are the following ones one can find in one form or the other in [73, 83, 1, 46, 74, 53, 47]:

- Restriction to *certain forms of programs*, often viewed as sequences of operations (straight-line programs). As a consequence programs which are refined in this way are even structurally equivalent to their abstract counterpart, i.e. with corresponding operations occurring in the same places, thus precluding to analyze the role of other forms of control for refinement, e.g. various forms of parallelism or iteration (as encountered for example in some process algebraic refinement notions).
- Restriction to programs with *only monolithic state operations*, expressed by global functions on the state without possibility to modify elements of the state. This makes it difficult to exploit combinations of local effects for overall refinements and leads to the well-known frame problem, which typically makes formal specifications of programs more difficult to write and to read than the programs that they describe.
- Restriction to *observations interpreted as pairs of input/output sequences or of pre-post-states*, often with the same input/output representation at the abstract and the refined level. Such a focus on functional input/output behavior of terminating runs or on pre-post-states of data refined operations precludes to relate arbitrary segments of abstract and refined computations. As a consequence, to be compared invariants of abstract and refined programs are viewed in terms of pre- or post-condition strengthenings or weakenings, which restricts a more general analysis of the effect of invariants as retrenchment of the class of possible models. The fact that often no change of input/output representation is permitted also precludes the possibility to refine 'abstract input', e.g. input coming in the form of monitored data, by 'controlled data' which are computed through concrete computation steps.
- Restriction to *logic or proof-rule oriented* refinement schemes. Tailoring refinement schemes to fit a priori fixed proof principles quickly leads to severe restrictions of the design space.

In contrast, the ASM refinement method is not based upon any concrete principle, but it is rooted in mathematics (which includes formalized logic as a tiny fraction). One can make up the notion of refinement one needs, adapting to what the investigated system levels (read: the chosen levels of abstraction) demand for comparing program runs. This openness does not imply that the designer is left with an insecure footing, since the confidence given to refinements of ASM specifications to implementations goes together with the degree of mathematical precision with which the refinement is provided. The range of mathematical rigor is by far more comprehensive than that of any formalized theory of logic.

$$\begin{tikzcd}
\text{State } S \arrow[r, "\overbrace{\tau_1 \cdots \tau_m}^{m \text{ steps of } M}"] \arrow[d, leftrightarrow, "\equiv"'] & S' \arrow[d, leftrightarrow, "\equiv"] \\
\text{State } S^* \arrow[r, "\underbrace{\sigma_1 \cdots \sigma_n}_{n \text{ steps of } M^*}"'] & S^{*\prime}
\end{tikzcd}$$

$\equiv$ is an equivalence notion between data
in locations of interest in corresponding states.

Fig. 1. The ASM refinement scheme.

The problem-orientation of the ASM refinement method has shaped its development during the last decade which was driven by practical refinement tasks and geared to support divide-and-conquer techniques for both design and verification, without privileging one to the detriment of the other. What for short we call 'freedom of abstraction' offered by ASMs, i.e. the availability in ASMs of arbitrary structures to reflect the underlying notion of state, provides the necessary instrument to fine tune the mapping of a given (the "abstract") machine to a more concrete (the "refined") one, with its observable (typically more detailed) state and its observable (typically more involved) computation, in such a way that the intended "equivalence" between corresponding run segments of the two ASMs becomes observable, whereby we mean that it can be explicitly defined and proved to hold under precisely stated boundary conditions.

The focus is not on generic notions of refinements which can be proved to work in every context and to provide only effects which can never be detected by any user of the new program. Instead the concern is to support a disciplined use of refinements which correctly reflect and explicitly document an intended design decision, adding more details to a more abstract design description, e.g. for making an abstract program executable, for improving a program by additional features or by restricting it through precise boundary conditions which exclude certain undesired behaviors. Exploiting the freedom of abstraction offered by ASMs one has the possibility not to let oneself get bound by an a priori commitment to particular notions of state, program, run, equivalence, or to any particular method to establish the correctness of the refinement step. The major and usually difficult task is to first listen to the subject, to find the right granularity and to formulate an appropriate refinement—or abstraction in case of a reengineering project—that faithfully reflects the underlying design decision or reengineering idea, and only then to look for appropriate means to justify that under the precisely stated conditions the refinement correctly implements the given model, respectively that the reengineered abstract model correctly abstracts from the given code. With the ASM refinement method, whatever feasible accurate method is out there can—indeed should—be adopted, whether for verification (by reasoning) or for validation (e.g. testing model-based runtime assertions through a simulation), to establish that the intended design assumptions hold in the implementation and that refined runs correctly translate the effect of abstract ones. In particular, by appropriate instantiations of the widely open ASM refinement concept one can capture the various more restricted refinement notions studied in the literature. In this sense the ASM refinement method provides a uniform framework to reflect established specific refinement notions.

These principles can be realized based upon the general scheme for an ASM refinement step which is illustrated by the familiar commutative diagram in Figure 1 and underlies the definition of ASM refinement below. The scheme can also be viewed as describing an abstraction step if it is used to model an implementation, as happens in reengineering projects, see [4] for an illustration by an industrial case study.

Refinement based upon this scheme generalizes in particular the standard notion of a simulation which is used to verify data refinements (see [46]). But it does more: to refine an ASM M to an ASM M^*, as designer one has the freedom (and the task) to define the following items:

- a notion of *refined state*,
- a notion of *states of interest* and of *correspondence* between M-states S and M^*-states S^* of interest, i.e. the pairs of states in the runs one wants to relate through the refinement, including usually the correspondence of initial and (if there are any) of final states,
- a notion of abstract *computation segments* $\tau_1, \ldots, \tau_m$, where each τ_i represents a single M-step, and of corresponding refined computation segments $\sigma_1, \ldots, \sigma_n$, of single M^*-steps σ_j, which in given runs lead from corresponding states of interest to (usually the next) corresponding states of interest (the resulting diagrams are called (m,n)-diagrams and the refinements (m,n)-refinements),
- a notion of *locations of interest* and of *corresponding locations*, i.e. pairs of (possibly sets of) locations one wants to relate in corresponding states, where locations represent abstract containers for data,
- a notion of *equivalence* $\equiv$ of the data in the locations of interest; these local data equivalences usually accumulate to a notion of equivalence of corresponding states of interest.

The scheme shows that an ASM refinement allows one to combine in a natural way a change of the signature (through the definition of states and of their correspondence, of corresponding locations and of the equivalence of data) with a change of the control (defining the "flow of operations" appearing in the corresponding computation segments). Many notions of refinement in the literature keep these two features on purpose separated, see for example the notions of data refinement in VDM [53] and of operation refinement in B [1]. Once the notions of corresponding states and of their equivalence have been determined, one can define that M^* is a correct refinement of M if and only if every (infinite) refined run simulates an (infinite) abstract run with equivalent corresponding states, as is made precise by the following definition. By this definition, refinement correctness implies for the special case of terminating runs the inclusion of the input/output behavior of the abstract and the refined machine, a special feature on which numerous refinement notions in the literature are focussed.
Definition Fix any notions $\equiv$ of equivalence of states and of initial and final states. An ASM M^* is called a *correct refinement* of an ASM M if and only if for each M^*-run $S_0^*, S_1^*, \ldots$ there is an M-run $S_0, S_1, \ldots$ and sequences $i_0 < i_1 < \ldots, j_0 < j_1 < \ldots$ such that $i_0 = j_0 = 0$ and $S_{i_k} \equiv S_{j_k}^*$ for each k and either

- both runs terminate and their final states are the last pair of equivalent states, or
- both runs and both sequences $i_0 < i_1 < \ldots$, $j_0 < j_1 < \ldots$ are infinite.

Often the M^*-run $S_0^*, S_1^*, \ldots$ is said to simulate the M-run $S_0, S_1, \ldots$. The states $S_{i_k}, S_{j_k}^*$ are the corresponding states of interest. They represent the end points of the corresponding computation segments (those of interest) in Figure 1, for which the equivalence is defined in terms of a relation between their corresponding locations (those of interest). Sometimes it is convenient to assume that terminating runs are extended to infinite sequences which become constant at the final state. We refer to Figure 1 when using the term (m,n)-refinement.

M^* is called a *complete refinement* of M if and only if M is a correct refinement of M^*.

It is easy to show that in this refinement definition, the sequences of corresponding states can be chosen to be minimal in the sense that between two sequence elements there are no other equivalent states, i.e. there are no $i_k < i < i_{k+1}, j_k < j < j_{k+1}$ such that $S_i \equiv S_j^*$.

The pairs of the initial and possibly of the final states by definition are pairs of corresponding states. Therefore refinement correctness and completeness imply for terminating runs the equivalence of any input/output behavior of the abstract and the refined machine that is defined in terms of initial and final states and of the refinement.

In the literature correct and complete refinements with respect to terminating runs considering only the input/output behavior appear under the name of bisimulation. Correct refinements with respect to terminating runs are known under the name of preservation of partial correctness, whereas preservation of total correctness is used for refinements adding to the correctness condition for terminating runs that every infinite refined run admits an infinite abstract run with an equivalent initial state. For details see [75].

The ASM literature surveyed in [16] is full of examples of the above definition. To convey an idea to the reader we conclude this section by explaining the refinement of an ASM model for the semantics of a high-level programming language to an ASM model for compiled code. As example we choose the comparison of runs of Java programs to equivalent runs of their compilation to bytecode on the Java Virtual Machine, in terms of ASM models $Java_E$ and JVM_E defined in [79] for the language and the virtual machine. The correspondence of states is defined inductively by an order preserving function σ which assigns to the n-th state of a Java program run an equivalent state $\sigma(n)$ of the run of the compiled program on the JVM. In defining the equivalence notion one has to identify the relevant locations in the states of the two machines, whose value evolution guarantees that the two runs yield the same result. The definition is the result not of an a priori given refinement scheme, but of a detailed analysis of how Java instructions are interpreted, how they are compiled, how the compiled code is interpreted, covering in particular an analysis of how Java values are data refined by JVM values.

For example, the following locations are required to have the same value in corresponding states: the current method, the class state, the global variables and the two heaps. The current positions in the programs must belong to the same phrase (read: boolean or non-boolean or instance creation expression, statement, value, normal or abrupt termination) and the two machines must be in the same execution phase for that phrase (entering or exiting). The values of intermediate results must be the same via some data refinement function which maps Java values to JVM operand stack values. The local variables and the corresponding registers must contain the same values. The return addresses from possibly nested subroutines must correspond to each other. The correctness of subroutine return addresses is the most delicate part in defining what it means that in corresponding runs of $Java_E$ and JVM_E, the Java method call frames are equivalent to the JVM stack.

To show the equivalence of corresponding states means to verify that the two machines when started on corresponding programs P and $compile(P)$, upon navigating through their code produce in corresponding method code segments the same values for (local, global, heap) variables and the same results of intermediate calculations, for the current method as well as for the still to be completed method calls. The entire section 14.1 in [79] is devoted to turning the refinement correctness statement into an accurate mathematical form, just to make a correctness proof possible. The proof itself has to investigate the 83 different cases which may happen when executing Java instructions ([79, pp.178-203]. No a priori scheme was helpful to establish the proof, instead we had to understand, model and investigate the behavior of Java and of bytecode—which the freedom of abstraction in ASMs allowed us to express directly in terms of ASMs, avoiding any formal overhead.

4. Frequent Refinement Types

We present and illustrate here some specializations to refinement patterns which turned out to be useful for practical system design and analysis. For a detailed technical comparison of various specializations of the above ASM refinement scheme to outstanding refinement notions in the literature we refer the reader to [75].

It is important for the practicability of ASM refinements that the size of m and n in (m, n)-refinements is allowed to dynamically depend on the state. An early example where n has no a priori bound and can be determined only dynamically appeared in a refinement step of Prolog to WAM code in [33, 34, 36] (see below). The ASMs for Lamport's mutual exclusion algorithm *Bakery* in [26] provide a case where n is fixed, but grows with the number of protocol members, or where n is finite but without a priori bound, depending on the execution time of the participating processes. The correctness proof of a Java-to-JVM compiler in [79, Sect.14.2] uses $(1, n)$-refinements with $0 \leq n \leq 3$ depending on the length of the computation which leads the JVM machine from one to its next state of interest (i.e. a state which corresponds to a state of the Java machine). In [41] the correctness proof for exception handling in Java/JVM uses (m, n)-refinements where m is determined by the number of Java statements jumped over during the search for the exception handler. Although by a theorem of Schellhorn [75, Th.12] every (m, n)-refinement with $n > 1$ can be reduced to $(m, 1)$-refinements, this is typically at the price of having more involved equivalence notions which may complicate the proofs. Practical experience shows that (m, n)-refinements with $n > 1$ and including $(m, 0), (0, n)$-steps support the feasibility of decomposing complex (global) actions into simpler (locally describable) ones whose behavior can be verified in practice. They also provide a handle to classify refinements. A comprehensive classification and study of the interesting forms of ASM refinement and of ways to compose them is still waiting to be undertaken. We list in the following some patterns which turned out to be useful for practical applications.

4.1. Conservative extension

Conservative extension, a purely incremental refinement analogous to conservative theory extensions in logic, is typically used to introduce new behavior in a modular fashion, like exception handling, robustness features, etc. To define a conservative extension of a given machine, one has to do the following:

- Define the *condition for the 'new case'* in which the extended machine should execute and the given machine either has no well-defined behavior or should be prevented from executing. For example to add error handling to a machine, this condition can be expressed by a shared Boolean-valued *error* function which is supposed to become true when an emergency has happened and to be reset by the added exception handling machine.
- Define the *new machine* with the desired additional behavior, in the example an exception handling machine which is executed in case an error has been thrown. The return from the 'new' to the 'old' machine, if there is any, is typically a result of the computation of the new machine.
- Add the new machine and *restrict the given machine to the 'old case'* by guarding it by the negation of the 'new case' condition, in the example $error \neq true$.

A real-life example which follows this rather frequent pattern is the refinement of the Java machine by a proven to be correct exception handling mechanism in [79, Ch.6] or the incorporation of a bytecode verifier machine into the JVM interpreter in [79, Ch.17]. As one reviewer pointed out, the use of 'extension' in LOTOS [8] seems to be similar to the conservative extension refinement described here.

4.2. Procedural refinement

Procedural refinement, also called submachine refinement, consists in replacing in a given machine one machine by another (usually more complex) machine. A characteristic example is the refinement of the Prolog ASM which uses an abstract function $unify$ to a machine which calls a submachine implementing a unification procedure [36]—the example mentioned above of a $(1, n)$-refinement where n can be determined only dynamically since it depends on the size of the to be unified terms. In fact such procedural $(1, n)$-refinements with $n > 1$ have their typical use in compiler verification when replacing a source code instruction by a chunk of target code; for numerous examples see [22, 21, 84, 60, 56]. A convenient way to hide the multiplicity of the steps of such a refining submachine is to use the concept of ASM submachines which has been defined in [38] and allows to "view" n submachine steps as one step of an overall (here the unrefined more abstract) computation. For a specialization of such submachine refinements to incorporate functional programming techniques see [18].

In general there is a difference between procedural refinement and the above discussed principle of substitutivity. Namely depending on the granularity of the submachine, in the refined machine one may be able to observe new features which cannot be observed in the abstract machine, although they are related by the refinement definition to the behavior of the high-level machine.

Procedural Refinement of Control State ASMs. A frequent special case of procedural refinement derives from the graph structure of control state ASMs, a generalization of Finite State Machines (FSMs) to a class of Abstract State Machines (ASMs) introduced in [14] and extensively used in the AsmBook [42]. They are defined as ASMs all of whose rules have the following form, where *ctl_state*s represent the so-called internal states of FSMs:

```
if $ctl\_state = i$ then
  if $cond_1$ then
    $rule_1$
    $ctl\_state := j_1$
        ...
  if $cond_n$ then
    $rule_n$
    $ctl\_state := j_n$
```

In a given control state i, these machines do nothing when no condition $cond_k$ is satisfied; otherwise for every $cond_k$ which is satsfied, $rule_k$ is executed and the control state changes to j_k so that usually the conditions are supposed or guaranteed to be disjoint to avoid conflicting updates which would stop the ASM computation. Control state ASMs represent a normal form for UML activity diagrams (see[19]) from where they inherit the graphical representation of control states by circles or by (possibly named) directed arcs, to visually distinguish the control-passing role of control states from that of the update actions concerning the underlying data structure which are expressed by the ASM rules inscribed into rectangles, separated from the rule guards written into rhombs. Control state ASMs thus offer to use arbitrarily complex parallel (synchronized) data structure manipulations below the main control structure of finite state machines. The overall FSM-control structure is reflected by the following notation:

```
$\textsc{Fsm}(i, \textbf{if } cond \textbf{ then } rule, j) =$
  $\textbf{if } ctl\_state = i \textbf{ and } cond \textbf{ then}$
    $rule$
    $ctl\_state := j$
```

Using this notation the control state ASM rule above becomes the set of rules $\textsc{Fsm}(i, \textbf{if } cond_k \textbf{ then } rule_k, j_k)$ for $k = 1, \ldots, n$.

The procedural refinement of control state ASMs which is suggested by the FSM-diagram notation consists in replacing a control state transition—a machine *rule* at a node with well-defined 'entries' i and 'exits' j—by a new submachine M with the same number of entries and exits, formally replacing $\textsc{Fsm}(i, rule, j)$ by $\textsc{Fsm}(i, M, j)$ (tacitly assuming the renaming of the entry/exit nodes of M to the given ones i, j which is taken care of by the diagram notation).

The reader should be aware that due to the synchronous parallelism of ASMs, in a procedural ASM refinement an action—a part of a parallel step, not limited to a single 'operation'—can be replaced by multiple parallel actions which are viewed as part of a new parallel step. To state it technically, a rule can be refined by finitely many other rules which are executed in parallel. We mention two examples from programming language semantics and from a debugger reengineering case study. In order to separate the treatment of the semantics of Java thread execution from thread scheduling, in [79, Ch.7.2] a machine ExecJavaThread is defined using a submachine ExecJava. This submachine is refined to the parallel composition of four modules $\textsc{ExecJava}_I$ dealing with imperative control constructs, $\textsc{ExecJava}_O$ dealing with object-oriented language features, $\textsc{ExecJava}_E$ describing exception handling, and $\textsc{ExecJava}_T$ specifying thread-related language constructs. This refinement allowed us to define the semantics instructionwise so that it can easily be extended to modify or add new language elements. An example from a reengineering case study appears in [4, 4.3] where the callback for loading modules in a control state debugger model is refined to first bind in parallel each of the shell's breakpoints to the module in question, and only then to call the debugee to continue. It is noteworthy that analysing this rather abstract high-level rule and the symmetric rule refinement for unloading of modules in the debugger object model, a mismatch was detected between the way loading and unloading of module callbacks was implemented.

Asynchronous Procedural Refinement of Atomic Actions. An example of a procedural refinement which implements an atomic action by multiple actions which are executed in an asynchronous manner is the refinement of the atomic communication between two processes by an asynchronous channel communication in Occam. Channels for a communication to take place require exactly one reader x (positioned to execute an instruction $c?v$ to read into $val(v, env(x))$ the value of channel say $bind(c, env(x)))$ and one writer y (positioned to execute an instruction $d!t$ to write $val(t, env(y))$ into the same channel $bind(d, env(y)))$. The following instantaneous channel communication rule

```
$\textsc{OccamCommunication} =$
  $\textbf{if } mode(x) = running \textbf{ and } instr(pos(x)) = c?v \textbf{ and}$
    $mode(y) = running \textbf{ and } instr(pos(y)) = d!t \textbf{ and}$
    $bind(c, env(x)) = bind(d, env(y))$
  $\textbf{then } \{val(v, env(x)) := val(t, env(y)),\ \text{proceed } x,\ \text{proceed } y\}$
  $\textbf{where } \text{proceed } z = (pos(z) := next(pos(z)))$
```

has been refined in [23] as follows by introducing a channel agent which establishes the communication once a *reader* and a *writer* have arrived independently recording their identity and variable respectively message.

```
$\textsc{In}(x, c, v) =$
  **if** $mode(x) = running$ **and** $instr(pos(x)) = c?v$ **then**
    put $x$ asleep at $next(pos(x))$
    $\{reader(bind(c, env(x)) := x, var(bind(c, env(x)) := v\}$
$\textsc{Out}(x, c, t) =$
  **if** $mode(x) = running$ **and** $instr(pos(x)) = c!t$ **then**
    put $x$ asleep at $next(pos(x))$
    $\{writer(bind(c, env(x)) := x, mssg(bind(c, env(x)) := val(t, env(x))\}$
$\textsc{Chan}(c) =$
  **if** $reader(c), writer(c) \neq nil$ **then**
    $val(var(c), env(reader(c))) := mssg(c)$
    {wake up $reader(c)$, wake up $writer(c)$, clear $c$}
**where**
  put $z$ asleep at $p = \{mode(z) := sleeping, pos(z) := p\}$
  wake up $z = (mode(z) := running)$
  clear $c = \{reader(c) := nil, writer(c) := nil\}$
```

A study of the role of timing constraints for proving the correctness of refinements of asynchronous ASMs with continuous time appears in [44], based upon the ASM models for Lamport's Bakery algorithm in [26] as a case study.

4.3. Data refinement

Many data refinements are given by $(1, 1)$-refinements where abstract states and rules are mapped to concrete ones in such a way that the effect of each concrete operation on concrete data types is the same as the effect of the corresponding abstract operation on abstract data types. Pure data refinements are the basis for numerous algebraic and set-theoretic refinement notions [46, 47], including those used in VDM [53], Z [83] and B [1]. Typically they are used there for corresponding operations with unchanged signature, tailored to provide 'unchanged' properties. For example forward simulation is used to carry over equivalence from pre-states to post-states, backward simulation to carry over equivalence from post-states to pre-states.

A frequently used special type of ASM data refinements which exploits the generalization of the concept of 'operation' to 'ASM rule' is provided by what is called *instantiation* where the ASM rules remain unchanged and only the abstract (mostly the external) functions and predicates occurring in them are specified further. Instantiation turned out to be rather useful for the transition in requirements engineering from a use case model with abstract (symbolic) rules to a model which assigns a state transformation meaning to the rule names. Numerous such examples which for reasons of space cannot be exposed here can be found in the AsmBook [42]. We provide here two small examples to illustrate ASM data refinements. In the first example we define an ASM to compute a backtracking scheme which captures the notion of tree generation and traversal in such a way that applying to this machine appropriate data refinements yields well-known logic and functional programming patterns and generative grammars (context free and attribute grammars). In the second more theoretical example we show how classical automata models can be obtained by specialization of the predicates, functions and updates which constitute Turing-like ASMs.

Tree Traversal. The BACKTRACK machine below dynamically constructs a tree of alternatives and controls its traversal. When its *mode* is *ramify*, the submachine RAMIFY creates as many new children nodes to be computation *candidates* for its *currnode* as there are computation *alternatives*, provides them with the necessary *env*ironment and switches to *select*ion mode. In $mode = select$, if at *currnode* there is no more candidate the machine BACKtracks, otherwise the submachine SELECT lets the control move to TRYNEXTCANDIDATE to get *execute*d. The external function *alternatives* determines the solution space depending upon its parameters and possibly the current state. The dynamic function *env* records the information every new node needs to carry out the computation determined by the alternative it is associated with. The macro BACK moves *currnode* one step up in the tree, to *parent*(*currnode*), until the *root* is reached where the computation stops. TRYNEXTCANDIDATE moves *currnode* one step down in the tree to the *next* candidate, where *next* is a possibly dynamic choice function which determines the order for trying out the alternatives. Typically the underlying execution machine EXECUTE will update *mode* from *execute* to *ramify*, in case of a successful execution, or to *select* if the execution fails. This machine for tree computations is summarized by the following definition.

```
$\textsc{Backtrack} = \{\textsc{Ramify}, \textsc{Select}\}$ where
$\textsc{Ramify} =$
  if $mode = ramify$ then
    let $k = |alternatives(Params)|$
    let $o_1, \ldots, o_k = new(NODE)$
      $candidates(currnode) := \{o_1, \ldots, o_k\}$
      forall $1 \leq i \leq k$
        $parent(o_i) := currnode$
        $env(o_i) := i\text{-th}(alternatives(Params))$
      $mode := select$
  $\textsc{Select} =$
    if $mode = select$ then
      if $candidates(currnode) = \emptyset$
        then $\textsc{Back}$
        else
          $\textsc{TryNextCandidate}$
          $mode := execute$
  $\textsc{Back} =$
    if $parent(currnode) = root$
      then $mode := Stop$
      else $currnode := parent(currnode)$
  $\textsc{TryNextCandidate} =$
    $currnode := next(candidates(currnode))$
    $\textsc{Delete}(next(candidates(currnode)), currnode)$
```

We show now that by data refinements BACKTRACK can be turned into the backtracking engine for the core of ISO Prolog [20], of IBM's constraint logic programming language CLP(R) [37], of the functional programming language Babel [27], and of context free and of attribute grammars [65].

Data Refinement to Logic Programming Engine. BACKTRACK can be data refined to the backtracking engine for Prolog by instantiating the function *alternatives* to the function $procdef(stm, pgm)$. This is a Prolog specific function which yields the sequence of clauses in *pgm* to be tried out in this order to execute the current goal *stm*; these clauses come together with the needed state information from *currnode*. We determine *next* as *head* function on sequences, reflecting the depth-first left-to-right tree traversal strategy of ISO Prolog. It remains to add the execution engine for Prolog specified as ASM in [20, 35], which switches *mode* to *ramify* if the current resolution step succeeds and otherwise switches *mode* to *select*.

The backtracking engine for CLP(R) is the same, one only has to extend *procdef* by an additional parameter for the current set of *constraints* for the indexing mechanism and to add the CLP(R) engine specified as ASM in [37].

The functional language Babel uses the same function *next*, whereas the function *alternatives* is instantiated to $fundef(currexp, pgm)$ yielding the list of defining rules provided in *pgm* for the outer function of *currexp*. The Babel execution engine specified as ASM in [27] applies the defining rules in the given order to reduce *currexp* to normal form (using narrowing, a combination of unification and reduction).

Data Refinement to Context-Free and Attribute Grammars. To instantiate BACKTRACK for context free grammars G generating leftmost derivations we define the function $alternatives(currnode, G)$ to yield the sequence of symbols $Y_1, \ldots, Y_k$ of the conclusion of a G-rule whose premise X labels *currnode*, so that *env* records the label of a node, either a variable X or terminal letter a. The definition of *alternatives* includes a choice between different rules $X \rightarrow w$ in G. For leftmost derivations *next* is defined as for Prolog. As machine in $mode = execute$ one can add the following rule. For nodes labeled by a variable it triggers further tree expansion, for terminal nodes it extracts the yield (concatenating the terminal letter to the word generated so far) and moves the control to the parent node to continue the derivation in $mode = select$.

```
EXECUTE(G) =
  if $mode = execute$ then
    if $env(currnode) \in VAR$ then $mode := ramify$ else
      $output := output * env(currnode)$
      $currnode := parent(currnode)$
      $mode := select$
```

For attribute grammars it suffices to extend the instantiation for context free grammars as follows. For the synthesis of the attribute $X.a$ of a node X from its childrens' attributes we add to the else-clause of the BACK macro the corresponding update, e.g. $X.a := f(Y_1.a_1, \ldots, Y_k.a_k)$ where $Y_i = env(o_i)$ for children nodes o_i and $X = env(parent(currnode))$. Inheriting an attribute from the parent and siblings can be included in the update of *env* (e.g. upon node creation), extending it to update also node attributes. The attribute conditions for grammar rules are included into EXECUTE(G) as additional guard to yielding output, of the form $Cond(currnode.a, parent(currnode).b, siblings(currnode).c)$.

In a similar way one can formulate an ASM for tree adjoining grammars, generalizing Parikh's analysis of context free languages by 'pumping' of context free trees from *basis trees* (with terminal yield) and *recursion trees* (with terminal yield except for the root variable), see [65].

Instantiating Turing-like ASMs. We define here a TURINGLIKEMACHINE and specialize it by instantation of its predicates, functions and updates to the following examples of classical automata models: 1-way or 2-way finite state machines, timed automata, pushdown automata, Turing machines, Eilenberg's X-machines, Wegner's interactive Turing machines. For further specializations, including models of distributed computation like Petri nets, we refer the reader to [15].

Turing-like machines are control state ASMs which in each step, placed in a certain *pos*ition of their *mem*ory, check a *Cond*ition concerning the *mem*ory in the *env*ironment of that *pos*ition and react by updating $mem(env(pos))$ and *pos*. Variations of these machines are due to variations of $mem, pos, env, Cond$ and of the *Update*s, whereas their rules are all of the following form:

```
TURINGLIKEMACHINE($mem, pos, env, Cond, Update$) =
FSM($i$, if $Cond(mem(env(pos)))$ then $Update(mem(env(pos)), pos)$, $j$)
```

The specialization of TURINGLIKEMACHINE to 1-way finite automata in the sense of Moore is obtained by specifying *mem* as input function *in* ranging over letters a of an alphabet A. Since the monitored function *in* is 0-ary there is no *pos* or *env* to specify and *Cond* becomes $in = a$. Since Moore automata emit no output, the *Update* of *mem* is empty. So this refinement is a combination of a rather simple data refinement and an equally simple operation refinement.

1-way Mealy automata are obtained by adding to the memory of Moore automata also output, represented by a 0-ary function *out* ranging over letters b of an output alphabet B. Formally the data refinement yields $mem = (in, out)$ and the operation refinement lets *Update* become the assignment $out := b$, which changes the output part of *mem*.

The generalization of 1-way to 2-way Mealy/Moore automata comes up to specify *in* not as a 0-ary but as monadic function (a 'tape') with a dynamic reading head *pos*ition (where *env* always takes value *pos*). As a consequence of this data refinement, *Cond* reads $in(pos) = a$. For the operation refinement an assignment $pos := pos + move$ is added to *Update*.

Obviously instead of letting in, out range over letters a, b one may also have words or other value types and also sets or sequences of input or output lines (ports), like in networks of finite automata [43] or stream FSMs [63]. All such examples, obtainable by data and operation refinements similar to those exhibited above, are instances of *Mealy/Moore ASMs* defined in [14] as TURINGLIKEMACHINEs with arbitrary *mem* (including *in*) and any ASM rule as *Update*, but with the typical FSM *Cond*ition $in = a$. There are numerous examples of such MEALYASMs in the literature, e.g. the components of *co-design FSMs* (see [66]) where the ASM rules compute arbitrary combinational (external and instantaneous) functions. Another famous example is constituted by timed automata [3] where the letter *in*put comes at a real-valued occurrence time which is used in the transitions with clocks recording the time difference of the current input with respect to the previous input:

$$time_{\Delta} = occurrenceTime(in) - occurrenceTime(previousIn).$$

The rule *Cond*ition is specified as $TimedIn(a)$, reflecting that transitions may be subject to clock constraints. Typically the constraints are about input to occur within $(<, \leq)$ or after $(>, \geq)$ a given (constant) time interval, leaving some freedom for timing runs, i.e. choosing sequences of $occurrenceTime(in)$ to satisfy the constraints. The *mem*ory $Update = ClockUpdate$ is about resetting a clock (namely for all elements of a set *Reset*) or adding to it the last input time difference. This instantiation of TuringLikeMachine to timed automata yields the following form of control state ASM rules:

```
$\textsc{TimedAutomaton}(i, a, Constraint, Reset) =$
  $\textsc{Fsm}(i, \textbf{if } TimedIn(a) \textbf{ then } ClockUpdate(Reset), j)$
  **where**
    $TimedIn(a) = (in = a \textbf{ and } Constraint(time_\Delta) = true)$
    $ClockUpdate(Reset) =$
      **forall** $c \in Reset$ **do** $c := 0$
      **forall** $c \notin Reset$ **do** $c := c + time_\Delta$
```

In pushdown automata the Mealy automaton 'reading from input' and 'writing to output' is extended to reading from input and/or a *stack* and writing on the *stack*, formally specifying $mem = (in, stack)$ with a function $pos = top(stack)$. Therefore *Cond* becomes $Reading(a, b)$ from input and/or stack and *Update* becomes a *StackUpdate*, resulting in the following form of ASM control state rules. Since PDAs may have control states with no input-reading or no stack-reading, optional items are enclosed in []. We assume the usual meaning of the *stack* operations *push*, *pop*.

```
$\textsc{PushDownAutomaton} =$
  $\textsc{Fsm}(i, \textbf{if } Reading(a, b) \textbf{ then } StackUpdate(w), j)$
  **where**
    $Reading(a, b) = [in = a] \textbf{ and } [top(stack) = b]$
    $StackUpdate(w) \ = \ stack := push(w, [pop](stack))$
```

For the original *Turing* machines TuringLikeMachine is instantiated by specifying *mem* as a *tape* for both reading (input) and writing (output), with an integer position $pos\colon Z$ of the reading head on the tape where single letters are retrieved. Thus the reading *Cond*ition becomes $tape(pos) = a$ and *Update* like in 2-way FSMs concerns both *mem* and *pos*, namely $tape(pos) := b$ for some b and $pos := pos + move$ for some $move \in \{1, 0, -1\}$.

The extension of the 1-tape Turing machine to a k-tape or to an n-dimensional TM results from data refining further the 1-tape Turing *mem*ory and the related operations and functions. Register machines are a data refined instance of k-tape Turing machines ([9, Ch.AI1]). Eilenberg's *X-machines* [51] (and similarly their stream processing version) instead of read/write operations on words stored in a tape provide data processing for arbitrary data, residing in abstract *mem*ory, by arbitrarily complex global *mem*-transforming functions. Therefore they can be instantiated like Mealy ASMs whose rules in addition to yielding *out*put also update *mem* via global memory functions f (one for each input and control state):

$$\textsc{XMachine} = \textsc{Fsm}(i, \textbf{if } in = a \textbf{ then } \{out := b, mem := f(mem)\}, j)$$

If one prefers to write Turing machine programs in the usual tabular form, with one entry (i, a, j, b, m) for every instruction "in control state i reading input a, go to control state j, print output b and move the reading head by m", one obtains the following guard-free Turing machine rule scheme for updating $(ctl\_state, tape(head), head)$. The parameters $Nxtctl, Write, Move$ are the three projection functions which define the program table, mapping 'configurations' (i, a) of control state and letter under the reading head to the next control state j, the letter b to be written in the reading head position and the move m to be performed by the reading head.

```
$\textsc{TuringMachine}(Nxtctl, Write, Move) =$
  $ctl\_state := Nxtctl(ctl\_state, tape(head))$
  $tape(head) := Write(ctl\_state, tape(head))$
  $head := head + Move(ctl\_state, tape(head))$
```

Wegner's interactive Turing machines [81] in each step can receive some input from the environment and yield output to the environment. Thus they simply extend the TuringMachine as follows by an additional *input* parameter (data refinement) and an *output* action (operation refinement):

```
TuringInteractive($Nxtctl, Write, Move$) =
    $ctl\_state := Nxtctl(ctl\_state, tape(head), input)$
    $tape(head) := Write(ctl\_state, tape(head), input)$
    $head := head + Move(ctl\_state, tape(head), input)$
    $output(ctl\_state, tape(head), input)$
```

Considering the output as written on an in-out tape comes up to define

$output := concatenate(input, Out(control, tape(head), input))$

as the output action using a function *Out* defined by the program. Viewing the input as a combination of preceding inputs/outputs with the new user input comes up to define *input* as a derived function $input = combine(output, user\_input)$ depending on the current *output* and *user_input*. The question of single-stream versus multiple-stream interacting Turing machines (SIM/MIM) is only a question of instantiating input to a stream vector $(inp_1, \dots, inp_n)$.

5. Proving Refinement Correctness

In this section we show how to use ASM refinements for proving system properties. We explain the general scheme and Schellhorn's analysis in [75] for modularizing ASM refinement correctness proofs aimed at mechanizable proof support.

The ASM refinement method provides the designer with a powerful method, which is well known in mathematics for centuries, to (0) show that an implementation S^* satisfies a desired property P^*. Namely instead of directly proving P^* for S^*, one can

1. build an abstract model S,
2. prove a possibly abstract form P of the property in question to hold under appropriate assumptions for S,
3. show S to be correctly refined by S^* and the assumptions to hold in S^*.

The practice of system design shows that the overall task (0), which for real-life systems is usually too complex to be tackled at a single blow, can be accomplished by splitting it into a series of manageable subtasks (1)–(3), each step reflecting a part of the design. Numerous examples illustrating this use of ASM refinements appear in the AsmBook [42], a large real-life case study which makes widely use of that scheme is the book [79]. A rather complete list of applications of that scheme appears in the ASM research survey in [16].

As basis for the machine verification in KIV [76, 77] of the proven to be correct hierarchy of ASMs relating Prolog to its compilation to WAM code [36], Schellhorn has devised in [75] a general scheme for establishing invariants to prove the correctness of an ASM refinement. It is an adaptation of features known from the well established theory of forward simulations. The idea consists in decomposing the commuting diagram in Fig. 1 into more basic diagrams with end points s, s^* which satisfy an invariant $\approx$ implying the to be established equivalence $\equiv$. The method is to follow the two runs, for each pair of corresponding states—not both final—satisfying $\approx$, looking for a successor pair $s', s^{*\prime}$ (of corresponding states satisfying $\approx$). Three cases are possible for such run extensions: only one of the two runs can be extended or both are extendable. These cases give rise to three types of basic diagrams shown in Fig. 2:

- $(m, 0)$-triangles: computation segments where only the abstract run makes progress performing a positive number m of steps to reach an $s' \approx s^*$,
- $(0, n)$-triangles: computation segments where only the concrete run makes progress performing a positive number n of steps to reach an $s^{*\prime} \approx s$,
- (m, n)-trapezoids: representing a computation segment which leads in $m > 0$ steps to an s' and in $n > 0$ steps to an $s^{*\prime}$ such that $s' \approx s^{*\prime}$. Any of the three possible subcases $m < n$, $m > n$ (typical for optimizations) or $m = n$ is allowed here.

To formulate Schellhorn's theorem we first define the **Forward Simulation Condition FSC**. It is defined as the following run condition: for every pair (s, s^*) of states, if $s \approx s^*$ and not both are final states, then

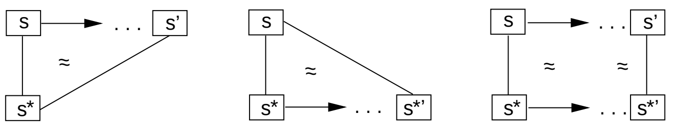

Fig. 2. Components of ASM Refinement Diagrams

- either the abstract run can be extended by an $(m, 0)$-triangle leading in $m > 0$ steps to an $s' \approx s^*$ satisfying $(s', s^*) <_{m0} (s, s^*)$ for a well-founded relation $<_{m0}$ limiting successive applications of $(m, 0)$-triangles,
- or the refined run can be extended by a $(0, n)$-triangle leading in $n > 0$ steps to an $s^{*\prime} \approx s$ satisfying the condition $(s, s^{*\prime}) <_{0n} (s, s^*)$ for a well-founded relation $<_{0n}$ limiting successive applications of $(0, n)$-triangles[1],
- or both runs can be extended by an (m, n)-trapezoid leading in $m > 0$ abstract steps to an s' and in $n > 0$ refined steps to an $s^{*\prime}$ such that $s' \approx s^{*\prime}$.

Schellhorn's *Theorem on Decomposition of ASM Refinement Diagrams* in [75] can now be formulated as follows; it has been proved using the KIV verification system. M^* is a correct refinement of M with respect to an equivalence notion $\equiv$ and a notion of initial/final states if there is a relation $\approx$ (a coupling invariant) such that

1. the coupling invariant implies the equivalence,
2. each refined initial state s^* is coupled by the invariant to an abstract initial state $s \approx s^*$,
3. the forward simulation condition FSC holds.

Also in [49] generic PVS theories are described which define refinement relations between ASMs, without allowing arbitrary non-determinism or triangles. The PVS-based approach to mechanical verification of ASMs has been applied in [84, 50] to prove the correctness of back-end rewrite system specifications with ASM-specified source and target languages.

6. Conclusion: Using ASM Refinements

In this paper we have explained the ASM refinement method. We have defined the general notion of ASM refinement (Sect.3) and have shown (Sect.4) that some of its specializations integrate other refinement approaches in the literature. In Sect. 5 we have analyzed a general correctness proof scheme for ASM refinements, which turned out to be useful for analyzing simulations of abstract runs by concrete runs. Summarizing one can say that the main usage of ASM refinements is directed towards capturing orthogonalities by modular machines. One way to exhibit orthogonalities is the construction of hierarchical levels for *horizontal piecemeal extensions* and adaptations of systems (design for change), as experienced for the first time by the numerous extensions of the ISO Prolog ASM model in [10, 11, 12, 20] by constraints (for Prolog III in [39]), polymorphism (for Protos-L in [5, 7, 6]), narrowing (for Babel in [27]), object-orientation (see [69, 68]), parallelism (Parlog, Concurrent Prolog etc in [72, 30, 31, 71]), and an abstract execution strategy (language Gödel in [32]). Another direction of laying down orthogonalities consists in the use of ASM refinements for the *vertical stepwise detailing* of models (design for reuse) in a proven to be correct way down to their implementation, as experienced for the first time by the model chains leading from Prolog to WAM code [36], from Occam to Transputer code [22] and from the serial to the pipelined RISC architecture DLX [28] which initiated the use of the ASM-refinement method for design-driven architecture verification [61, 80]. Such ASM refinements support an effective *reuse of models and proof techniques*, as experienced for the first time reusing the Prolog-to-WAM compilation models and analysis for compiling CLP(R) to IBM's Constraint Logic Arithmetical Machine CLAM [37] and for the compilation of Protos-L to IBM's Protos Abstract Machine [7, 6].

[1] This well-founded order condition is guaranteed in refinements of event-based B systems by the VARIANT clause, containing an expression for a natural number which has to be shown to decrease by each rule application [2].

An important practical effect of using ASM refinements which scales to industrial-size systems is the enhancement of the communication of designs and of system documentations, based upon the report of the design decisions which led to refinement steps and to the accompanying analysis and justification. Outstanding examples which can be mentioned in this context are standardization efforts which have been supported by ASM modeling and analysis work, namely for the ISO standard of Prolog [20, 35], for the IEEE standard of VHDL'93 [78, 24, 25] and for the ITU standard of SDL-2000 [62, 54, 55, 52, 70]. Last but not least writing manuals and system maintenance are supported by the accurate, precise, richly indexed and easily searchable documentation coming with refinement reports in electronical form. As example we refer to the documentation of the Java programming language and its implementation on the Java Virtual Machine provided by the ASM models for Java/JVM and their analysis in [79] which following a recent evaluation in [59, Section 6.2] "... gives the most comprehensive and consistent formal account of the combination of Java and the JVM, to date".

Acknowledgement. We thank E. Boiten (Cambridge), M-L Potet (Grenoble), G. Schellhorn (Augsburg) and two anonymous referees for valuable critical comments on the first version of this paper.

References

[1] J.-R. Abrial. *The B-Book.* Cambridge University Press, 1996.

[2] J.-R. Abrial and L. Mussat. Introducing dynamic constraints in b. In D. Bert, editor, *B'98: Recent Advances in the Development and Use of the B Method*, volume 1393 of *LNCS*, pages 82–128. Springer, 1998.

[3] R. Alur and D. L. Dill. A theoryof timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[4] M. Barnett, E. Börger, Y. Gurevich, W. Schulte, and M. Veanes. Using Abstract State Machines at Microsoft: A case study. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 367–380. Springer-Verlag, 2000. .

[5] C. Beierle and E. Börger. Correctness proof for the WAM with types. In E. Börger, G. Jäger, H. Kleine Büning, and M. M. Richter, editors, *Computer Science Logic*, volume 626 of *LNCS*, pages 15–34. Springer-Verlag, 1992.

[6] C. Beierle and E. Börger. Refinement of a typed WAM extension by polymorphic order-sorted types. *Formal Aspects of Computing*, 8(5):539–564, 1996.

[7] C. Beierle and E. Börger. Specification and correctness proof of a WAM extension with abstract type constraints. *Formal Aspects of Computing*, 8(4):428–462, 1996.

[8] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.

[9] E. Börger. *Computability, Complexity, Logic (English translation of* Berechenbarkeit, Komplexität, Logik , volume 128 of *Studies in Logic and the Foundations of Mathematics.* North-Holland, 1989.

[10] E. Börger. A logical operational semantics for full Prolog. Part I: Selection core and control. In E. Börger, H. Kleine Büning, M. M. Richter, and W. Schönfeld, editors, *CSL'89. 3rd Workshop on Computer Science Logic*, volume 440 of *LNCS*, pages 36–64. Springer-Verlag, 1990.

[11] E. Börger. A logical operational semantics of full Prolog. Part II: Built-in predicates for database manipulation. In B. Rovan, editor, *Mathematical Foundations of Computer Science*, volume 452 of *LNCS*, pages 1–14. Springer-Verlag, 1990.

[12] E. Börger. A logical operational semantics for full Prolog. Part III: Built-in predicates for files, terms, arithmetic and input-output. In Y. Moschovakis, editor, *Logic From Computer Science*, volume 21 of *Berkeley Mathematical Sciences Research Institute Publications*, pages 17–50. Springer-Verlag, 1992.

[13] E. Börger. Logic programming: The evolving algebra approach. In B. Pehrson and I. Simon, editors, *IFIP 13th World Computer Congress*, volume I: Technology/Foundations, pages 391–395, Elsevier, Amsterdam, the Netherlands, 1994.

[14] E. Börger. High level system design and analysis using abstract state machines. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Current Trends in Applied Formal Methods (FM-Trends 98)*, number 1641 in LNCS, pages 1–43. Springer-Verlag, 1999.

[15] E. Börger. Computation and specification models. A comparative study. In P. D. Mosses, editor, *Proceedings of the Fourth International Workshop on Action Semantics*, volume NS-02-8 of *BRICS Series*, pages 107–130. Department of Computer Science at University of Aarhus, December 2002.

[16] E. Börger. The origins and the development of the ASM method for high level system design and analysis. *J. of Universal Computer Science*, 8(1):2–74, 2002.

[17] E. Börger. Abstract State Machines: A unifying view of models of computation and of system design frameworks. *Annals of Pure and Applied Logic*, 2003.

[18] E. Börger and T. Bolognesi. Remarks on turbo asms for computing functional equations and recursion schemes. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003*, volume 2589 of *LNCS*. Springer, 2003.

[19] E. Börger, A. Cavarra, and E. Riccobene. An ASM semantics for UML activity diagrams. In T. Rus, editor, *Algebraic Methodology and Software Technology, 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 20-27, 2000 Proceedings*, volume 1816 of *LNCS*, pages 293–308. Springer-Verlag, 2000.

[20] E. Börger and K. Dässler. Prolog: DIN papers for discussion. ISO/IEC JTCI SC22 WG17 Prolog Standardization Document 58, National Physical Laboratory, Middlesex, England, 1990.

[21] E. Börger and G. Del Castillo. A formal method for provably correct composition of a real-life processor out of basic components (The APE100 Reverse Engineering Study). In B. Werner, editor, *Proceedings of the First IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'95)*, pages 145–148, November 1995.

[22] E. Börger and I. Durdanović. Correctness of compiling Occam to Transputer code. *Computer Journal*, 39(1):52–92, 1996.

[23] E. Börger, I. Durdanović, and D. Rosenzweig. Occam: Specification and compiler correctness. Part I: Simple mathematical interpreters. In U. Montanari and E. R. Olderog, editors, *Proc. PROCOMET'94 (IFIP Working Conference on Programming Concepts, Methods and Calculi)*, pages 489–508. North-Holland, 1994.

[24] E. Börger, U. Glässer, and W. Müller. The semantics of behavioral VHDL'93 descriptions. In *EURO-DAC'94. European Design Automation Conference with EURO-VHDL'94*, pages 500–505, Los Alamitos, California, 1994. IEEE Computer Society Press.

[25] E. Börger, U. Glässer, and W. Müller. Formal definition of an abstract VHDL'93 simulator by ea-machines. In C. Delgado Kloos and P. T. Breuer, editors, *Formal Semantics for VHDL*, pages 107–139. Kluwer Academic Publishers, 1995.

[26] E. Börger, Y. Gurevich, and D. Rosenzweig. The bakery algorithm: Yet another specification and verification. In E. Börger, editor, *Specification and Validation Methods*, pages 231–243. Oxford University Press, 1995.

[27] E. Börger, F. J. López-Fraguas, and M. Rodríguez-Artalejo. A model for mathematical analysis of functional logic programs and their implementations. In B. Pehrson and I. Simon, editors, *IFIP 13th World Computer Congress*, volume I: Technology/Foundations, pages 410–415, Elsevier, Amsterdam, the Netherlands, 1994.

[28] E. Börger and S. Mazzanti. A practical method for rigorously controllable hardware design. In J. P. Bowen, M. B. Hinchey, and D. Till, editors, *ZUM'97: The Z Formal Specification Notation*, volume 1212 of *LNCS*, pages 151–187. Springer-Verlag, 1997.

[29] E. Börger and L. Mearelli. Integrating ASMs into the software development life cycle. *J. of Universal Computer Science*, 3(5):603–665, 1997.

[30] E. Börger and E. Riccobene. A mathematical model of concurrent Prolog. Research Report CSTR-92-15, Dept. of Computer Science, University of Bristol, Bristol, England, 1992.

[31] E. Börger and E. Riccobene. A formal specification of Parlog. In M. Droste and Y. Gurevich, editors, *Semantics of Programming Languages and Model Theory*, pages 1–42. Gordon and Breach, 1993.

[32] E. Börger and E. Riccobene. Logic + control revisited: An abstract interpreter for Gödel programs. In G. Levi, editor, *Advances in Logic Programming Theory*, pages 231–154. Oxford University Press, 1994.

[33] E. Börger and D. Rosenzweig. From Prolog algebras towards WAM — a mathematical study of implementation. In E. Börger, H. Kleine Büning, M. M. Richter, and W. Schönfeld, editors, *CSL'90, 4th Workshop on Computer Science Logic*, volume 533 of *LNCS*, pages 31–66. Springer-Verlag, 1991.

[34] E. Börger and D. Rosenzweig. WAM algebras — a mathematical study of implementation, Part 2. In A. Voronkov, editor, *Logic Programming*, volume 592 of *Lecture Notes in Artificial Intelligence*, pages 35–54. Springer-Verlag, 1992.

[35] E. Börger and D. Rosenzweig. A mathematical definition of full Prolog. *Science of Computer Programming*, 24:249–286, 1995.

[36] E. Börger and D. Rosenzweig. The WAM — definition and compiler correctness. In C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, Studies in Computer Science and Artificial Intelligence, chapter 2, pages 20–90. North-Holland, 1995.

[37] E. Börger and R. Salamone. CLAM specification for provably correct compilation of CLP($\mathcal{R}$) programs. In E. Börger, editor, *Specification and Validation Methods*, pages 97–130. Oxford University Press, 1995.

[38] E. Börger and J. Schmid. Composition and submachine concepts for sequential ASMs. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic (Proceedings of CSL 2000)*, volume 1862 of *LNCS*, pages 41–60. Springer-Verlag, 2000.

[39] E. Börger and P. Schmitt. A formal operational semantics for languages of type Prolog III. In E. Börger, H. Kleine Büning, M. M. Richter, and W. Schönfeld, editors, *CSL'90, 4th Workshop on Computer Science Logic*, volume 533 of *LNCS*, pages 67–79. Springer-Verlag, 1991.

[40] E. Börger and W. Schulte. Programmer friendly modular definition of the semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, number 1523 in LNCS. Springer-Verlag, 1998.

[41] E. Börger and W. Schulte. A practical method for specification and analysis of exception handling: A Java/JVM case study. *IEEE Transactions on Software Engineering*, 26(10):872–887, October 2000.

[42] E. Börger and R. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis.* Springer-Verlag, 2003.

[43] A. Brüggemann, L. Priese, D. Rödding, and R. Schätz. Modular decomposition of automata. In E. Börger, G. Hasenjäger, and D. Rödding, editors, *Logic and Machines: Decision Problems and Complexity*, number 171 in LNCS, pages 198–236. Springer, 1984.

[44] J. Cohen and A. Slissenko. On verification of refinements of asynchronous timed distributed algorithms. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 34–49. Springer-Verlag, 2000.

[45] O. Dahl, E. W. Dijkstra, and C. Hoare. *Structured Programming.* Academic Press, 1972.

[46] W. P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison.* Cambridge University Press, 1998.

[47] J. Derrick and E. Boiten. *Refinement in Z and Object-Z.* Springer, 2001.

[48] E. Dijkstra. Notes on structured programming. In *Structured Programming*, pages 1–82. Academic Press, 1972.

[49] A. Dold. A formal representation of Abstract State Machines using PVS. Verifix Technical Report Ulm/6.2, Universität Ulm, July 1998.

[50] A. Dold, T. Gaul, V. Vialard, and W. Zimmermann. ASM-based mechanized verification of compiler back-ends. In U. Glässer and P. Schmitt, editors, *Proceedings of the Fifth International Workshop on Abstract State Machines*, pages 50–67. Magdeburg University, 1998.

[51] S. Eilenberg. *Automata, Machines and Languages Vol.A.* Academic Press, 1974.

[52] R. Eschbach, U. Glässer, R. Gotzhein, and A. Prinz. On the formal semantics of SDL-2000: A compilation approach based on an abstract SDL machine. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 242–265. Springer-Verlag, 2000.

[53] J. Fitzgerald and P. G. Larsen. *Modelling Systems. Practical Tool and Techniques in Software Development.* Cambridge University Press, 1998.

[54] U. Glässer. *Analysis and Validation of Formal Requirement Specifications in Model-Based Engineering of Concurrent Systems.* Habilitationsschrift, University of Paderborn, Germany, 1999.

[55] U. Glässer, R. Gotzhein, and A. Prinz. Towards a new formal SDL semantics based on Abstract State Machines. In G. v. Bochmann, R. Dssouli, and Y. Lahav, editors, *SDL'99 — The Next Millenium, Proceedings of the 9th SDL Forum*, pages 171–190. Elsevier Science B.V., 1999.

[56] G. Goos and W. Zimmermann. Verifiying compilers and ASMs. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 177–202. Springer-Verlag, 2000.

[57] Y. Gurevich. A new thesis. *Abstracts, American Mathematical Society*, page 317, August 1985.

[58] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[59] P. Hartel and L. Moreau. Formalizing the safety of Java, the Java Virtual Machine and Java Card. *ACM Computing Surveys*, 33(4):517–558, 2001.

[60] A. Heberle. *Korrekte Transformationsphase — der Kern korrekter Übersetzer.* PhD thesis, Universität Karlsruhe, 2000.

[61] J. Huggins and D. Van Campenhout. Specification and verification of pipelining in the ARM2 RISC microprocessor. *ACM Transactions on Design Automation of Electronic Systems*, 3(4):563–580, October 1998.

[62] ITU-T. SDL formal semantics definition. ITU-T Recommendation Z.100 Annex F, International Telecommunication Union, November 2000.

[63] J. W. Janneck. *Syntax and Semantics of Graphs.* PhD thesis, ETH Zürich, 2000.

[64] J.M.Morris. A theoretical basis for stepwise refinement. *Science of Computer Programming*, 9(3), 1987.

[65] D. E. Johnson and L. S. Moss. Grammar formalisms viewed as Evolving Algebras. *Linguistics and Philosophy*, 17:537–560, 1994.

[66] L. Lavagno, A. Sangiovanni-Vincentelli, and E. M. Sentovitch. Models of computation for system design. In E. Börger, editor, *Architecture Design and Validation Methods*, pages 243–295. Springer, 2000.

[67] C. Morgan. *Programming from Specification.* Prentice-Hall, 1990.

[68] B. Müller. *Eine objektorientierte Prolog-Erweiterung zur Entwicklung wissensbasierter Systeme.* PhD thesis, University of Oldenburg, Germany, 1994.

[69] B. Müller. A semantics for hybrid object-oriented Prolog systems. In B. Pehrson and I. Simon, editors, *IFIP 13th World Computer Congress*, volume I: Technology/Foundations, Elsevier, Amsterdam, the Netherlands, 1994.

[70] A. Prinz. *Formal Semantics for SDL. Definition and Implementation.* Habilitationsschrift, Humboldt University of Berlin, Germany, 2000.

[71] E. Riccobene. A formal computational model for PANDORA. Technical Report CSTR-92-16 and ACRC-92-15, University of Bristol, Department of Computer Science, 1992.

[72] E. Riccobene. *Modelli Matematici per Linguaggi Logici.* PhD thesis, University of Catania, Academic year 1991/92.

[73] R.J.R.Back. On correct refinement of programs. *J. Computer and System Sciences*, 23(1):49–68, 1979.

[74] R.J.R.Back and J. von Wright. *Refinement Calculus: A Systematic Introduction.* Springer, 1998.

[75] G. Schellhorn. Verification of ASM refinements using generalized forward simulation. *J. of Universal Computer Science*, 7(11):952–979, 2001.

[76] G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM case study. *J. of Universal Computer Science*, 3(4):377–413, 1997.

[77] G. Schellhorn and W. Ahrendt. The wam case study: Verifying compiler correctness for prolog with kiv. In W. Bibel and P. Schmitt, editors, *Automated Deduction—A Basis for Applications.* Kluwer, 1998.

[78] I. Standardization. Ieee standard VHDL language reference manual. Technical Report Std 1076-1993, IEEE, 1993.

[79] R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation.* Springer-Verlag, 2001. .

[80] J. Teich, R. Weper, D. Fischer, and S. Trinkert. A joint architecture/compiler design environment for asips. In *Proc. International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES2000)*, pages 26–33. ACM Press, November 2000.

[81] P. Wegner. Why interaction is more powerful than algorithms. *Communications of the ACM*, 40:80–91, 1997.

[82] N. Wirth. Program development by stepwise refinement. *Comm. ACM*, 1971.

[83] J. C. P. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof.* Prentice-Hall, 1996.

[84] W. Zimmerman and T. Gaul. On the construction of correct compiler back-ends: An ASM approach. *J. of Universal Computer Science*, 3(5):504–567, 1997.