

# Towards ASM Engineering and Modelling

Bernhard Thalheim and Peggy Schmidt

Christian-Albrechts-University Kiel, Department of Computer Science, 24098 Kiel, Germany  
thalheim | pesc@is.informatik.uni-kiel.de

## Abstract

*The ASM approach has gained a maturity that permits the use of ASM as the foundation for all computation processes. All known models of computation can be expressed through specific abstract state machines. These models can be given in a representation independent way. Stepwise refinement supports separation of concerns during software development and will support component-based construction of systems, thus providing a foundation of new computational paradigms such as industrial programming, programming-in-the-large, and programming-in-the-world.*

*Despite the theoretical and application maturity a modelling theory for ASM specifications does not exist. Pragmatism and methodologies are necessary whenever larger systems have to be specified. We develop a number of principles and approaches to ASM specification that allow one to develop modular and surveyable ASMs. Our approach is based on the Turbo ASMs, abstraction layers, and on refinement. ASM engineering is based on a well-defined methodology that promises to be manageable.*

## 1 Modelling of Applications Based on ASM

### 1.1 Properties of Modelling and Kinds of Abstraction

Modelling is one of the most difficult tasks in software engineering. It aims at a representation or simulation of reality and at identification of a particular model. The given application is subject to analysis by modelling if it can be described in terms of expressions in the language used for modelling. The model is a result of modelling. It relates things  $\mathcal{D}$  under consideration with concepts  $\mathcal{C}$ . This relationship  $\mathcal{R}$  is characterised by restrictions  $\rho$  to its applicability, by a modality  $\theta$  or rigidity of the relationship, and by the confidence  $\Psi$  in the relationship. The model is agreed upon within a group  $\mathcal{G}$  and valid in a certain world  $\mathcal{W}$ . Stachowiak [Sta92] defines three characteristic properties of models: the *mapping* property (have an original), *truncation* property (the model lacks some of the ascriptions made to the original), and *pragmatic* property (the model use is only justified for particular model users, tools of investigation, and period of time). In [KT06] is additionally considered the *extension* property. The property allows models to represent judgments which are not observed for the originals. In computing, for example, it is often important to use executable models. Finally, the *distortion* property is often used for improving the physical world or for inclusion of visions of better reality.

Software engineering uses a number of principles that refine different kinds of abstraction [Tha00] such as construction abstraction, context abstraction and refinement

abstraction. Construction abstraction uses the principles of hierarchical structuring, constructor composition, and generalisation. Refinement abstraction uses the principle of modularisation. Hierarchical structuring uses the decomposition of software into subparts in such a way that constituents form a tree. Modularisation encapsulates components and uses interfaces for exclusive communication of components with the environment. Constructor composition depends on the constructors used [Tha05]. Additionally, principles such as well-founded structuring may be applied. The last principle requires that only such constructors are applicable (sequence, bounded iteration, choice, etc.) for which the compositionality principle is preserved and semantics can be derived based on the inductive construction.

## 1.2 Challenges of Modern Software Engineering

Software engineering is still based on *programming in the small* although a number of approaches has been proposed for *programming in the large*. Programming in the large uses strategies for programming, is based on architectures, and constructs software from components which collaborate, are embedded into each other, or are integrated for formation of new systems. Programming constructs are then patterns or high-level programming units and languages. The next generation of programming observed nowadays is *programming in the world* within a collaboration of programmers and systems. It uses advanced scripting languages such as Groovy with dynamic integration of components into other components, standardisation of components with guarantees of service qualities, collaboration of components with communication, coordination and cooperation features, distribution of workload, and virtual communities. The next generation of software engineering envisioned is currently called *programming by composition* or construction. In this case components form the kernel technology for software and hardware.

Software development is mainly based on stepwise development from scratch. Software reuse has been considered but has never reached the maturity for application engineering. Software development is also mainly *development in the small*. Specifications are developed step by step, extended type by type, and normalized locally type by type. Software engineering is still be considered as *handicraft* work which requires the skills of an *artisan*. Instead, we need techniques for this century [Boe06]. Classical software development methods are mainly appropriate for programming in the small and combination of such programs into a program system. The ASM approach has the expressivity to handle also programming in the large together with programming in the small. Engineering in other disciplines has already gained the maturity for industrial development and application we need to reach.

Software engineering can be based on the trilogy consisting of the application domain description, the requirements prescriptions, and finally the systems specifications [Bjo06,Hei96]. This approach extends modern software engineering approaches by explicit consideration of the application domain.

Advanced applications such as web information systems require novel specification and development methods since their specification is oriented towards systems that are easy and intuitively to use. [Tha03,ST05] extend these approaches by (1) explicit con-

sideration of user expectations, profiles and portfolio and (2) by storyboards and story spaces.

### 1.3 Achievements of the ASM Approach

The ASM method nicely supports high-level design, analysis, validation and verification of computing systems:

- ASM-based specification improves industrial practice by proper orchestration of all phases of software development, by supporting a high-level modelling at any level of abstraction, and by providing a scientific and formal foundation for systems engineering. All other specification frameworks known so far only provide a loose coupling of notions, techniques, and notations used at various levels of abstraction. By using the ASM method, a system engineer can derive a general application-oriented understanding, can base the specification on a uniform algorithmic view, and can refine the model until the implementation level is achieved. The three ingredients to achieve this generality are the notion of the ASM itself, the ground model techniques, and the proper treatment of refinement.
- Abstract state machines entirely capture the four principles [ZT04] of computer science: structuring, evolution, collaboration, and abstraction.

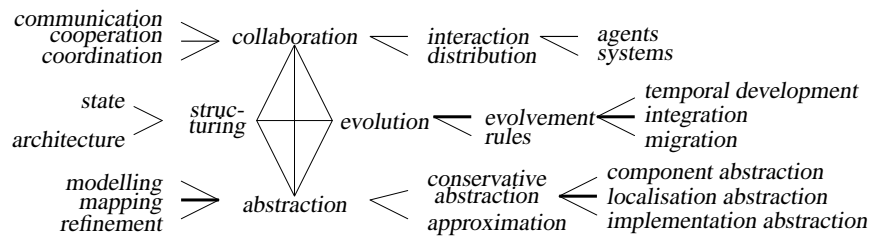


Fig. 1. The Four Principles of Computer Science

This coverage of all principles has not been achieved in any other approach of any other discipline of computer science. Due to this coverage, the ASM method underpins computer science as a whole. We observe the following by comparing current techniques and technologies and ASM methods: ASM are running in parallel. Collaboration is currently mainly discussed at the logical or physical level. Evolution of systems is currently considered to be a hot but difficult topic. Architecture of systems has not yet been systematically developed.

- The ASM method is clearly based on a number of postulates restricting evolution of systems. For instance, sequential computation is based on the postulate of sequential time, the postulate of abstract state, and the postulate of bounded exploration of the state space. These postulates may be extended to postulates for parallel and concurrent computation, e.g., by extending the last postulate to the postulate of finite exploration.

## 1.4 Plan of the Paper

We are completely aware of the complexity of the modelling problem and do not expect that it can be solved within a single conference paper. Therefore, we restrict our efforts to modelling instruments of the ASM method, and to separation of concerns into development layers. In Section 2, choices for modelling are discussed: modularisation, agent orientation, styles and pattern. The section concludes with general properties. These choices are illustrated in Section 3 for a specific modelling method: layered ASM modelling. Due to space limitations we do not use sophisticated examples. We also do not discuss architectures of ASM machines. Section 4 concludes the paper with a discussion on future work.

## 2 ASM Modelling Alternatives

In [KT06] a general approach is proposed to modelling that starts with a clarification of the properties of modelling and of the kinds of abstraction that are considered and with an elaborated and reasoned selection of the modelling language that includes detailed knowledge of deficiencies of this language and therefore avoids the Sapir-Whorf hypothesis [Who80]. We extend this framework by an application-driven choice of architecture and platform, by a collection of modelling styles, and by orchestration of modelling techniques such as pattern. In this case, we shall be able to derive properties of the modelling process.

### 2.1 Modularisation

Modular modelling supports information abstraction and hiding by encouraging and facilitating the decomposition of systems [BM97] into components and their modular development based on a precise definition of interfaces and the collaboration of components through which the systems are put together. Implicit modularisation can be achieved by introduction of name spaces on signatures. Explicit modularisation offers a better understanding of structure and architecture of systems and thus supports consideration of evolution of systems and of collaboration of systems.

Modularisation offers a number of advantages: separation of concerns, discovery of basic concepts, validation and verification of development, efficiency of tool support, and - last but not least - scoped changes. The last advantage of modularisation is based on an explicit framing of development to a number of elements while preserving all other elements in its current form. We model this impact by introducing name spaces on signatures.

Typically, small submachines capture smaller models that are easier to understand and to refine. Small models can better be ascertained as to whether we need to apply refinements.

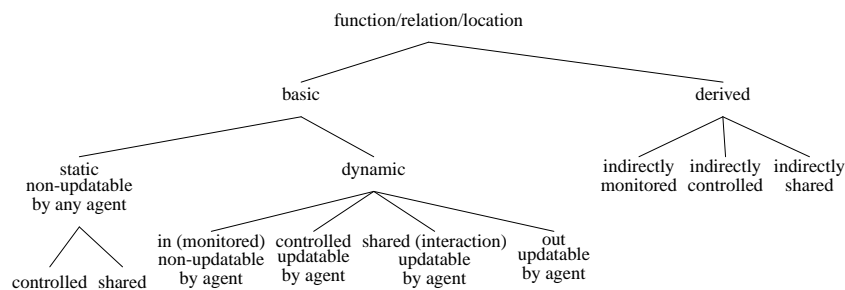
Modularization is a specification technique of structuring large specifications into modules. It is classically based on structural and functional decomposition [BS00]. We additionally consider control decomposition. Modules form a lattice of associated submachines having their own states and their own control.

Modularisation is based on implementation abstraction and on localization abstraction. *Implementation abstraction* selectively hides information about structures, semantics and the behavior of ASM concepts. Implementation abstraction is a generalization of *encapsulation* and *scoping*. It provides *data independence* through the implementation, allowing the private portion of a concept to be changed without affecting other concepts using that concept. *Localization abstraction* “factors out” repeating or shared patterns of concepts and functionality from individual concepts into a shared application environment. Naming is the basic mechanism for achieving localization. Parametrisation can be used for abstraction over partial object descriptions. We use the *name space* for handling localisation abstraction.

## 2.2 Agent-Oriented Specification

An ASM submachine consists of a vocabulary and a set of rules. In this case, any clustering of rules and of elements from the vocabulary may define a submachine. Turbo ASM [BS03] capture our notion of a submachine by encapsulating elements of the vocabulary and rules into an ASM. They hide the internals of subcomputations within a separate ASM. The submachine has its own local state and its own interface.

The set of functions of each submachine can be separated into basic and derived functions. Basic functions may be static functions or dynamic functions. Classically [BS03] dynamic functions can be classified as in(put) functions, out(put) functions, controlled or local functions that are hidden from the environment, and shared functions that are visible to the environment. A similar classification can also be applied to basic static functions. They are either functions only used by a its own machine or read by several environments. We thus extend the notion of shared and controlled functions to static functions as well. We do not use derived static functions since they can be considered as syntactic sugar. We differentiate these functions according to their role in Figure 2 which displays the functions *internal* for an agent ASM. A similar classification can be developed for functions *external* to an agent. An agent ASM consists of all functions that assigned to the agent and of rules that are assigned to the agent and that use only those functions assigned to the agent.



**Fig. 2.** The Kinds of Internal Functions for Agent ASMs

Static functions may also be local functions. They are not updated by any submachine. [BM97] distinguish derived function to whether these functions are monitored functions, controlled functions, or shared functions. Typically, derived functions are functions that do not exist on their own right, but may be dynamically computed from one or more base functions. They provide a powerful and flexible information hiding mechanism. Updates made in the base functions that affect the derived function are immediately reflected in derived functions.

We may additionally assume that derived functions are allowed to update dynamic functions. In this case, dynamic functions may be used as a security mechanism, as an access mechanism, and as a simplification mechanism that allows to use complex derived functions in rules instead of complex computations in rules.

### 2.3 Perspectives and Styles of ASM Modelling

Different modelling perspectives can be distinguished:

1. The *structure-oriented perspective* focuses on structural description of the ASM. Sometimes, the structure-oriented perspective is unified with the semantic perspective. In this case, design of the structure is combined with design of invariants.
2. The *behavior-oriented perspective* is concerned with the behavior of the ASM during its lifetime. It can be based on event approaches or on Petri-net approaches and predicate transition systems.
3. The *process-oriented perspective* is concerned with the operation of the system.

The structure-oriented perspective is often used for data-intensive applications. Almost all recognized database design approaches are based on the structure-oriented perspective. The process-oriented perspective uses approaches considered in software engineering. The behavior-oriented perspective is a high-level descriptive approach to an integrated specification of the vocabulary and rules.

Modelling styles provide a very abstract description of a particular set of general characteristics of a model. Different constructional notations may be useful for describing a machine. We use the Turbo ASM approach for component or submachine description. Typically, the role of the components of the system follow the rules specified by the style. The modelling style explains the structure, the abstraction and grouping of the elements. Parts of the ASM may follow different modelling styles.

The style of modelling is a specification of the high level structure and organisation of ASM modelling. The structure describes the handling of elements of the vocabulary, the topology or relationships between elements, the semantical limitations for their usage, and the interaction mechanism between the elements such as blackboard, submodule calls, etc. The organisational style describes relevant local and global structures, the decomposition strategy, and control mechanisms between parts of the ASM machine. The organisational style is based on the architectural style. It is our aim to maintain and to preserve the strategy over the life cycle of the system.

The perspective and the style result in *strategies* that are used for step-wise development of specifications. The different strategies [Tha00] based on the structure-oriented perspective are sketched in Figure 3.

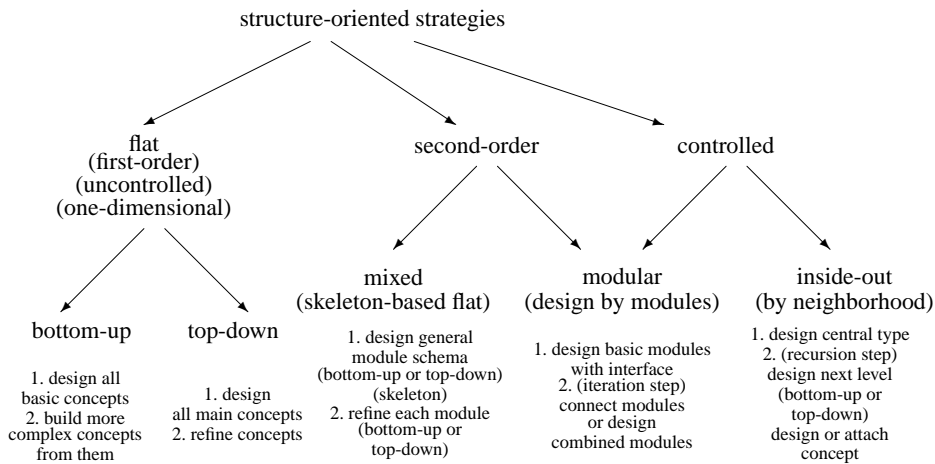


Fig. 3. Structure-Oriented Specification Strategies

## 2.4 Pattern of ASM Vocabulary and Rule Descriptions

The notion of pattern originates from traditional architecture and denotes a general repeatable solution to a commonly occurring problem in software design. Typically, patterns are instantiatable expressions and can be transformed directly into specifications.

The vocabulary description may follow a number of different patterns. Structure-oriented strategies may apply a number of different patterns, for instance the following:

**Compacting patterns** integrate functions and represent them through one function. They provide a compact representation. For instance, the file specification in [Stä04] uses

```
argFile : isActive → isFile
```

which assigns files to agents.

The compacted function

```
argName : isActive → string
```

assigns a new file name to the file to be created or assigns a new name to the file to be moved. In both cases, an agent has only one task and thus is assigned to one file by `argFile` for each file operation. The function compacts the functions

```
argNameCreate : (isActive → isFile) → string .
```

```
argNameMove : isFile → string .
```

**Typing patterns** divide the vocabulary into types and define functions within a type or within associations among types. The vocabulary can be divided into types. For instance, the file specification is separated from the activities of agents.

**Unfolding patterns** provide all functions that are associated with a domain. In our example, all functions that are definable for agents are specified.

**Union patterns** use the most general range for functions. For instance, an agent has a main parent directory that is essentially a file. Therefore, [Stä04] specifies

```
argParent : isActive → isFile
```

and tests in rules whether the result of `argParent` is a directory.

Each of these patterns has its advantages. Compacting and unfolding patterns allow convenient rule description. Refinement is supported by typing patterns. Unfolding patterns

lead to high redundancy that must be effectively supported. Union patterns avoid the covariance / contra variance problem but lead to problematic rules.

These patterns result in description styles. Typical description styles for structure-oriented perspective are

- predicative representation that uses a Boolean functions for the vocabulary or predicates and
- functional representation that uses functions with ranges of arbitrary domain types is appropriate for specification of event systems.

The structure-oriented perspective is typically based on a *predicative pattern* of specification.

The description of the state space can be either given based on *open world* pattern or a *closed world* pattern. The closed world pattern allows only those values that are explicitly given. The values are typically given through ground-term algebras, e.g., enumeration types. The open world pattern allows to extend the state space whenever this is necessary.

The description of terms, variable assignment, formulas, and interpretation is typically based on the *canonical specification pattern* of mathematical logics.

The rule description also may follow patterns. Typical ASM patterns are the following:

**Event-condition-action patterns** are based on a separation of the state space into event states and other states.

**Control state patterns** are based on explicit usage of control states. These states are used to separate activity of rules into those that are applicable and those that are not applicable.

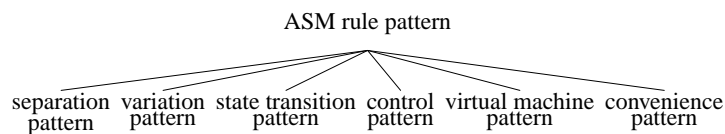
**Error patterns** can be combined with any rule. They may be folded into the activities of the rule or into the condition of the rule.

**State transition patterns** are used for transition to a new state from the current state.

They are typically folded into activities of rules.

**Macro patterns** are parameterised rules that allow to reuse fragments of ASM [SSB01].

We can use almost all software engineering patterns within ASM specifications. Therefore, the pattern list might be rather large. Figure 4 surveys different kinds of patterns for rules.



**Fig. 4.** Kind of ASM Rule Pattern

Pattern instantiation requires *fitting* of values to the parameters. This fitting is specified through context conditions. For instance, values used in conditions must fit into domains that are valid for the states potentially given for the vocabulary.

## 2.5 Pattern for Invariants

Invariants, e.g. integrity constraints in database applications, are used to define semantics of applications. We know different pattern for their specification:

- *Operational representation* of invariants incorporates invariants into the programs or rules. The invariant enforcement mechanism may be hidden because of control conditions or to the specification of actions.
- *Descriptive representation* uses explicit specification and refinement obligations. These descriptions are combined with the specification of invariant enforcement:
  - Eager enforcement maintains invariants based on a scheduling mechanism for maintenance of invariants. Transactional systems are typical scheduling mechanisms. They bind invariant enforcement to programs.
  - Lazy enforcement maintains invariants in a delayed mode. Inconsistency is temporarily tolerated. This tolerance reduces some of the cost of enforcing invariants within large structures.
  - Refusal enforcement maintains invariants by rollback of all activities since the last consistent state and by executing a subset of activities. Partially ordered runs are based on refusal enforcement.

Depending on the pattern chosen invariant handling is varies. If we choose an implicit invariant handling then any change applied to the current ASM must explicitly consider all invariants and must be entirely aware of the effects of these. Therefore this pattern is the most inefficient for early design phases. This pattern is however applicable during implementation if later revision is going to be based on a more general ASM.

The completeness of invariant specification is a dream that is never satisfied. Sets of invariants are inherently open since we cannot know all invariants valid in the current application, we cannot envision all possible changes in invariant sets, and we cannot choose the most appropriate selection of invariants from which all other invariants follow. Therefore, we use a separation into

- hard (or iron) invariants that must be preserved and which are valid over a long time in the application and
- soft invariants that can be preserved or are causing later corrections or which are not valid for a longer time in the application.

## 2.6 ASM Modelling Assumptions

The *unique name assumption* requires that elements with different names are different. If we need to use the same name for different purposes then we use *name spaces* if a unique identification is needed. The *closed world assumption* presumes that the only possible elements are those which are specified. The *domain closure assumption* limits the elements in the language to those that can be named through the vocabulary, the states of the vocabulary or the rules.

Two additional assumptions we may apply are the *unique meaning assumption* and the *universal machine assumption*. The first assumption postulates that any function or rule of the ASM has the same meaning despite modularisation. The second assumption postulates that the behaviour of the entire ASM can be defined by composition applied to the submachines.

Due to the variety of choices we might use additional assumptions for the development. The most general *architectural assumption* is the possibility of *layering* a system into sub-systems. We might use other assumptions such as common data pools, transactional systems providing an exclusive write to a location for one sub-system and a guided read with un-read to this location for all other subsystems. The use of shared functions determines whether a system consists of strictly separated components that do not have shared functions or consists of a system of components with overlapping, i.e., shared functions.

## 2.7 Properties of ASM Modelling

The software development or generally the modelling process is intentionally or explicitly ruled by a number of development strategies, development steps, and development policies. Modelling steps lead to new specifications to which quality criteria can be applied. Typical quality criteria are completeness and correctness in both the syntactical and semantical dimensions. We assume that at least these four quality criteria are taken into consideration. The modelling process can be characterised by a number of (ideal) properties:

**Monotonicity:** The modelling process is monoton in, if any change to be applied to one specification leads to a refinement. It thus reflects requirements in a better form.

**Incrementality:** A modelling process is iterative or incremental if any step applied to a specification is only based on new requirements or obligations and on the current specification.

**Finiteness:** The modelling process is finite if any quality criteria can be checked in finite time applying a finite number of checks.

**Application domain consistency:** Any specification developed corresponds to the requirements and the obligations of the application domain. The appropriateness can be validated in the application domain.

**Conservativeness:** A modelling process is conservative if any model revision that cannot be reflected already in the current specification is entirely based on changes in the requirements.

Typical matured modelling processes are at least conservative and application domain consistent. Any finite modelling process can be transformed into a process that is application domain consistent. The inversion is not valid but depends on quality criteria we apply additionally. If the modelling process is application domain consistent then it can be transformed in an incremental one if we can extract such area of change in which consistency must be enforced.

## 3 Layered ASM Modelling

We elaborate one kind of ASM modelling in more detail. Layered ASM modelling is based on modularisation and on architectures of the ASM. The language layering approach we use has already been reported in a similar form in [Wal97].

### 3.1 Assumptions of Layered ASM Modelling

Layered ASM modelling is based on the architectural assumption that the system can be separated into components and a general layering is achievable. We base layered ASM modelling on the unique name assumption, the domain closure assumption, and the universal machine assumption. We may use the closed world assumption and the unique meaning assumption. Layered modelling is not restricted to the last two assumptions.

In general, we use architecture-driven development that starts first with the prescription of the architecture pattern and style. The agent-oriented specification of ASM allows the development of a system as a collaborating society of sub-systems. This society uses shared functions where the sharing is based on contracts for the usage of these functions, on workflows that describe the cooperation among these sub-systems, and on implicit communication based on the locations for these functions [ST07]. We may use different views of the same architecture [Sie04] such as technical views displaying the modules with their functionality, application views displaying activity zones depending on the stage of the application, infrastructure views displaying the dependence of the system from its infrastructure and supporting systems, or the context view that considers the whole organisational, story and application context.

### 3.2 Vocabulary Modelling

The five properties of modelling are: mapping, truncation, pragmatic, extension and distortion properties. These properties govern abstraction. We consider six different aspects for vocabulary modelling: *intention*, *usage*, *content*, *functionality*, *context*, and *realisation*. The intention aspect is a very general one centered around a mission statement for the system. The primary question is: what is the purpose of the system? Once some clarity with respect to the intentions of the system has been obtained, it is important to anticipate the behaviour of the users. The content aspect concerns the question: Which information should be provided? The functionality aspect is coupled with the question, whether the system should be passive or active. The context aspect deals with the context of the system with respect to society, to time, to expected users, to the history of utilisation and to the paths of these users through the system. The realisation aspect concerns the final implementation.

Vocabulary modelling must cover all these aspects in a proper form. It may be based on partial order-sorted signatures. Depending on the choices we made for modularisation, separation of concerns through agent-oriented specification, perspectives and styles, pattern, and properties of modelling itself we can use different representations and conventions. Typical *conventions* are naming and binding conventions.

Layered ASM modelling can be based on typing pattern. We are additionally interested in incrementality and application domain consistency. We additionally assume *value-identifiability* for each element of the state. This assumption allows to define equality and inequality for each pair of elements of the state. Therefore we develop a multi-layered vocabulary modelling approach that uses the following ingredients:

**Domain types** are used for introducing the set of states envisioned. The superuniverse is the union of all states. Beside basic value types such as *string*, *int*, *float* etc. we assume domain types

BOOL, NULL,  $\emptyset$ , ID

consisting of the truth values *true*, *false*, of the value *undef*, of the empty set, and of a set of identifiers, correspondingly.

Domain types can be complex types that are inductively constructed from basic types by applying constructors such as (Cartesian) product, set, list, and multi-set constructors.

Domain types can be labelled by an abstract name. Domain types are typically order-sorted. We restrict the partial order to lattices or Brouwerian algebras.

**Abstract types** are denoted by a triple consisting of an abstract name denoting the abstract type, of a domain type used for values, and a set of invariants limiting the state space that can be used for interpreting the abstract type. The set of invariants may be empty. In this case we omit the third part of the triple. If the abstract name coincides with the domain type name then it can be omitted as well.

**Predicates** are specified by their name, an arity, a sequence of abstract types, and a set of invariants, where the length of the sequence coincides with the arity. Invariants limit the state space that can be used for interpreting the predicate. It can be empty. Predicates are interpreted on the basis of set semantics.

**Functions** are specified by their name, an arity, a sequence of abstract types used for the domain of the function, an abstract type used for the range of the function, and a set of invariants where the length of the sequence coincides with the arity. Invariants limit the state space that can be used for interpreting the function. It can be empty. Functions are interpreted on the basis of set semantics for their domain.

Predicates and functions are often partial. Due to application domain consistency we assume that each abstract type has a natural meaning in the application domain. We also may assume that labels have a natural meaning in the application domain or are used as an abstraction for convenience of specification. We use the unique name assumption for all labels, types, predicates and functions. It is convenient to assume that abstract names and the superuniverse are disjoint.

Additionally we support the introduction of derived notions:

**View functions** are derived functions. They can be *virtual* or *materialized*. Virtual view functions are computed whenever this is necessary. They are not stored in the ASM. View functions can be given in

- an *explicit* form by their introduction in the vocabulary and
- an *implicit* form by their introduction in rules using the **let**, **choose**, **for all** and **where** introductions. We may distinguish
  - *transient view functions* that are used for definition of values in **let** and **choose** rules and may be directly removed whenever a value has been chosen and
  - *collector view functions* that are used for parallel execution in **forall** rules and in **where** introduction and must have a lifespan that last over the entire rule computation.

Virtual view functions are the view functions typically used. In this case, it is assumed that the view is computed first before applying a rule, e.g. in a **where** introduction.

**Clusters functions** are disjoint unions of functions. They are well known in programming languages and are mainly used as syntactic sugar since generalisation of functions by combination eases the treatment.

**Control state functions** are often introduced in rules through conditions that use a certain control state as the enabler or disabler of the rule.

**Special purpose functions** are used for *default functions* and *exception functions*.

Each of the elements of the vocabulary has its *scope* that consists of the space of all locations that are used for the functions and of all rules that use these elements.

All these derived notions may be used inductively for the construction of other derived functions. Therefore, the vocabulary becomes layered depending on the construction.

Additionally, we need to consider *metadata* describing the specific purpose of elements of the vocabulary. They represent the content and the meaning of the functions. The meaning can be partially described by the name if the wording used has a mini-semantics. Metadata also include technical data that guide refinements applied to the vocabulary. Metadata may be partially given through a glossary or thesaurus.

Finally, a well-developed specification uses *naming conventions* for readability.

We distinguish between

**sketchy specification** of the vocabulary that uses an implicit definition of domain types and of abstract types and that specifies predicates and functions through declaration of the mappings and

**sophisticated specifications** of the vocabulary with detailed description of any element of the vocabulary.

The typical specification of the vocabulary uses a specification approach between these extremes. Whenever refinements need to be made then sophistication is necessary for the verification of refinement correctness.

### 3.3 State Space Modelling

State and vocabulary are two sides of the same coin and must be developed in a co-design process. We distinguish two extremes:

- Orientation to most general domain types: We use the most general domains for the specification. If we need specific domains then we can use unary predicates. The exclusive utilisation of most general domain types leads to complex invariants for values with specific meaning and for relations among the values and to extensive specification of various ‘exceptions’. The refinement is simpler but the burden for correct functioning is transferred to rule specification.
- Tight coupling of domain types and abstract types: Any abstract type is associated with the most specialised domain type. Specialised domain types are often associated to a specific meaning in the application. Refinement becomes more difficult. At the same time, rule specification can concentrate on the essentials. Exception due to weak domain types do not appear.

Enumeration states are often used for control states. These control states separate transitions and lead to implicit clustering of rules. Control states have a implicit *scope* that

is defined by their utilisation in conditions and by transfer assignments in rules from or to another control state. Rules that are enabled by a control state form a submachine or a module. The transfer from one control state to another control state is represented by a state transition graph.

### 3.4 Agent-Based Modularisation

We combine modularisation and agent-oriented specification. Each function and each predicate may be visible or may not be visible to an agent. Any agent has its *vocabulary share*. This vocabulary share states whether the given predicate or function is an in element (a monitored function or predicate), a controlled element, a shared element or an out element. A function or predicate may also be external (denoted by  $E$ ) for an agent.

Therefore, given a set  $\mathcal{A}$  of agents and a vocabulary  $\mathcal{V}$ , the vocabulary share is given by a function

$$\text{share} : \mathcal{A} \times \mathcal{V} \rightarrow \{E, I, O, C, S\}$$

assigning agents their kind of vocabulary use.

If an agent uses a dynamic function or predicate as its controlled function then no other agent can use it. If an agent uses a dynamic function or predicate as its in or out element then we should require that another agent uses this function or predicate as an out or an in element, correspondingly. We may additionally require that dynamic functions and predicates of the vocabulary are partitioned into in, out, controlled, and shared functions. Typically, we require that the shared dynamic function predicate are consistently assigned, i.e., the function or predicate is either external or shared for any agent that uses it internally. The same restriction can be made for static functions. This restriction is a variant of the well-known open-closed principle [Mey88] for vocabulary in layered ASM modelling.

According to Figure 2 we restrict static functions and predicates to controlled and shared share. We also exclude using derived functions and predicates as indirectly out functions.

Derived notions may be restricted to be used only as means for the simplification of vocabulary definition or as services provided by agents. In the later case they will have the same behaviour as out functions and predicates. They may be used by other agents.

It is sometimes convenient to use the elements of the vocabulary in a mixed form. For instance, a dynamic function is an in function for one agent and a shared function for other agents. We avoid this approach for layered ASM modelling since it can be a source for confusion.

Layered ASM modelling also aims to establish a clear definition of the kind of sharing. We introduce an abstract type `right` that denotes the rights an agent may have for updating the shared elements and an abstract type `obligation` that denotes the obligations an agent must follow if this element is a shared element for the agent. We introduce the contract for an agent by assigning rights and obligations to agent for each shared function or predicate by a partial function

$$\text{contract} : \mathcal{A} \times \mathcal{V} \rightarrow \text{right} \times \text{obligation}$$

that assigns the right and the obligation to each agent for each function or predicate the agent shares. It is often convenient to use the contract function as a dynamic function that can be changed by an agent that acts in the role of a controller or scheduler. In this case we can assign to an agent an exclusive update right while excluding other agents from writing. These agents have in this case only read rights. This approach eases introduction of transactional systems.

Additionally, we may use a general agent `Main` for the main ASM.

### 3.5 Modular Rule Modelling

Rules are mainly specified based on different condition-action patterns, e.g., event-condition-action pattern, control state pattern, state transition pattern. The basic specification of a rule is extended by

**state dependence** that describes whether a rule can be invoked in dependence of certain conditions on a state,

**access environment** that determines where the invocation of a rule may appear, and

**control guards** that restrict invocation of a rule and are handled by a controller ASM.

The extension has been introduced for convenience. It can be expressed by an ASM that is generically added to the given ASM. We also use this extension for explicit treatment of conflicts in update sets of rules. This extension does not limit the application of partial ordered runs but allow an explicit treatment of such updateset conflicts that must be treated.

State dependence condenses conditions on the existence of certain objects in the state, value conditions for the invocation of a rule, and collaboration conditions that relate the given rule to the invocation of other rules. Collaboration conditions can also be used for explicit synchronisation of parallel rule invocation. State dependence specification supports subject-oriented programming that focuses on capturing different subjective perspectives on a single object model. It basically allows composing applications out of “subjects” (partial object models) by means of declarative composition rules.

The access environment specification contains the views on a state that are initialised when a rule is going to be executed and the internal functions that are used for execution of the rule. Each of the rules has its *scope* that consists of the space of all functions and predicates that are used in the rule. The scope contains all parameters of the rule. The scope can be used to bind an element of the vocabulary to all rules have this element within its scope. This inversion is called *vocabulary element scope*. We assume the validity of the domain closure assumption for the vocabulary used in rules. Any name used in a rule must be either a value or a type, function, predicate or view given by the vocabulary. We may allow implicit views. We also assume the **open-closed principle** for rules in layered ASM modelling: *If an agent is assigned to a rule then the scope must only contain such elements of the vocabulary on which the agent has an internal share.*

Control guards allow one to avoid inconsistent update sets. We use a partition of control states for separation of runs of abstract state machines. Control guards may be used as entry guards that restrict invocation of a rule and accept guards that restrict updates of a rule. They allow one to express rely-conditions and guarantee-conditions by

both pre- and post-conditions. Rely conditions state what can be tolerated by the party. Guarantee-conditions record the interference that other processes will have to denote with if they are allowed to run in parallel. We envision in this paper that these conditions can be generalized to a specific style of *assumption-commitment specification*. [Wal97] and [SSB01] use exception types which are a very specific type of control guards. Control guards may also be used for introduction of break and continue statements for a rule with temporary lock and wait views.

Rules may be combined to form *macros*, which may be reused by other submachines. Macros support intentional programming and aspect-oriented separation of functionality. They can be generalised for adaptive programming, generative programming, and pattern-based development. Intentional programming provides an extendible programming environment based on transformation technology and direct manipulation of active program representations. New programming notations and transformations can be distributed and used as plug-ins in a play-in/play-out engine [HM03]. Aspect-oriented programming improves the modularity of designs and implementations by allowing a better encapsulation of cross-cutting concerns such as distributed transfer, synchronization, data traversal, tracing, caching, etc. in a new kind of modularity called “aspects”. Generative programming aims to increase the productivity, quality, and time-to-market in software development thanks to the deployment of both standard component and production automation. System families are developed rather than single systems. Generative programming uses government and binding [BST06].

## 4 Concluding: Future Plans for Evolution of ASM Modelling

### 4.1 Treatment of Over-Specification

[HT00] discussed the downside of over-specification. A good specification contains a protocol for future extension and a portfolio for the current implementation. It is a contract with the application stakeholder and an unambiguous description of the application domain. Therefore, it seems that the specification must be as complete as only possible. This ‘completeness’ leads to the *over-specification*. Formal methods should not be misused for a hyper-detailed description of an application but should provide robustness against other interpretations and understandings, against changes in the application itself or within the computing environment, against evolution of the application domain, and against multiple styles of the specification. A hyper-detailed specification suffers from the *straitjacket* effect that limits the flexibility of specification.

The ASM methods offers executability of the specification and thus treats the strait-jacket effects and supports robustness. The over-specification problem can however be solved by explicit introduction of *checkpoints* that allow to overcome the dangers of over-specification. A check point measures the specification itself. These measures might be build in a similar form as metrics. One possible measure could be the *vocabulary complexity* of rules. We might use a threshold value which should not be exceeded for any rules. This threshold value can be based on the scope of the rules.

## 4.2 Deriving Plans and Primitives for Refinement

The perspectives and styles of modelling rule the kind of refinement styles. As an example we consider structure-oriented strategies of development depicted in Figure 3:

**Inside-out refinement:** Inside-out refinement uses the given ASM machine for extending it by additional part. These parts are hooked onto the current specification without changing it.

**Top-down refinement:** Top-down refinement uses decomposition of functions in the vocabulary and refinement of rules. Additionally, the ASM may be extended by functions and rules that havenot yet been considered.

**Bottom-up refinement:** Bottom-up refinement uses composition and generalisation of functions and of rules to more general or complex ones. Bottom-up refinement also uses generation of new functions and rules that have not yet been considered.

**Modular refinement:** Modular refinement is based on parqueting of applications and separation of concern. Refinement is only applied to one module and does not affect others. Modules may also be decomposed.

**Mixed skeleton-driven refinement:** Mixed refinement is a combination of refinement techniques. It uses a skeleton of the application or a draft of the architecture. This draft is used for deriving plans for refinement. Each component or module is developed on its own based on top-down or bottom-up refinement.

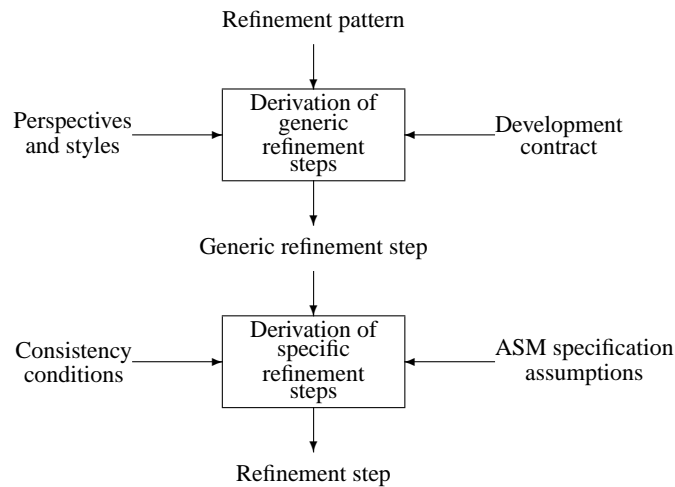
These different kinds of refinement styles allow one to derive *plans* for refinement and *primitives* for refinement.

## 4.3 Generic Refinement Steps and Their Correctness

[Bör03,Sch05] have developed a general theory to refinement. Control of correctness of refinement takes into account (a) a notion of refined state and refined vocabulary, (b) a restriction to states of interest, (c) abstract computation segments, (d) a description of locations of interest, and (e) an equivalence relation among those states of interest. The theory developed in [Bör03,Sch05] allows to check whether a given refinement is correct or not.

A typical engineering approach to development of work products such as programs or specifications is based on a general methodology, operations for specification evolution, and a specification of restrictions to the modelling itself. Each evolution step must either be correct according to some correctness criterion or must lead to obligations that can be used for later correction of the specification. The correctness of a refinement step is defined in terms of two given ASM together with the equivalence relations. Already in [Sch05] it has observed that refinement steps can be governed by contracts. We may consider a number of governments [BST06] in the sense of [Cho82]. However we should take into account the choices for style and perspectives.

Given a refinement pattern, perspectives, styles and contract, we may derive generic refinement steps such as data refinement, purely incremental refinement, submachine refinement, and (m,n) refinement. The generic refinement is adapted to the assumptions made for the given application and to consistency conditions. Typically such consistency are binding conditions of rules to state and vocabulary through the scope of rules. The general approach we envision is depicted in Figure 5.



**Fig. 5.** The Derivation of Correct Refinement Steps

We are currently developing a number of refinement steps that take preconditions for their enactment and use postconditions for their deployment. The derivation of pre- and postconditions and of is based on principles used for government and binding.

## References

- [Bjo06] D. Bjorner. *Software Engineering 3: Domains, requirements, and software design*. Springer, Berlin, 2006.
- [BM97] E. Börger, , and L. Mearelli. Integrating ASM into the software development life cycle. *J. Universal Computer Science*, 3(5):603–665, 1997.
- [Boe06] B. Boehm. A view of 20th and 21st century software engineering. In *Proc. ICSE'06*, pages 12–29, ACM Press, 2006.
- [Bör03] E. Börger. The ASM refinement method. *Formal Aspects of Computing*, 15:237–257, 2003.
- [BS00] E. Börger and W. Schulte. *Architecture Design and Validation Methods*, chapter Modular design for the Java virtual machine architecture, pages 297–357. Springer, Berlin, 2000.
- [BS03] E. Börger and R. Stärk. *Abstract state machines - A method for high-level system design and analysis*. Springer, Berlin, 2003.
- [BST06] A. Bienemann, K.-D. Schewe, and B. Thalheim. Towards a theory of genericity based on government and binding. In *Proc. ER'06, LNCS 4215*, pages 311–324. Springer, 2006.
- [Cho82] N. Chomsky. *Some concepts and consequences of the theory of government and binding*. MIT Press, 1982.
- [Hei96] L. J. Heinrich. *Informationsmanagement: Planung, Überwachung und Steuerung der Informationsinfrastruktur*. Oldenbourg Verlag, München, 1996.
- [HM03] D. Harel and R. Marelly. *Come, Let's play: Scenario-based programming using LSCs and the play-engine*. Springer, Berlin, 2003.

- [HT00] A. Hunt and D. Thomas. *The pragmatic programmer - From Journeyman to master*. Addison-Wesley, Boston, 2000.
- [KT06] R. Kaschek and B. Thalheim. Towards a theory of conceptual modelling. Submitted for publication, 2006.
- [Mey88] B. Meyer. *Object-oriented software construction*. Prentice Hall, New York, 1988.
- [Sch05] G. Schellhorn. ASM refinement and generalizations of forward simulation in data refinement: A comparison. *Theor. Comput. Sci.*, 336(2-3):403–435, 2005.
- [Sie04] J. Siedersleben. *Moderne Softwarearchitektur*. dpunkt-Verlag, Heidelberg, 2004.
- [SSB01] R. Stärk, J. Schmid, and E. Börger. *Java and the Java virtual machine*. Springer, Berlin, 2001.
- [ST05] K.-D. Schewe and B. Thalheim. Conceptual modelling of web information systems. *Data and Knowledge Engineering*, 54:147–188, 2005.
- [ST07] K.-D. Schewe and B. Thalheim. Development of collaboration frameworks for web information systems. In *IJCAI'07 (20th Int. Joint Conf on Artificial Intelligence, Section EMC'07 (Evolutionary models of collaboration)*, pages 27–32, Hyderabad, 2007.
- [Sta92] H. Stachowiak. Modell. In Helmut Seiffert and Gerard Radnitzky, editors, *Handlexikon Zur Wissenschaftstheorie*, pages 219–222. Deutscher Taschenbuch Verlag GmbH & Co. KG, München, 1992.
- [Stä04] R. Stärk. Abstract state machines: A method for high-level design and analysis - Lectures given at ETH Zürich. <http://www.inf.ethz.ch/~staerk/asm04/>, 2004.
- [Tha00] B. Thalheim. *Entity-relationship modeling – Foundations of database technology*. Springer, Berlin, 2000.
- [Tha03] B. Thalheim. Informationssystem-Entwicklung. In *BTU Cottbus, Computer Science Institute, Technical Report I-15-2003*, Cottbus, 2003.
- [Tha05] B. Thalheim. Component development and construction for database design. *Data and Knowledge Engineering*, 54:77–95, 2005.
- [Wal97] C. Wallace. The semantics of the Java programming language. Technical Report CSE-TR-355-97, University of Michigan, EECS Dept., December 1997.
- [Who80] B.L. Whorf. *Lost generation theories of mind, language, and religion*. Popular Culture Association, University Microfilms International, Ann Arbor, Mich., 1980.
- [ZT04] W. Zimmermann and B. Thalheim. Preface. In *ASM 2004*, number 3052 in LNCS, pages V–VII, Berlin, 2004. Springer.

Remark: This research proposal is an answer to an email exchange between Daniel Klünder and Andreas Prinz who summarised: “Engineering or modelling of ASM itself has not yet given the right attention.” This paper attempts in development of a general ASM modelling approach.

Acknowledgement: We are very thankful to our reviewers and S. Hegner for their detailed, fruitful and challenging proposals, remarks and requests.