Refinement and Implementation Techniques for Abstract State Machines

Dissertation
zur Erlangung des Grades eines Doktors
der Naturwissenschaften (Dr. rer. nat.)
im Fachbereich Informatik
der Universität Ulm

vorgelegt von

Joachim Schmid

aus Tuttlingen

Abteilung für Programmiermethodik und Compilerbau
(Leiter: Prof. Dr. Helmuth Partsch)

2002

Amtierender Dekan: Prof. Dr. Günther Palm

Gutachter: Prof. Dr. Helmuth Partsch (Universität Ulm)
Prof. Dr. Friedrich von Henke (Universität Ulm)
Prof. Dr. Egon Börger (Universität Pisa)

Tag der Prüfung: 17. Juni 2002

Danksagung

An dieser Stelle möchte ich mich bei allen Personen bedanken, die mich bei der Erstellung dieser Arbeit unterstützt haben.

Besonderen Dank gilt Herrn Prof. Dr. Egon Börger, der stets ein offenes Ohr für Fragen und Diskussionen hatte und mich zu dieser Arbeit motivierte.

Bedanken möchte ich mich für die tatkräftige Unterstützung durch Dr. Peter Päppinghaus, der meine Promotion bei der Siemens AG betreute. Dank gilt auch allen anderen Mitarbeitern von CT SE 4, welche manchmal von ihrer Arbeit abgehalten wurden.

Dank gilt der Siemens AG, die mir die notwendigen Arbeitsmittel zur Verfügung stellte und mich finanziell unterstüzte. Insbesondere möchte ich mich bei dem Fachzentrumsleiter Prof. Dr. Wolfram Büttner bedanken, der sich bereit erklärt hatte, diese Dissertation durch die Siemens AG zu unterstützen.

Contents

Introduction

The notion of Abstract State Machines (ASMs), defined by Gurevich in [33, 34], has been used successfully for the design and the analysis of numerous complex software and hardware systems (see [16] for an overview). The outstanding features which are responsible for this success are the simple yet most general notion of state—namely mathematical structures, providing arbitrary abstract data types—together with the simultaneous execution of multiple atomic actions as the notion of state transforming basic machine step.

This view, of multiple atomic actions executed in parallel in a common state, comes however at a price: the standard composition and structuring principles, which are needed for high-level system design and programming in the large, are not directly supported and can be introduced only as refinement steps. Also the freedom in the choice of the appropriate data structures, which provides a powerful mechanism to describe systems at various levels of abstraction, has a price: it implies an additional step to turn these abstractions into executable models, so that experimental validation can be done via simulation.

In this thesis we enhance the usefulness of ASMs for software engineering practice by:

- defining three major composition concepts for ASMs, namely component machine, parameterized submachine, and iteration, which cannot be dispensed with in software engineering (Chap. 1, Chap. 2)
- developing a tool that allows to execute these extended ASMs and comes with a graphical user interface, to support experimental analysis (Chap. 3, Chap. 4)

We have tested the practicality of the proposed concepts and of their implementation (Chap. 5), namely by the design and the analysis

- of a well-known software engineering case study in the literature and of a middle sized industrial software development project (for a train timetable construction and validation system). This project work also included the development of a proprietary compiler from ASMs to *C++*.
- of the Java language and its implementation on the Java Virtual Machine
- of an industrial ASIC design and verification project. This project work also included the development of a compiler from ASM components to VHDL.

Part of this work has been published already in [17, 18, 19, 61, 63, 64].

Notational conventions

Throughout this thesis we stick to standard mathematical terminology. Nevertheless, we list here some frequently used notations.

We write $\mathbb{P}(X)$ for the set of all subsets of X.

We write $X \lhd f$ for the domain restriction of the function f to the set X:

$$X \lhd f := \{(x, f(x)) \mid x \in \mathrm{dom}(f) \cap X\}$$

We use ε for the empty sequence and $\cdot$ to separate elements in a sequence. Further, we write $R^{\geq n}$ for the universe containing all sequences of R with length greater or equal than n.

$$\begin{aligned} R^{\geq n} &= R^n \cup R^{n+1} \cup \ldots \\ R^n &= \underbrace{R \cdot \ldots \cdot R}_{n} \end{aligned}$$

Since only some functions are partial in this thesis we denote them by the symbol $\nrightarrow$ instead of symbol $\rightarrow$ for total functions. For example:

$$f \colon R_1 \nrightarrow R_2$$

The composition of functions is denoted by $f \circ g$.

We use $\mathbb{N}_n$ to denote the set of natural numbers from zero to n.

Chapter 1

Submachine Concept

It has often been observed that Gurevich's definition of Abstract State Machines (ASMs) [33] uses only conditional assignments and supports none of the classical control or data structures. On the one side this leaves the freedom—necessary for *high-level* system design and analysis—to introduce during the modeling process any control or data structure whatsoever which may turn out to be suitable for the application under study. On the other hand it forces the designer to specify standard structures over and over again when they are needed, at the latest when it comes to implement the specification. In this respect ASMs are similar to Abrial's Abstract Machines [1] which are expressed by non-executable pseudo-code without sequencing or loop (Abstract Machine Notation, AMN). In particular there is no notion of submachine and no calling mechanism. For both Gurevich's ASMs and Abrial's Abstract Machines, various notions of refinement have been used to introduce the classical control and data structures. See for example the definition in [37] of recursion as a distributed ASM computation (where calling a recursive procedure is modeled by creating a new instance of multiple agents executing the program for the procedure body) and the definition in [1, 12.5] of recursive AMN calls of an operation as calls to the operation of importing the implementing machine.

Operations of B-Machines [1] and of ASMs come in the form of atomic actions. The semantics of ASMs provided in [33] is defined in terms of a function *next* from states (structures) to states which reflects one step of machine execution. We extend this definition to a function describing, as one step, the result of executing an a priori unlimited number n of basic machine steps. Since n could go to infinity, this naturally leads to consider also non halting computations. We adapt this definition to the view of simultaneous atomic updates in a global state, which is characteristic for the semantics of ASMs, and avoid prescribing any specific syntactic form of encapsulation or state hiding. This allows us to integrate the classical control constructs for *sequentialization and iteration* into the global state based ASM view of computations. Moreover this can be done in a compositional way, supporting the corresponding well known structured proof principles for proving properties for complex machines in terms of properties of their components. We illustrate this by providing structured ASMs for computing arbitrary computable functions, in a way which combines the advantages of functional and of imperative programming. The atomicity of the ASM iteration constructor we define below turned out to be the key for a

rigorous definition of the semantics of event triggered exiting from compound actions of UML activity and state machine diagrams, where the intended instantaneous effect of exiting has to be combined with the request to exit nested diagrams sequentially following the subdiagram order, see [11, 12].

For structuring large ASMs extensive use has been made of macros as notational shorthands. We enhance this use here by defining the semantics of *named parameterized ASM rules* which include also recursive ASMs. Aiming at a foundation which supports the practitioners' procedural understanding and use of submachine calls, we follow the spirit of the basic ASM concept [33] where domain theoretic complications—arising when explaining what it means to iterate the execution of a machine "until ..."—have been avoided, namely by defining only the one-step computation relation and by relegating fixpoint ("termination") concerns to the metatheory. Therefore we define the semantics of submachine calls only for the case that the possible chain of nested calls of that machine is finite. We are thus led to a notion of calling submachines which mimics the standard imperative calling mechanism and can be used for a definition of recursion in terms of *sequential* (not distributed) ASMs. This definition suffices to justify the submachines used in [64] for a hierarchical decomposition of the Java Virtual Machine into loading, verifying, and executing machines for the five principal language layers (imperative core, static classes, object oriented features, exception handling, and concurrency).

The third kind of structuring mechanism for ASMs we consider in this paper is of syntactical nature, dealing essentially with name spaces. Parnas' [56] information hiding principle is strongly supported by the ASM concept of external functions which provides also a powerful interface mechanism (see [10]). A more syntax oriented form of information hiding can be naturally incorporated into ASMs through the notion of *local machine state*, of machines with *return values* and of *error handling* machines which we introduce in Section 1.4.

1.1 Standard ASMs

We start from the definition of basic sequential (i.e. non distributed) ASMs in [33] and survey in this section our notation.

Basic ASMs are built up from *function updates* and *skip* by *parallel composition* and constructs for *if then else*, *let* and *forall*. We consider the *choose*-construct as a special notation for using choice functions, a special class of external functions. Therefore we do not list it as an independent construct in the syntactical definition of ASMs. It appears however in the appendix because the non-deterministic selection of the *choose*-value is directly related to the non-deterministic application of the corresponding deduction rule.

The interpretation of an ASM in a given state $\mathfrak{A}$ depends on the given environment *Env*, i.e. the interpretation $\zeta \in Env$ of its free variables. We use the standard interpretation $[\![t]\!]^{\mathfrak{A}}_{\zeta}$ of terms t in state $\mathfrak{A}$ under variable interpretation ζ, but we often suppress mentioning the underlying interpretation of variables. The semantics of standard ASMs is defined in [33] by assigning to each rule R, given a state $\mathfrak{A}$ and a variable interpretation ζ, an update set $[\![R]\!]^{\mathfrak{A}}_{\zeta}$ which—if consistent—is fired in state $\mathfrak{A}$ and produces the next state $next_R(\mathfrak{A}, \zeta)$.

An update set is a set of *updates*, i.e. a set of pairs (loc, val) where *loc* is a location and *val* is an element in the domain of $\mathfrak{A}$ to which the location is

intended to be updated. A location is n-ary function name f with a sequence of length n of elements in the domain of $\mathfrak{A}$, denoted by $f\langle a_1, \ldots, a_n\rangle$. If u is an update set then $Locs(u)$ denotes the set of locations occurring in elements of u ($Locs(u) = \{loc \mid \exists\, val : (loc, val) \in u\}$). An update set u is called *inconsistent* if u contains at least two pairs (loc, v_1) and (loc, v_2) with $v_1 \neq v_2$ (i.e. $|u| > |Locs(u)|$), otherwise it is called *consistent*.

For a consistent update set u and a state $\mathfrak{A}$, the state $fire_{\mathfrak{A}}(u)$, resulting from firing u in $\mathfrak{A}$, is defined as state $\mathfrak{A}'$ which coincides with $\mathfrak{A}$ except $f^{\mathfrak{A}'}(a) = val$ for each $(f\langle a\rangle, val) \in u$. Firing an inconsistent update set is not allowed, i.e. $fire_{\mathfrak{A}}(u)$ is not defined for inconsistent u. This definition yields the following (partial) next state function $next_R$ which describes *one* application of R in a state with a given environment function $\zeta \in Env$. Sometimes, we omit the environment ζ.

$$\begin{aligned} &next_R && : State(\Sigma) \times Env \to State(\Sigma) \\ &next_R(\mathfrak{A}, \zeta) && = fire_{\mathfrak{A}}([\![R]\!]^{\mathfrak{A}}_{\zeta}) \end{aligned}$$

The following definitions describe the meaning of standard ASMs. We use R and S for rules, x for variables, s and t for expressions, p for predicates (boolean expressions), and u, v for semantical values and update sets. We write $f^{\mathfrak{A}}$ for the interpretation of the function f in state $\mathfrak{A}$ and $\zeta' = \zeta\frac{x}{u}$ is the variable environment which coincides with ζ except for x where $\zeta'(x) = u$.

$$\begin{aligned}
&[\![x]\!]^{\mathfrak{A}}_{\zeta} && = \zeta(x) \\
&[\![f(t_1, \ldots, t_n)]\!]^{\mathfrak{A}}_{\zeta} && = f^{\mathfrak{A}}([\![t_1]\!]^{\mathfrak{A}}_{\zeta}, \ldots, [\![t_n]\!]^{\mathfrak{A}}_{\zeta}) \\
&[\![\textbf{skip}\ ]\!]^{\mathfrak{A}}_{\zeta} && = \varnothing \\
&[\![f(t_1, \ldots, t_n) := s]\!]^{\mathfrak{A}}_{\zeta} && = \{(f\langle [\![t_1]\!]^{\mathfrak{A}}_{\zeta}, \ldots, [\![t_n]\!]^{\mathfrak{A}}_{\zeta}\rangle, [\![s]\!]^{\mathfrak{A}}_{\zeta})\} \\
&[\![\{R_1, \ldots, R_n\}]\!]^{\mathfrak{A}}_{\zeta} && = [\![R_1]\!]^{\mathfrak{A}}_{\zeta} \cup \cdots \cup [\![R_n]\!]^{\mathfrak{A}}_{\zeta} \\
&[\![\textbf{if } t \textbf{ then } R \textbf{ else } S]\!]^{\mathfrak{A}}_{\zeta} && = \begin{cases} [\![R]\!]^{\mathfrak{A}}_{\zeta}, & \text{if } [\![t]\!]^{\mathfrak{A}}_{\zeta} = \text{true}^{\mathfrak{A}} \\ [\![S]\!]^{\mathfrak{A}}_{\zeta}, & \textbf{otherwise} \end{cases} \\
&[\![\textbf{let } x = t \textbf{ in } R]\!]^{\mathfrak{A}}_{\zeta} && = [\![R]\!]^{\mathfrak{A}}_{\zeta\frac{x}{v}} \text{ where } v = [\![t]\!]^{\mathfrak{A}}_{\zeta} \\
&[\![\textbf{forall } x \textbf{ with } p \textbf{ do } R]\!]^{\mathfrak{A}}_{\zeta} && = \bigcup_{v \in V} [\![R]\!]^{\mathfrak{A}}_{\zeta\frac{x}{v}} \text{ where } V = \{v \mid [\![p]\!]^{\mathfrak{A}}_{\zeta\frac{x}{v}} = \text{true}^{\mathfrak{A}}\}
\end{aligned}$$

Remark 1.1 Usually the parallel composition $\{R_1, \ldots, R_n\}$ of rules R_i is denoted by displaying the R_i vertically one above the other.

For a standard ASM R, the update set $[\![R]\!]^{\mathfrak{A}}_{\zeta}$ is defined for any state $\mathfrak{A}$ and for any variable environment ζ, but $next_R(\mathfrak{A}, \zeta)$ is undefined if $[\![R]\!]^{\mathfrak{A}}_{\zeta}$ is inconsistent.

1.2 Sequential Composition and Iteration

The basic composition of ASMs is parallel composition (see [35]). It is for practical purposes that in this section we incorporate into ASMs their sequential composition and their iteration, but in a way which fits the basic paradigm of parallel execution of all the rules of a given ASM. The idea is to treat the sequential execution P **seq** Q of two rules P and Q as an "atomic" action, in the same way as executing a function update $f(t_1, \ldots, t_n) := s$, and similarly for the iteration **iterate**(R) of rule R, i.e. the repeated application of sequential composition of R with itself, as long as possible. The notion of repetition yields

a definition of the traditional **while** (*cond*) R construct which is similar to its proof theoretic counterpart in [1, 9.2.1]. Whereas Abrial explicitly excludes sequencing and loop from the specification of abstract machines [1, pg. 373], we take a more pragmatic approach and define them in such a way that they can be used coherently in two ways, depending on what is needed, namely to provide black-box descriptions of abstract submachines or glass-box views of their implementation (refinement).

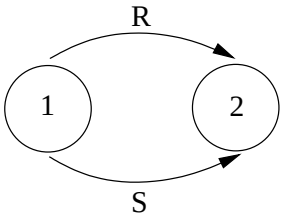
1.2.1 Sequence Constructor

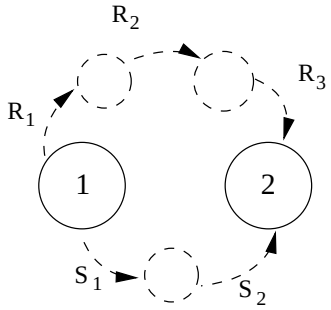
If one wants to specify executing one standard ASM after another, this has to be explicitly programmed. Consider for example the function *pop_back* in the Standard Template Library for C++ (abstracting from concrete data structures). The function deletes the last element in a list. Assume further that we have already defined rules *move_last* and *delete* where *move_last* sets the list pointer to the last element and *delete* removes the current element. One may be tempted to program *pop_back* as follows to first execute *move_last* and then *delete*:

pop_back ≡
  if *mode* = *Move* **then**
    move_last
    mode := *Delete*
  if *mode* = *Delete*
    delete
    mode := *Move*

This definition has the drawback that the user of *pop_back* must know that the action to be completed needs two steps, which really is an implementation feature. Moreover the dynamic function *mode*, which is used to *program* the sequential ordering, is supposed to be initialized by *Move*. Such an explicit programming of execution order quickly becomes a stumbling block for large specifications, in particular the initialization is not easily guaranteed without introducing an explicit initialization mechanism.

Another complication arises when sequentialized rules are used to refine abstract machines. In the machine on the lefthand side of the picture below, assume that the simultaneous execution of the two rules R and S in state 1 leads to state 2. The machine on the right side is supposed to refine the machine on the lefthand side with rules R and S refined into the sequence of rules $R_1R_2R_3$ and S_1S_2 respectively. There is no obvious general scheme to interleave the R_i-rules and the S_j-rules, using a *mode* function as above. What should happen if rule R_2 modifies some locations which are read by S_2? In such cases R and S could not be refined independently of each other.





Therefore we introduce a sequence constructor yielding a rule P **seq** Q which can be inserted into another ASM but whose semantical effect is nevertheless the sequential execution of the two rules P and Q. If the new rule P **seq** Q has to share the same status as any other ASM rule together with which it may be executed in parallel, one can define the execution of P **seq** Q only as an atomic action. Obviously this is only a way to "view" the sequential machine from outside; its refined view reveals its internal structure and behavior, constituted by the non atomic execution, namely in two steps, of first P and then Q.

Syntactically the sequential composition P **seq** Q of two rules P and Q is defined to be a rule. The semantics is defined as first executing P, obtaining an intermediate state, followed by executing Q in the intermediate state. This is formalized by the following definition of the update set of P **seq** Q in state $\mathfrak{A}$.

Definition 1.2.1 (Sequential execution) *Let P and Q be rules. We define*

$$[\![P \textbf{ seq } Q]\!]^{\mathfrak{A}} = [\![P]\!]^{\mathfrak{A}} \oplus [\![Q]\!]^{\mathfrak{A}'}$$

where $\mathfrak{A}' = next_P(\mathfrak{A})$ is the state obtained by firing the update set of P in state $\mathfrak{A}$, if $next_P(\mathfrak{A})$ is defined; otherwise $\mathfrak{A}'$ can be chosen arbitrarily.

The notion $u \oplus v$ denotes the merging of the update set v with update set u where updates in v overwrite updates in u. We merge an update set v with u only if u is consistent, otherwise we stick to u because then we want both $fire_{\mathfrak{A}}(u)$ and $fire_{\mathfrak{A}}(u \oplus v)$ to be undefined.

$$u \oplus v = \begin{cases} \{(loc, val) \mid (loc, val) \in u \wedge loc \notin Locs(v)\} \cup v, & consistent(u) \\ u, & \textbf{otherwise} \end{cases}$$

Proposition 1.2.1 (Persistence of inconsistency) *If the update set $[\![P]\!]^{\mathfrak{A}}$ is not consistent, then $[\![P \textbf{ seq } Q]\!]^{\mathfrak{A}} = [\![P]\!]^{\mathfrak{A}}$*

The next proposition shows that the above definition of the **seq** constructor captures the intended classical meaning of sequential composition of machines, if we look at them as state transforming functions[1]. Indeed we could have defined **seq** via the composition of algebra transforming functions, similarly to its axiomatically defined counterpart in Abrial's AMN [1] where **seq** comes as concatenation of generalized substitutions.

[1] We assume that $f(x)$ is undefined if x is undefined, for every function f (f is strict).

Proposition 1.2.2 (Compositionality of seq)

$$next_{P\,\mathbf{seq}\,Q} = next_Q \circ next_P$$

This characterization illustrates that **seq** has the expected semiring properties on update sets:

Proposition 1.2.3 *The ASM constructor* **seq** *has a left and a right neutral element and is associative, i.e. for rules P, Q, and R the following holds:*

$$[\![\mathbf{skip}\ \ \mathbf{seq}\ R]\!]^{\mathfrak{A}} = [\![R\ \mathbf{seq}\ \mathbf{skip}\ ]\!]^{\mathfrak{A}} = [\![R]\!]^{\mathfrak{A}} \quad \textit{(left and right neutral)}$$
$$[\![P\ \mathbf{seq}\ (Q\ \mathbf{seq}\ R)]\!]^{\mathfrak{A}} = [\![(P\ \mathbf{seq}\ Q)\ \mathbf{seq}\ R]\!]^{\mathfrak{A}} \quad \textit{(associative)}$$

1.2.2 Iteration Constructor

Once a sequence operator is defined, one can apply it repeatedly to define the iteration of a rule. This provides a natural way to define for ASMs an iteration construct which encapsulates a computation with a finite but a priori not explicitly known number of iterated steps into an atomic action (one-step computation). As a by-product we obtain the classical loop and while constructs, cf. [1, 9.2].

The intention of rule iteration is to execute the given rule again and again—as long as *needed* and as long as *possible*. We define

$$R^n = \begin{cases} \mathbf{skip}\ , & n = 0 \\ R^{n-1}\ \mathbf{seq}\ R, & n > 0 \end{cases}$$

Denote by $\mathfrak{A}_n$ the state obtained by firing the update set of the rule R^n in state $\mathfrak{A}$, if defined (i.e. $\mathfrak{A}_n = next_{R^n}(\mathfrak{A})$).

There are obviously two stop situations for iterated ASM rule application, namely when the update set becomes empty (the case of successful termination) and when it becomes inconsistent (the case of failure, given the persistence of inconsistency as formulated in Proposition 1.2.1).[2] Both cases provide a fixpoint $\lim_{n\to\infty} [\![R^n]\!]^{\mathfrak{A}}$ for the sequence $([\![R^n]\!]^{\mathfrak{A}})_{n>0}$ which becomes stable if a number n is found where the update set of R, in the state obtained by firing R^{n-1}, is empty or inconsistent.

Proposition 1.2.4 (Fixpoint Condition) *$\forall\, m \geq n > 0$ the following holds: if $[\![R]\!]^{\mathfrak{A}_{n-1}}$ is not consistent or if it is empty, then $[\![R^m]\!]^{\mathfrak{A}} = [\![R^n]\!]^{\mathfrak{A}}$*

Therefore we extend the syntax of ASM rules by **iterate**(R) to denote the iteration of rule R and define its semantics as follows.

Definition 1.2.2 (Iteration) *Let R be a rule. We define*

$$[\![\mathbf{iterate}(R)]\!]^{\mathfrak{A}} = \lim_{n\to\infty} [\![R^n]\!]^{\mathfrak{A}}, \quad \textit{if } \exists\ n \geq 0 : [\![R]\!]^{\mathfrak{A}_n} = \varnothing \vee \neg \textit{consistent}([\![R]\!]^{\mathfrak{A}_n})$$

[2] We do not include here the case of an update set whose firing does not change the given state, although including this case would provide an alternative stop criterion for implementations of ASMs.

The sequence $(\llbracket R^n \rrbracket^{\mathfrak{A}})_{n>0}$ eventually becomes stable only upon termination or failure. Otherwise the computation diverges and the update set for the iteration is undefined. An example for a machine R which naturally produces a diverging (though in other contexts useful) computation is **iterate**$(a := a+1)$, see [46, Exl. 2, pg. 350].

Example 1.2.1 (Usage of iterate) The ASM model for Java in [20] includes the initialization of classes which in Java is done implicitly at the first use of a class. Since the Java specification requires that the superclass of a class c is initialized before c, the starting of the class initialization is iterated until an initialized class c' is encountered (i.e. satisfying $\mathit{initialized}(c')$, as eventually will happen towards the top of the class hierarchy). We define the initialization of class *class* as follows:

$\mathit{initialize} \equiv$
  $c := \mathit{class}$ **seq iterate**(**if** $\neg \mathit{initialized}(c)$ **then**
    $\mathit{createInitFrame}(c)$
  if $\neg \mathit{initialized}(\mathit{superClass}(c))$ **then**
    $c := \mathit{superClass}(c)$)

The finiteness of the acyclic class hierarchy in Java guarantees that this rule yields a well defined update set. The rule abstracts from the standard sequential implementation (where obviously the class initialization is started in a number of steps depending on how many super classes the given class has which are not yet initialized) and offers an atomic operation to push all initialization methods in the right order onto the frame stack (the frame stack contains the method calls).

The macro to create new initialization frames can be defined as follows. The current computation state, consisting of *method*, *program*, program position *pos* and *localVars*, is pushed onto the *frames* stack and is updated for starting the initialization method of the given *class* at position 0 with empty local variables set.

$\mathit{createInitFrame}(c) \equiv$
  $\mathit{classState}(c) := \mathit{InProgress}$
  $\mathit{frames} := \mathit{frames} \cdot (\mathit{method}, \mathit{program}, \mathit{pos}, \mathit{localVars})$
  $\mathit{method} := c/$`<clinit>`
  $\mathit{program} := \mathit{body}(c/$`<clinit>`$)$
  $\mathit{pos} := 0$
  $\mathit{localVars} := \varnothing$

While and Until. The iteration yields a natural definition of while loops. A *while loop* repeats the execution of the *while body* as long as a certain condition holds.

while $(\mathit{cond})\ R =$ **iterate**(**if** cond **then** R)

This *while* loop, if started in state $\mathfrak{A}$, terminates if eventually $\llbracket R \rrbracket^{\mathfrak{A}_n}$ becomes empty or the condition *cond* becomes *false* in $\mathfrak{A}_n$ (with consistent and non empty previous update sets $\llbracket R \rrbracket^{\mathfrak{A}_i}$ and previous states $\mathfrak{A}_i$ satisfying *cond*). If the iteration of R reaches an inconsistent update set (failure) or yields an infinite

sequence of consistent non empty update sets, then the state resulting from executing the while loop starting in $\mathfrak{A}$ is not defined (divergence of the while loop). Note that the function $next_{\mathbf{while}(cond)\,R}$ is undefined in these two cases on $\mathfrak{A}$.

A *while* loop may satisfy more than one of the above conditions, like **while** (*false*) **skip** . The following examples illustrate the typical four cases:

- (success) **while** (*cond*) **skip**
- (success) **while** (*false*) R
- (failure) **while** (*true*) $a := 1$
  $a := 2$
- (divergence) **while** (*true*) $a := a$

Example 1.2.2 (Usage of while)
The following iterative ASM defines a while loop to compute the factorial function for given argument x and stores the result in a location *fac*. It uses multiplication as given (static) function. We will generalize this example in the next section to an ASM analogue to the Böhm-Jacopini theorem on structured programming [8].

$$compute\_fac \equiv (fac := 1)\ \mathbf{seq}\ (\mathbf{while}\ (x > 0)\ fac := x * fac \quad x := x - 1)$$

Remark 1.2 As usual one can define the *until loop* in terms of **while** and **seq** as first executing the body once and then behaving like a while loop:

$$\mathbf{do}\ R\ \mathbf{until}\ (cond) = R\ \mathbf{seq}\ (\mathbf{while}\ (\neg cond)\ R).$$

The sequencing and iteration concepts above apply in particular to the *Mealy-ASMs* defined in [10] for which they provide the sequencing and the feedback operators. The fundamental parallel composition of ASMs provides the concept of parallel composition of Mealy automata for free. These three constructs allow one to apply to Mealy-ASMs the decomposition theory which has been developed for finite state machines in [22].

1.2.3 Böhm-Jacopini ASMs

The sequential and iterative composition of ASMs yields a class of machines which are known from [8] to be appropriate for the computation of partial recursive functions. We illustrate in this section how these *Böhm-Jacopini-ASMs* naturally combine the advantages of the Gödel-Herbrand style functional definition of computable functions and of the Turing style imperative description of their computation.

Let us call Böhm-Jacopini-ASM any ASM which can be defined, using the sequencing and the iterator constructs, from basic ASMs whose functions are restricted as defined below to input, output, controlled functions and some simple static functions. For each Böhm-Jacopini-ASM M we allow only one external function, a 0-ary function for which we write in_M. The purpose of this function is to contain the number sequence which is given as input for the computation of the machine. Similarly we write out_M for the unique (0-arp) function which will be used to receive the output of M. Adhering to the usual

practice one may also require that the M-output function appears only on the lefthand side of M-updates, so that it does not influence the M-computation and is not influenced by the environment of M. As static functions we admit only the *initial* functions of recursion theory, i.e. the following functions from Cartesian products of natural numbers into the set $\mathbb{N}$ of natural numbers: $+1$, all the projection functions U_i^n, all the constant functions C_i^n and the characteristic function of the predicate $\neq 0$.

Following the standard definition we call a number theoretic function $f : \mathbb{N}^n \to \mathbb{N}$ computable by an ASM M if for every n-tuple $x \in \mathbb{N}^n$ of arguments on which f is defined, the machine started with input x terminates with output $f(x)$. By "M started with input x" we mean that M is started in the state where all the dynamic functions different from in_M are completely undefined and where $in_M = x$. Assuming the external function in_M not to change its value during an M-computation, it is natural to say that M terminates in a state with output y, if in this state out_M gets updated for the first time, namely to y.

Proposition 1.2.5 (Structured Programming Theorem)
Every partial recursive function can be computed by a Böhm-Jacopini-ASM.

Proof. We define by induction for each partial recursive function f a machine F computing it. Each initial function f of recursion theory is computed by the following machine F consisting of only one function update which reflects the defining equation of f. The following machine F is well-defined, since it contains a singleton function update and therefore, the update set is always consistent.

$$F \equiv out_F := f(in_F)$$

For the inductive step it suffices to construct, for any partial recursive definition of a function f from its constituent functions f_i, a machine F which mimics the standard evaluation procedure underlying that definition. We define the following macros for using a machine F for given arguments in, possibly including to assign its output to a location out:

$$\begin{aligned} F(in) \qquad &\equiv in_F := in \textbf{ seq } F \\ out := F(in) &\equiv F(in) \textbf{ seq } out := out_F \end{aligned}$$

We start with the case of function composition. If functions $g, h_1, \ldots, h_m$ are computed by Böhm-Jacopini-ASMs $G, H_1, \ldots, H_m$, then their composition f defined by $f(x) = g(h_1(x), \ldots, h_m(x))$ is computed by the following machine F:

$$F \equiv \{H_1(in_F), \ldots, H_m(in_F)\} \textbf{ seq } out_F := G(out_{H_1}, \ldots, out_{H_m})$$

For reasons of simplicity but without loss of generality we assume that the submachines have pairwise disjoint signatures. Hence, the machine F is well-defined, since the locations in the update sets of $H_1, \ldots, H_m$ are disjoint.

Unfolding this structured program reflects the order one *has* to follow for evaluating the subterms in the defining equation for f, an order which is implicitly assumed in the equational (functional) definition. First the input is passed to the constituent functions h_i to compute their values, whereby the input functions of H_i become controlled functions of F. The parallel composition of the

submachines $H_i(in_F)$ reflects that any order is allowed here. Then the sequence of out_{H_i} is passed as input to the constituent function g. Finally g's value on this input is computed and assigned as output to out_F.

Similarly let a function f be defined from g, h by primitive recursion:

$$f(x,0) = g(x), \quad f(x,y+1) = h(x,y,f(x,y))$$

and let Böhm-Jacopini-ASMs G, H be given which compute g, h. Then the following machine F computes f, composed as sequence of three submachines. The start submachine of F evaluates the first defining equation for f by initializing the recursor *rec* to 0 and the intermediate value *ival* to $g(x)$. The *while* submachine evaluates the second defining equation for f for increased values of the recursor as long as the input value y has not been reached. The output submachine provides the final value of *ival* as output.

$F \equiv$ **let** $(x,y) = in_F$ **in**
  $\{ival := G(x),\ rec := 0\}$ **seq**
  (**while** $(rec < y)\,\{ival := H(x, rec, ival),\ rec := rec + 1\})$ **seq**
  $out_F := ival$

If f is defined from g by the μ-operator, i.e. $f(x) = \mu\, y(g(x,y) = 0)$, and if a Böhm-Jacopini-ASM G computing g is given, then the following machine F computes f. The start submachine computes $g(x, rec)$ for the initial recursor value 0, the iterating machine computes $g(x, rec)$ for increased values of the recursor until 0 shows up as computed value of g, in which case the reached recursor value is set as output.

$F \equiv \{G(in_F, 0),\ rec := 0\}$ **seq**
  (**while** $(out_G \neq 0)\,\{G(in_F, rec + 1),\ rec := rec + 1\})$ **seq**
  $out_F := rec$

Remark 1.3 The construction of Böhm-Jacopini-ASMs illustrates, through the idealized example of computing recursive functions, how ASMs allow to pragmatically reconcile the often discussed conceptual dichotomy between functional and imperative programming. In the context of discussing the "functional programming language" Gödel used to exhibit undecidable propositions in Principia Mathematica, as opposed to the "imperative programming language" developed by Turing and used in his proof of the unsolvability of the *Entscheidungsproblem* (see [14]), Martin Davis [28] states:

> "The programming languages that are mainly in use in the software industry (like C and FORTRAN) are usually described as being *imperative*. This is because the successive lines of programs written in these languages can be thought of as *commands* to be executed by the computer ... In the so-called *functional* programming languages (like LISP) the lines of a program are definitions of operations. Rather than telling the computer what to do, they *define* what it is that the computer is to provide."

The equations which appear in the Gödel-Herbrand type equational definition of partial recursive functions "define what it is that the computer is to

provide" only within the environment for evaluation of subterms. The corresponding Böhm-Jacopini-ASMs constructed above make this context explicit, exhibiting how to evaluate the subterms when using the equations (updates), as much as needed to make the functional shorthand work correctly. We show in the next section how this use of shorthands for calling submachines, which appear here only in the limited context of structured WHILE programs, can be generalized as to make it practical without loss of rigor.

1.3 Parameterized Machines

For structuring large ASMs extensive use has been made of macros which, semantically speaking, are mere notational shorthands, to be substituted by the body of their definition. We enhance this use here by introducing named parameterized ASM rules which in contrast to macros also support recursive ASMs.

We provide a foundation which justifies the application of named parameterized ASMs in a way which supports the practitioners' procedural understanding. Instead of guaranteeing within the theory, typically through a fixpoint operator, that under certain conditions iterated calls of recursive rules yield as "result" a first-class mathematical "object" (namely the fixpoint), we take inspiration from the way Kleene proved his recursion theorem [46, Section 66] and leave it to the programmer to guarantee that a possibly infinite chain of recursive procedure calls is indeed well founded with respect to some partial order.

We want to allow a named parameterized rule to be used in the same way as all other rules. For example, if f is a function with arity 1 and R is a named rule expecting two parameters, then $R(f(1), 2)$ should be a legitimate rule, too. In particular we want to allow rules as parameters, like in the following example where the given dynamic function *stdout* is updated to "hello world":

rule $R(output) =$
  $output($`"hello world"`$)$

rule $output\_to\_stdout(msg)$
  $stdout := msg$

$R(output\_to\_stdout)$

Therefore we extend the inductive syntactic definition for rules by the following new clause, called a rule application with actual parameters $a_1, \ldots, a_n$:

$R(a_1, \ldots, a_n)$

and coming with a rule definition of the following form:

rule $R(x_1, \ldots, x_n) = body$

where *body* is a rule. R is called the rule name, $x_1, \ldots, x_n$ are the formal parameters of the rule definition. They bind the free occurrences of the variables $x_1, \ldots, x_n$ in *body*.

The basic intuition the practice of computing provides for the interpretation of a named rule is to define its semantics as the interpretation of the rule body with the formal parameters replaced by the actual arguments. In other words

we unfold nested calls of a recursive rule R into a sequence $R_1, R_2, \ldots$ of rule incarnations where each R_i may trigger one more execution of the rule body, relegating the interpretation of possibly yet another call of R to the next incarnation R_{i+1}. This may produce an infinite sequence, namely if there is no ordering of the procedure calls with respect to which the sequence will decrease and reach a basis for the recursion. In this case the semantics of the call of R is undefined. If however a basis for the recursion does exist, say R_n, it yields a well defined value for the semantics of R through the chain of successive calls of R_i; namely for each $0 \leq i < n$ with $R = R_0$, R_i inherits its semantics from R_{i+1}.

Definition 1.3.1 (Named ruled) *Let R be a named rule declared by* **rule** $R(x_1, \ldots, x_n) = body$, *let $\mathfrak{A}$ be a state.*

If $[\![body[a_1/x_1, \ldots, a_n/x_n]]\!]^{\mathfrak{A}}$ is defined, then
$[\![R(a_1, \ldots, a_n)]\!]^{\mathfrak{A}} = [\![body[a_1/x_1, \ldots, a_n/x_n]]\!]^{\mathfrak{A}}$

For the rule definition **rule** $R(x) = R(x)$ this interpretation yields no value for any $[\![R(a)]\!]^{\mathfrak{A}}$, see [46, Example 1, page 350]. In the following example the update set for $R(x)$ is defined for all $x \leq 10$, with the empty set as update set, and is not defined for any $x > 10$.

rule $R(x) =$ **if** $x < 10$ **then** $R(x+1)$
if $x = 10$ **then skip**
if $x > 10$ **then** $R(x+1)$

Example 1.3.1 (Defining while by a named rule) Named rules allow us to define the *while loop* recursively instead of iteratively:

rule $while_r(cond, R) =$
if *cond* **then**
R **seq** $while_r(cond, R)$

This recursively defined $while_r$ behaves differently from the **while** of the preceding section in that it leads to termination only if the condition *cond* will become eventually false, and not in the case that eventually the update set of R becomes empty. For example the semantics of $while_r(true, \textbf{skip})$ is not defined.

Example 1.3.2 (Starting Java class initialization)
We can define the Java class initialization of Example 1.2.1 also in terms of a recursive named rule, avoiding the local input variable to which the actual parameter is assigned at the beginning.

rule *initialize*$(c) =$
if *initialized*(*superClass*(c)) **then**
createInitFrame(c)
else
createInitFrame(c) **seq** *initialize*(*superClass*(c))

Remark 1.4 Iterated execution of (sub)machines R, started in state $\mathfrak{A}$, unavoidably leads to possibly undefined update sets $[\![R]\!]^{\mathfrak{A}}$. As a consequence

$[\![R]\!]^{\mathfrak{A}} = [\![S]\!]^{\mathfrak{A}}$ denotes that either both sides of the equation are undefined or both are defined and indeed have the same value. In the definitions above we adhered to an algorithmic definition of $[\![R]\!]^{\mathfrak{A}}$, namely by computing its value from the computed values $[\![S]\!]^{\mathfrak{A}}$ of the submachines S of R. In the appendix we give a deduction calculus for proving statements $[\![R]\!]^{\mathfrak{A}} = u$ meaning that $[\![R]\!]^{\mathfrak{A}}$ is defined and has value u.

1.4 Further Concepts

In this section we enrich named rules with a notion of local state, show how parameterized ASMs can be used as machines with return value, and introduce error handling for ASMs which is an abstraction of exception handling as found in modern programming languages.

1.4.1 Local State

Basic ASMs come with a notion of state in which all the dynamic functions are global. The use of only locally visible parts of the state, like variables declared in a class, can naturally be incorporated into named ASMs. It suffices to extend the definition of named rules by allowing some dynamic functions to be declared as local, meaning that each call of the rule works with its own incarnation of local dynamic functions f which are to be initialized upon rule invocation by an initialization rule $Init(f)$. Syntactically we allow definitions of named rules of the following form:

rule $name(x_1, \ldots, x_n) =$
  local $f_1[Init_1]$
  $\vdots$
  local $f_k[Init_k]$
  body

where *body* and $Init_i$ are rules. The formal parameters $x_1, \ldots, x_n$ bind the free occurrences of the corresponding variables in *body* and $Init_i$. The functions $f_1, \ldots, f_k$ are treated as local functions whose scope is the rule where they are introduced. They are not part of the signature of the ASM. $Init_i$ is a rule used for the initialization of f_i. We write **local** $f := t$ for **local** $f[f := t]$.

For the interpretation of a call of a rule with local dynamic functions, the updates to the local functions are collected together with all other function updates made through executing the body. This includes the updates required by the initialization rules. The restriction of the scope of the local functions to the rule definition is obtained by then removing from the update set u, which is available after the execution of the body of the call, the set $Updates(f_1, \ldots, f_k)$ of updates concerning the local functions $f_1, \ldots, f_k$. This leads to the following definition.

Definition 1.4.1 (Name rule with local state) *Let R be a rule declaration with local functions as given above. If the right side of the equation is defined, we set:*

$[\![R(a_1, \ldots, a_n)]\!]^{\mathfrak{A}} =$
  $[\![(\{Init_1, \ldots, Init_k\}$ **seq** $body)[a_1/x_1, \ldots, a_n/x_n]]\!]^{\mathfrak{A}} \setminus Updates(f_1, \ldots, f_k)$

We assume that there are no name clashes for local functions between different incarnations of the same rule (i.e. each rule incarnation has its own set of local dynamic functions).

Example 1.4.1 (Usage of local dynamic functions) The use of local dynamic functions is illustrated by the following rule computing a function f defined by a primitive recursion from functions g and h which are used here as static functions. The rule mimics the corresponding Böhm-Jacopini machine in Proposition 1.2.5.

rule $F(x, y) =$
  local $ival := g(x)$
  local $rec := 0$
  (**while** $(rec < y)$ $\{ival := h(x, rec, ival),\ rec := rec + 1\}$) **seq**
  $out := ival$

1.4.2 ASMs with Return Value

In the preceding example, for outputting purposes the value resulting from the computation is stored in a global dynamic function *out*. This formulation violates good information hiding principles known from Software Engineering. To store the return value of a rule R in a location which is determined by the rule caller and is independent of R, we use the following notation for a new rule:

$$l \leftarrow R(a_1, \ldots, a_n)$$

where R is a named rule with n parameters in which a 0-ary (say reserved) function *result* does occur with the intended role to store the return value. Let **rule** $R(x_1, \ldots, x_n) =$ *body* be the declaration for R, then the semantics of $l \leftarrow R(a_1, \ldots, a_n)$ is defined as the semantics of $R_l(a_1, \ldots, a_n)$ where R_l is defined like R with *result* replaced by l:

rule $R_l(x_1, \ldots, x_n) =$ *body*$[l/result]$

In the definition of the rule R by *body*, the function name *result* plays the role of a placeholder for a location, denoting the interface which is offered for communicating results from any rule execution to its caller. One can apply simultaneously two rules $l \leftarrow R(a_1, \ldots, a_n)$ and $l' \leftarrow R(a'_1, \ldots, a'_n)$ with different return values for l and l'.

Remark 1.5 When using $l \leftarrow R(a_1, \ldots, a_n)$ with a term l of form $f(t_1, \ldots, t_n)$, a good encapsulation discipline will take care that R does not modify the values of t_i, because they contribute to determine the location where the caller expects to find the return value.

Example 1.4.2 (Using return values) Using this notation the above Example 1.4.1 becomes $f(x, y) \leftarrow F(x, y)$ where moreover one can replace the use of the auxiliary static functions g, h by calls to submachines G, H computing them, namely $ival \leftarrow G(x)$ and $ival \leftarrow H(x, rec, ival)$.

Example 1.4.3 A recursive machine computing the factorial function, using multiplication as static function.

rule $Fac(n) =$
  local $x := 1$
  if $n = 1$ **then**
    $result := 1$
  else
    $(x \leftarrow Fac(n-1))$ **seq** $result := n * x$

1.4.3 Error Handling

Programming languages like C++ or Java support exceptions to separate error handling from "normal" execution of code. Producing an inconsistent update set is an abstract form of throwing an exception. We therefore introduce a notion of catching an inconsistent update set and of executing error code.

The semantics of **try** R **catch** $f(t_1, \ldots, t_n)$ S is the update set of R if either this update set is consistent ("normal" execution) or it is inconsistent and the location loc determined by $f(t_1, \ldots, t_n)$ is not updated inconsistently. Otherwise it is the update set of S.

Since the rule enclosed by the **try** block is executed either completely or not at all, there is no need for any **finally** clause to remove trash.

Definition 1.4.2 (Try catch) *Let R and S be rules, f a dynamic function with arguments $t_1, \ldots, t_n$. We define*

$$[\![\textbf{try } R \textbf{ catch } f(t_1, \ldots, t_n)\ S]\!]^{\mathfrak{A}} = \begin{cases} v, & \exists\ v_1 \neq v_2 : (loc, v_1) \in u \wedge (loc, v_2) \in u \\ u, & \textbf{otherwise} \end{cases}$$

where $u = [\![R]\!]^{\mathfrak{A}}$ and $v = [\![S]\!]^{\mathfrak{A}}$ are the update sets of R and S respectively, and loc is the location $f\langle [\![t_1]\!]^{\mathfrak{A}}, \ldots, [\![t_n]\!]^{\mathfrak{A}} \rangle$.

1.5 Related Work

The sequence operator defined by Zamulin in [74] differs from our concept with respect to rules leading to inconsistent update sets for which it is not defined. In case everything is consistent, both definitions compute the same resulting update set. For consistent update sets Zamulin's loop constructor coincides with our while definition in Example 1.2.2.

In Anlauff's XASM [2], calling an ASM is the iteration of a rule until a certain condition holds. [2] provides no formal definition of this concept, but for consistent update sets the XASM implementation seems to behave like our definition of *iterate*.

Named rules with parameters appear in the ASM Workbench [29] and in XASM [2], but with parameters restricted to terms. The ASM Workbench does not allow recursive rules. Recursive ASMs have also been proposed by Gurevich and Spielmann [37]. Their aim was to justify recursive ASMs within distributed ASMs [33]. If R is a rule executed by agent a and has two recursive calls to R,

then a creates two new agents a_1 and a_2 which execute the two corresponding recursive calls. The agent a waits for termination of his slaves a_1 and a_2 and then combines the result of both computations. This is different from our definition where executing a recursive call needs only one step, from the caller's view, so that the justification remains within purely sequential ASMs without invoking concepts from distributed computing. Through our definition the distinction between suspension and reactivation tasks in the iterative implementation of recursion becomes a matter of choosing the black-box or the glass-box view for the recursion. The updates of a recursive call are collected and handed over to the calling machine as a whole to determine the state following in the black-box view the calling state. Only the glass-box view provides a refined inspection of how this collection is computed.

Chapter 2

Component Concept

The last chapter extended the ASM semantics by several structuring principles like parametrized machines and sequential execution of rules. However, for large specifications, these concepts are not sufficient, because it is difficult to divide a specification into smaller independent parts. One key technique to solve such a problem is the usage of components. Hence, we introduce in this chapter a notion of ASM component.

Our component notion is in particular useful for specifying the abstract behavior of digital hardware circuits. See [9] for a general discussion about hardware design with Abstract State Machines.

The idea is to write abstract models of the hardware to be designed using the component concept. These models can be used to validate the design. In the second step, we refine those validated abstract models to their final implementations (the concrete models) and show that the refined models behave in some sense as the abstract models. To do this, we introduce a component-wise verification technique. This proceeding has the advantage, that we can use the abstract system (the composition of the abstract models) for validating system properties instead of doing this in the much more complex concrete system (the composition of the concrete models). The abstract system model is the ground model in the sense of [10].

There are two main languages for hardware design, namely VHDL [39, 24] and Verilog [65]. Since VHDL is the language commonly used in Europe, we will focus on that language. It is a powerful programming language for designing hardware (see [13] for a rigorous ASM description of the semantics of VHDL) but it is generally recognized that VHDL is not suited for high-level descriptions. On the other hand, a hardware designer would not be happy if he has to write the formal piece of a specification (the abstract models) in one language and later he has to encode it in a different language. Hence, the solution is to design a language which is very similar to VHDL, but which can be used for high-level descriptions.

We are now going to describe our specification language for components and their composition and show how our formal composition model can be used to simplify formal verification in large hardware systems. This chapter is divided into four parts. Section 2.1 introduces the formal component model and the specification language to define such components. In Section 2.2 we introduce the composition of components for the formal model and for the specification

Figure 2.1 Graphical notation of a component

language. Based on the composition model we introduce in Section 2.3 a verification technique which allows formal verification for large compositions where the verification can be done component-wise.

2.1 Component

The term component is widely used in software and hardware engineering. In this section we first define our formal component model and then we introduce a syntax to define such components. Our notation of component consists of inputs, outputs, state elements, an output function, and a next state function. The inputs and outputs constitute the component interface which is the only possibility to interact with the component.

Figure 2.1 illustrates the graphical notation used in this chapter for a component with inputs i_1, i_2 and outputs o_1, o_2 in a black-box view.

2.1.1 Formal Definition

The component interface is defined by inputs and outputs. Given the input values, the component computes—depending on the current state—the values for the outputs. For the relation between inputs (outputs) and values we use a notion of *input (output) state* which assigns a value to each input (output). The universe *Val* denotes the universe of values:

Definition 2.1.1 (Input and Output state) *For a set I of inputs and a set O of outputs, the total functions*

$$\mathfrak{i} : I \to Val$$
$$\mathfrak{o} : O \to Val$$

are called input state and output state, respectively. We denote the universe of all total functions from I to Val by the symbol $\mathfrak{I}$. Similarly, we use $\mathfrak{O}$ for the universe of all total functions from O to Val.

The behavior of a component depends on the input state and on the internal state. Similar to the input state we introduce a notion of *internal state* for the relation between state elements and their values:

Definition 2.1.2 (Internal state) *For a set S of state elements, a total function*

$$\mathfrak{s} : S \to Val$$

is called an internal state or simply a state. We denote the universe of all total functions from S to Val by the symbol $\mathfrak{S}$.

Our following component definition is similar to Finite State Machines [21]. We have inputs, outputs, an output function, and a state transition function. In contrast to automata, our state is represented as an assignment from state elements to values similar to dynamic functions in ASMs [33].

Definition 2.1.3 (Component) *A tuple $(I, O, S, next, out)$ is called a component. The sets I and O are the inputs and outputs, S is the set of state elements. The total output function*

$$out\colon \mathfrak{I} \times \mathfrak{S} \to \mathfrak{O}$$

computes the output state, given an input and internal state. Similar to out, the total function

$$next\colon \mathfrak{I} \times \mathfrak{S} \to \mathfrak{S}$$

determines the next (internal) state of the component. We require the sets I, O, and S to be disjoint.

Remark 2.1 The requirement, that I, O, and S have to be disjoint is not a restriction, because feedback wires can be introduced when composing components.

Let $\mathfrak{i}$ be an input state and let $\mathfrak{s}$ be an internal state for a component $(I, O, S, next, out)$. The output function $out(\mathfrak{i}, \mathfrak{s})$ defines the output values; $next(\mathfrak{i}, \mathfrak{s})$ defines the next internal state (one computation step). We now extend these two functions as usual for a sequence of input states:

Definition 2.1.4 (Run) *Let $(I, O, S, next, out)$ be a component. Let $\mathfrak{s}$ be an internal state. For a sequence of input states $\mathfrak{i}s \in_{\geq} n^*\mathfrak{I}$ we define the next state function*

$$next^n\colon \mathfrak{I}^{\geq n} \times \mathfrak{S} \to \mathfrak{S}$$

for $n \geq 0$ as follows:

$$\begin{aligned} next^0(\mathfrak{i}s, \mathfrak{s}) &= \mathfrak{s} \\ next^{n+1}(\mathfrak{i} \cdot \mathfrak{i}s, \mathfrak{s}) &= next^n(\mathfrak{i}s, next(\mathfrak{i}, \mathfrak{s})) \end{aligned}$$

For a sequence of input states $\mathfrak{i}_o \cdot \ldots \cdot \mathfrak{i}_n \cdot \ldots \cdot \mathfrak{i}_{n+k} (k \geq 0)$, we define the output function

$$out^n\colon \mathfrak{I}^{\geq n+1} \times \mathfrak{S} \to \mathfrak{O}$$

for $n \geq 0$ in terms of the next state function $next^n$:

$$out^n(\mathfrak{i}_0 \cdot \ldots \cdot \mathfrak{i}_{n+k}, \mathfrak{s}) = out(\mathfrak{i}_n, next^n(\mathfrak{i}_0 \cdot \ldots \cdot \mathfrak{i}_{n-1}, \mathfrak{s}))$$

Remark 2.2 The definition of this output function out^n for a sequence of input states $\mathfrak{i} \cdot \mathfrak{i}s$ implies (as expected) that

$$out^0(\mathfrak{i} \cdot \mathfrak{i}s, \mathfrak{s}) = out(\mathfrak{i}, \mathfrak{s})$$

The output function *out* computes for given input and internal state an output state. Usually, a subset of input values is sufficient to compute the value for an output o. Hence, we define a notion of dependency set for an output o which contains at least those inputs where the output value of o depends on, i.e., inputs not in this set can not influence the output value. This is formalized as follows:

Definition 2.1.5 (Dependency set) *Let $(I, O, S, next, out)$ be a component. A set*

$$I_o^{dep} \subseteq I$$

is a dependency set for output $o \in O$ if the following condition is true:

$$\forall\, \mathfrak{i}_1, \mathfrak{i}_2 \in \mathfrak{I}, \mathfrak{s} \in \mathfrak{S}:$$
$$(I_o^{dep} \lhd \mathfrak{i}_1 = I_o^{dep} \lhd \mathfrak{i}_2) \Rightarrow out(\mathfrak{i}_1, \mathfrak{s})(o) = out(\mathfrak{i}_2, \mathfrak{s})(o)$$

Obviously, the set I is always a dependency set. However, much more interesting is a dependency set which is as small as possible. We say a dependency set is minimal (with respect to set inclusion), if there is no proper subset which is a dependency set, too:

Definition 2.1.6 (Minimal dependency set) *Let I_o^{dep} be a dependency set for output $o \in O$ in component $(I, O, S, next, out)$. The set I_o^{dep} is called a minimal dependency set if there is no proper subset which is a dependency set for o, too.*

Obviously, for each output o there is a minimal dependency set, because I is a dependency set for o, and one can remove elements as long as the resulting set remains a dependency set for o. The following lemma states that there is exactly one minimal dependency set:

Lemma 2.1.1 (Minimal dependency set: uniqueness)
Let $(I, O, S, next, out)$ be a component. For each $o \in O$, there is exactly one minimal dependency set.

Proof by Contradiction. Assume, there are two different minimal dependency sets I^{dep_1} and I^{dep_2} for an output $o \in O$. We show that then $I^{dep_1} \cap I^{dep_2}$ is a dependency set which implies, that neither I^{dep_1} nor I^{dep_2} can be minimal.
Let $\mathfrak{i}_1$, $\mathfrak{i}_2$ be two input states coinciding on $I^{dep_1} \cap I^{dep_2}$:

$$(I^{dep_1} \cap I^{dep_2}) \lhd \mathfrak{i}_1 = (I^{dep_1} \cap I^{dep_2}) \lhd \mathfrak{i}_2$$

Then there is an input state v coinciding with $\mathfrak{i}_1$ on I^{dep_1} and coinciding with $\mathfrak{i}_2$ on I^{dep_2}:

$$I^{dep_1} \lhd v = I^{dep_1} \lhd \mathfrak{i}_1 \;\wedge\; I^{dep_2} \lhd v = I^{dep_2} \lhd \mathfrak{i}_2$$

This implies that the output function for o for input states $\mathfrak{i}_1$ and $\mathfrak{i}_2$ is equal:

$$out(\mathfrak{i}_1, \mathfrak{s})(o) = out(v, \mathfrak{s})(o) \;=\; out(\mathfrak{i}_2, \mathfrak{s})(o)$$

Hence, $I^{dep_1} \cap I^{dep_2}$ is a dependency set for o. □

Remark 2.3 If there are two disjoint dependency sets for an output o, then the output function for o does not depend on the input values, i.e., the empty set is a dependency set.

Remark 2.4 The above property about minimality of dependency sets is useful for theory. However, in practice it is difficult to compute this minimal set, but it is possible to compute a good approximation by analyzing the dependencies in the definition of the corresponding output function.

2.1.2 Abstraction

For two components C and A we define what it means that A is an IO-abstraction of C. Usually, we use A for the abstract component and C for the concrete component. In the literature ([26, 27, 62], e.g.) the notion *abstraction* is used with many different interpretations. Often, abstraction in the literature implies some mathematical relation between the abstract and concrete model.

Our notion of abstraction defines a relation (mapping) between inputs/outputs of the abstract and concrete component. To distinguish this notion from the term used in the literature, we call this an *IO-Abstraction*:

Definition 2.1.7 (IO-Abstraction) *Let C and A be two components with*

$$C = (I^C, O^C, S^C, next^C, out^C)$$
$$A = (I^A, O^A, S^A, next^A, out^A)$$

Without loss of generality, we require the sets $I^A, O^A, S^A, I^C, O^C, S^C$ to be pairwise disjoint. Let map^I and map^O be partial injective functions (not totally undefined) and let map be their union:

$$map^I : I^A \nrightarrow I^C$$
$$map^O : O^A \nrightarrow O^C$$
$$map = map^I \cup map^O$$

We call A an IO-abstraction of C with respect to the mapping function map.

The mapping function map defines a correspondence between the identifiers in the abstract component A and in the concrete component C. This definition of abstraction is very weak, because there is no need that all inputs or outputs in the abstract model are present in the concrete model and vice versa. We do enforce only, that two different identifiers in the abstract model—if they are mapped at all—are mapped to two different identifiers in the concrete model.

2.1.3 Verification

General techniques for verifying ASMs have been introduced for theorem proving in [59, 31] and for model checking in [72]. Hence, we could use one of these techniques to proof properties about our components. Since we also want to use the abstract models for simulation purpose, we have to translate them into VHDL code. For VHDL, there are already model checkers and we use one of them to proof properties. Hence, we translate our specification language into

VHDL and therefore we do not discuss how to verify ASMs. The translation to VHDL is not described in this thesis.

In Section 2.3 we will introduce a component based verification technique. For this technique, we need a notion of *formula*. Hence, we introduce a very simple property language—boolean combination of timed input/output variables—for the formal component model defined in the previous section. The notations i^t and o^t correspond to the input value of i at time t and output value of o at time t. For instance, the formula

$$i^t \wedge o^{t+1} \vee \neg i^t \wedge \neg o^{t+1}$$

states that the input value of i at time t is equal to the output value of o at time $t+1$. It follows the formal definition:

Definition 2.1.8 (Formula) *The formulas over an input set I and output set O with respect to time period w are inductively defined:*

- *for $i \in I, t \in \mathbb{N}_w$, i^t is a formula (i^t is also called a variable)*
- *for $o \in O, t \in \mathbb{N}_w$, o^t is a formula (o^t is also called a variable)*
- *if F, G are formulas, then also $F \wedge G$, $F \vee G$*
- *if F is a formula, then also $\neg F$*

We use $vars(\varphi)$ to denote the set of variables in φ.

For a formula φ and component C, we define what it means that φ is valid in C (C is a model for φ). In the following definition, C is a model for φ if the formula φ is valid for every possible sequence of input states and initial internal state. Without loss of generality, we assume that each input and output is of basic type *boolean* [1]:

Definition 2.1.9 (Model) *Let φ be a formula over input set I and output set O with respect to time period w. A component $C = (I, O, S, next, out)$ is a model for φ (denoted by $C \models \varphi$) if the following property holds:*

$$\forall\, \mathfrak{is} \in \mathfrak{I}^{\geq w+1}, \mathfrak{s} \in \mathfrak{S} \colon \varphi(\mathfrak{is}, \mathfrak{s}) = true$$

where $\varphi(\mathfrak{is}, \mathfrak{s})$ is defined as follows:

- *for $i \in I, 0 \leq t \leq w$: $i^t(\mathfrak{i}^0 \cdot \ldots \cdot \mathfrak{i}^t \cdot \ldots \cdot \mathfrak{i}^{t+k}, \mathfrak{s}) = \mathfrak{i}^t(i)$*
- *for $o \in O, 0 \leq t \leq w$: $o^t(\mathfrak{is}, \mathfrak{s}) = out^t(\mathfrak{is}, \mathfrak{s})$*
- *$(F \oplus G)(\mathfrak{is}, \mathfrak{s}) = F(\mathfrak{is}, \mathfrak{s}) \oplus G(\mathfrak{is}, \mathfrak{s})$, $\oplus = \wedge, \vee$*
- *$(\neg F)(\mathfrak{is}, \mathfrak{s}) = \neg(F(\mathfrak{is}, \mathfrak{s}))$*

We will use these definitions about formulas in Section 2.3 where we introduce the component based verification technique.

[1] Otherwise one has to extend the formula language in Def. 2.1.8.

Figure 2.2 An example: FlipFlop

```
component FlipFlop
use library std_logic
interface {
  S : in std_logic
  D : in std_logic
  R : in std_logic
  O : out std_logic
}
state { val : std_logic }
```

```
function O is val
rule FlipFlop is {
  if S='1' then
    val := D
  if R='1' then
    val := '0'
}
end component
```

2.1.4 Defining Components

In Section 2.1.1 we defined a formal model for components. For these components we now introduce a specification language inspired by VHDL. This subsection defines this language by introducing the syntax and by defining the translation from the syntax to our formal component model.

Syntax

Syntactically a component consists of a name (the component name), a (possible empty) sequence of used libraries, and a non-empty sequence of component declarations (*compdecl*).

The libraries are used to declare signatures of external functions and external types. A library itself is based on other libraries and a sequence of library declarations (*libdecl*).

component ::= `component` *id*
usedecl*
compdecl$^+$
`end component`

library ::= `library` *id*
usedecl*
libdecl$^+$
`end library`

usedecl ::= `use library` *id*

The grammar for *compdecl* and *libdecl* will be defined below. The symbols $'+'$, $'*'$, and $'?'$ denote one or more, zero or more, and zero or one iteration ($'?'$ will be used later).

Before we introduce the grammar rules, we want to give an impression about the language. Figure 2.2 defines a component called *FlipFlop*. The component has three inputs, namely S, D, R of type *std_logic* which is a commonly used type for bits in VHDL. Additionally, the interface contains an output O. The value for this output is defined by a function having the same name. The behavior of our *FlipFlop* is defined by the main rule *FlipFlop*: Whenever S is equal to $'1'$ we store the input value of D in a state element *val*; if the input R is $'1'$, then we reset *val* to zero. Setting S and R simultaneously to $'1'$ makes no sense, except when D is $'0'$. Note that *val* is a state element and an update to it is visible not until the next step. This means, if R is $'1'$ at time t, then O is $'0'$ at time $t + 1$.

We are now going to introduce the grammar rules in detail and then we define the relation to the formal component model.

A component declaration (*compdecl*) is either a rule declaration, a function declaration, a type declaration, an alias-type declaration, an interface declaration, or a state declaration. Type declarations and alias declarations are also admitted in library declarations. Additionally, a library declaration (*libdecl*) can be a signature declaration for an external function. The declarations are described in detail below. We underline syntactic symbols to distinguish them from the corresponding meta symbols. In particular, this applies for the symbols $'($, $')'$, $'\{'$, $'\}'$ and $','$.

compdecl ::= *ruledecl*
| *fundecl*
| *typedecl*
| *aliasdecl*
| `interface` $\underline{\{}$ *ifacedecl*$^+$ $\underline{\}}$
| `state` $\underline{\{}$ *statedecl*$^+$ $\underline{\}}$

libdecl ::= *sigdecl*
| *typedecl*
| *aliasdecl*

A signature declaration (*sigdecl*) in a library declares the argument types and the return type of a function. In the view of the component such a function is external and is defined by the run-time environment (*and*, *or* on bits, e.g.). A type declaration (*typedecl*) introduces a new type; the syntax for types is described in Appendix B. An alias declaration (*aliasdecl*) introduces an additional name for an already existing type.

sigdecl ::= `function` *id types*? : *type*
typedecl ::= `type` *id* `is` *typedef*
aliasdecl ::= `typealias` *id* `is` *type*

Rule declarations (*ruledecl*) and function declarations (*fundecl*) are very similar. Both have a name and may have a list of formal parameters which can be used in the rule and function body. It is not allowed to define multiple rules or functions with the same name. This applies to all other declarations, too.

ruledecl ::= `rule` *id parameters*? `is` *rule*
fundecl ::= `function` *id parameters*? `is` *term*

parameters ::= $\underline{(}$*parameter*$\underline{(,}$*parameter*$)^*\underline{)}$
parameter ::= *id* (: *type*)?

The interface of a component is defined in terms of inputs and outputs. As in VHDL, the types for inputs and outputs must not be function types [2].

A component reads input values and provides output values. Inputs and outputs are declared by `in` and `out`, respectively.

ifacedecl ::= *id* : (`in` | `out`) *type*

The internal state elements of a component can be defined with state declarations. In terms of ASMs, a state declaration defines a dynamic function and

[2] Otherwise we can not compile the language into VHDL.

Figure 2.3 Interface and state declarations

$$\frac{[type] \mapsto t}{[id : type] \mapsto \{State(id, t, \varepsilon)\}} \quad \text{(statedecl)}$$

$$\frac{[type] \mapsto t, [type_1] \mapsto t_1, \ldots, [type_n] \mapsto t_n}{[id\underline{(}type_1\underline{,}\ldots\underline{,}type_n\underline{)} : type] \mapsto \{State(id, t, \langle t_1, \ldots, t_n\rangle)\}}$$

$$\frac{[decl_1] \mapsto ds_1, \ldots, [decl_n] \mapsto ds_n}{[\texttt{state}\ \underline{\{}decl_1 \ldots decl_n\underline{\}}] \mapsto ds_1 \cup \ldots \cup ds_n} \quad \text{(state)}$$

$$\frac{[type] \mapsto t}{[id : \texttt{in}\ type] \mapsto \{In(id, t)\}} \qquad \frac{[type] \mapsto t}{[id : \texttt{out}\ type] \mapsto \{Out(id, t)\}} \quad \text{(ifacedecl)}$$

$$\frac{[decl_1] \mapsto iface_1, \ldots, [decl_n] \mapsto iface_n}{[\texttt{interface}\ \underline{\{}decl_1 \ldots decl_n\underline{\}}] \mapsto iface_1 \cup \ldots \cup iface_n} \quad \text{(interface)}$$

declares the argument and return type of it.

$$\begin{array}{lll} statedecl & ::= & id\ types?\ :\ type \\ types & ::= & \underline{(}type(\underline{,}type)^*\underline{)} \end{array}$$

The dynamic behavior of a component is defined by ASM rules. Rules are built from the `skip` rule, the function update, and the call rule.

$$\begin{array}{lll} rule & ::= & \texttt{skip} \\ & | & \underline{\{}rule^+\underline{\}} \\ & | & rulecall \\ & | & funterm := term \\ & | & \texttt{if}\ term\ \texttt{then}\ rule\ (\texttt{else}\ rule)? \end{array} \qquad \begin{array}{lll} rulecall & ::= & id\ terms? \\ funterm & ::= & id\ terms? \\ terms & ::= & \underline{(}term(\underline{,}term)^*\underline{)} \end{array}$$

The `skip` rule does nothing, it is like a semicolon in C++. The function update assigns a new value to the function symbol on the lefthand side for the given arguments. A call rule is similar to a function call: For the identifier in the call rule (leftmost identifier), there must be a rule declaration statement and the length and types of the formal arguments from the rule declaration statement must match the parameters in the call rule. A sequence of rules can be grouped to one rule by curly braces. The *else* part in the *if then* rule is optional. The semantics of rules is described in detail in the next section.

The syntax and semantics for terms and types is listed in Appendix B where we also list static constraints about the definitions introduced above. However, these constraints are similar as in other programming languages; for instance, every identifier must be defined, the program must be well typed, etc.

Interpretation

The remaining paragraphs in this subsection define the interpretation of the previous syntax definitions. More precisely, we introduce derivation rules to transform a syntactical component into the mathematical model of subsection

Figure 2.4 Type and signature declarations

$$\frac{[typedef] \mapsto def}{[\texttt{type}\ id\ \texttt{is}\ typedef] \mapsto \{Typedef(id, def)\}} \quad \text{(typedecl)}$$

$$\frac{[type] \mapsto t}{[\texttt{typealias}\ id\ \texttt{is}\ type] \mapsto \{Alias(id, t)\}} \quad \text{(aliasdecl)}$$

$$\frac{[type] \mapsto t}{[\texttt{function}\ id : type] \mapsto \{Sig(id, t, \varepsilon)\}} \quad \text{(sigdecl)}$$

$$\frac{[type] \mapsto t, [type_1] \mapsto t_1, \ldots, [type_n] \mapsto t_n}{[\texttt{function}\ id\_(type_1, \ldots, type_n)\_ : type] \mapsto \{Sig(id, t, \langle t_1, \ldots, t_n \rangle)\}}$$

2.1.1. The derivation rules are shown in Fig. 2.3, 2.4, 2.5. We use the notation

$$[str] \mapsto def$$

to denote that a syntactical string *str* is transformed into a set of abstract declarations *def*.

We gather all abstract declarations in an environment *env*. This is expressed by the derivation rule below. It states that if a syntactical string *decl* can be transformed (by Fig. 2.3 2.4, 2.5) to a set *ds*, then each element in this set is also contained in *env*.

$$\frac{[decl] \mapsto ds, d \in ds}{d \in env}$$

In our specification language we do not commit to any special ordering of the declarations. However, the environment *env* can only be computed, if all declarations can be sorted topologically (no cyclic definitions), because *env* is also used in the interpretation of types and terms (see Appendix B). Therefore, any kind of recursion for function and rule definitions is prohibited.

In the following paragraphs, we consider the interpretation of the introduced syntax parts such that we can eventually define the interpretation for a syntactical component definition.

Inputs, Outputs, States. Figure 2.3 defines the interpretation for inputs, outputs, and state elements. They are translated to the abstract declarations $In(\ldots), Out(\ldots), State(\ldots)$. Hence, the sets I, O, and S (the inputs, the outputs, and the state elements) can be defined by looking into the environment *env* ('_' matches anything):

$$\frac{In(id, \_) \in env}{id \in I} \qquad \frac{Out(id, \_) \in env}{id \in O} \qquad \frac{State(id, \_, \_) \in env}{id \in S}$$

Function and rules. Figure 2.5 shows the interpretation for function and rule declarations. The interpretation of rules (update sets) and functions (term values) depends on the input and internal state. This dependence is not evident

Figure 2.5 Function and rule declarations

$$\frac{[\mathit{rule}] \mapsto r}{[\texttt{rule}\ \mathit{id}\ \texttt{is}\ \mathit{rule}] \mapsto \{\mathit{Rule}(\mathit{id}, \varepsilon, r)\}} \quad \text{(ruledecl)}$$

$$\frac{[\mathit{rule}] \mapsto r, [p_1] \mapsto \mathit{id}_1, \ldots, [p_n] \mapsto \mathit{id}_n}{[\texttt{rule}\ \mathit{id}\underline{(}p_1\underline{,}\ldots\underline{,}p_n\underline{)}\ \texttt{is}\ \mathit{rule}] \mapsto \{\mathit{Rule}(\mathit{id}, \mathit{id}_1 \cdot \ldots \cdot \mathit{id}_n, r)\}}$$

$$\frac{[\mathit{term}] \mapsto f}{[\texttt{function}\ \mathit{id}\ \texttt{is}\ \mathit{term}] \mapsto \{\mathit{Fun}(\mathit{id}, \varepsilon, f)\}} \quad \text{(fundecl)}$$

$$\frac{[\mathit{term}] \mapsto f, [p_1] \mapsto \mathit{id}_1, \ldots [p_n] \mapsto \mathit{id}_n}{[\texttt{function}\ \mathit{id}\underline{(}p_1\underline{,}\ldots\underline{,}p_n\underline{)}\ \texttt{is}\ \mathit{term}] \mapsto \{\mathit{Fun}(\mathit{id}, \langle \mathit{id}_1, \ldots, \mathit{id}_n \rangle, f)\}}$$

from the figure. In the figure, we have the conditions $[\mathit{rule}] \mapsto r$ and $[\mathit{term}] \mapsto f$ and the question is what are r and f. Let us first consider r which is defined by the following rule:

$$\frac{\forall\, \mathfrak{i}, \mathfrak{s}, \zeta\colon\ [\![\mathit{rule}]\!]_\zeta^{\mathfrak{i},\mathfrak{s}} = r(\mathfrak{i}, \mathfrak{s}, \zeta)}{[\mathit{rule}] \mapsto r}$$

If $[\mathit{rule}] \mapsto r$ holds, then r is a function from input state, internal state, and local variable environment to an update set. The notation $[\![\mathit{rule}]\!]_\zeta^{\mathfrak{i},\mathfrak{s}} = r(\mathfrak{i}, \mathfrak{s}, \zeta)$ means that *rule* is interpreted as the update set $r(\mathfrak{i}, \mathfrak{s}, \zeta)$ with respect to input state $\mathfrak{i}$, internal state $\mathfrak{s}$, and local variable environment ζ.

The definition of f in $[\mathit{term}] \mapsto f$ is similar to definition of r in $[\mathit{rule}] \mapsto r$ described above.

$$\frac{\forall\, \mathfrak{i}, \mathfrak{s}, \zeta\colon\ [\![\mathit{term}]\!]_\zeta^{\mathfrak{i},\mathfrak{s}} = f(\mathfrak{i}, \mathfrak{s}, \zeta)}{[\mathit{term}] \mapsto f}$$

The notation $[\![\mathit{term}]\!]_\zeta^{\mathfrak{i},\mathfrak{s}}$ interprets *term* with respect to input state $\mathfrak{i}$, internal state $\mathfrak{s}$, and local variable environment ζ. For more information, see Appendix B where term interpretation is defined in detail for our specification language.

Output function. The value of an output is defined in our specification language by a nullary function having the same name.[3] Therefore, for any nullary function definition, where the function name *id* is in the set of outputs, we define the function $\mathit{out}_{\mathit{id}}$ which computes for given input state and internal state the value for the output. We use $\perp$ for the empty local variable environment.

$$\frac{\mathit{Fun}(\mathit{id}, \varepsilon, f) \in \mathit{env}, \mathit{id} \in O}{\forall\, \mathfrak{i} \in \mathfrak{I}, \mathfrak{s} \in \mathfrak{S}\colon\ \mathit{out}_{\mathit{id}}(\mathfrak{i}, \mathfrak{s}) = f(\mathfrak{i}, \mathfrak{s}, \perp)}$$

Usage of libraries. A component may include library declarations of a library *lib* by using the *usedecl* syntax

`use library` *lib*

Figure 2.6 Rules

$$
\begin{aligned}
&[\![\texttt{skip}]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta} && = \varnothing \\
&[\![\{rule_1 \ldots rule_n\}]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta} && = [\![rule_1]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta} \cup \cdots \cup [\![rule_n]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta} \\
&[\![f := t]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta} && = \{(f, \varepsilon, [\![t]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta})\} \\
&[\![f(t_1, \ldots, t_n) := t]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta} && = \{(f, \langle [\![t_1]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta} \cdot \ldots \cdot [\![t_n]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta} \rangle, [\![t]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta})\} \\
&[\![r(t_1, \ldots, t_n)]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta} && = [\![r]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta}([\![t_1]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta}, \ldots, [\![t_n]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta}) \\
&[\![\texttt{if } t \texttt{ then } rule_1 \texttt{ else } rule_2]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta} && = \begin{cases} [\![rule_1]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta}, & [\![t]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta} = true \\ [\![rule_2]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta}, & [\![t]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta} = false \end{cases} \\
&[\![r]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta} = f \textbf{ where } f(p_1, \ldots, p_n) = [\![body]\!]^{\mathfrak{i},\mathfrak{s}}_{id_1 \to p_1, \ldots, id_n \to p_n} \text{ and} \\
&\qquad\qquad Rule(r, id_1 \cdot \ldots \cdot id_n, body) \in env
\end{aligned}
$$

For the interpretation of such a *usedecl* statement we can include all library declarations into the environment *env*. This is done by the rules below where we assume, that the function *content* returns (by a database lookup, e.g.) the syntactical library content for the specified library name:

$$\frac{content([lib]) \mapsto ds}{[\texttt{use library } lib] \mapsto ds}$$

$$\frac{[uses] \mapsto ds, [decl_1] \mapsto ds_1, \ldots, [decl_n] \mapsto ds_n}{[\texttt{library } id \; uses \; decl_1 \ldots decl_n \texttt{ end library}] \mapsto ds \cup ds_1 \cup \cdots \cup ds_n}$$

Rule interpretation. Figure 2.6 defines the interpretation for rules for given input state, internal state, and local variable environment. The notation used is similar to the definition of ASMs in [64]. Rules are built up from `skip` and function updates. The update set of `skip` is the empty set and the update set of a function update is the singleton set containing the update itself. The identifier f in a function update must be an element of S and the number and types of arguments must match the state element definition of f.

The update set of a parallel execution is the union of the corresponding update sets and the update set of a call rule is the update set of the rule body (defined by the environment *env*) with instantiated parameters in the local variable environment.

Next state function. The next state function computes for given internal state (and input state) an internal state which is obtained by executing one step of the component. In our language, we execute the nullary rule which has the same name as the component and we call this the *main* rule of the component.[4]

Executing a rule with respect to an input and internal state yields an update set (see Figure 2.6). For a consistent update set u for the main rule with respect to given input state $\mathfrak{i}$ and internal state $\mathfrak{s}$, we define the next internal state

[3] The syntactic constraints in Appendix B ensure that there is a nullary function definition for each output.

[4] The syntactic constraints in Appendix B ensure that there is a nullary rule with the name of the component.

$next(\mathfrak{i}, \mathfrak{s}) = \mathfrak{s}_{new}$ which we obtain by applying the update set u to $\mathfrak{s}$ as follows: The internal state $\mathfrak{s}_{new}$ coincides with $\mathfrak{s}$ except for the following condition:

$$\text{For } (s\langle t_1 \cdot \ldots \cdot t_n \rangle, t) \in u \Rightarrow \mathfrak{s}_{new}(s)(t_1, \ldots, t_n) = t$$

Note that n denotes the arity of the state element s.

In case the update set u is inconsistent, the next internal state coincides with the current state: $next(\mathfrak{i}, \mathfrak{s}) = \mathfrak{s}$.

Component interpretation. Based on the previous definitions, we can define the interpretation for the whole syntactic component definition:

Definition 2.1.10 (Interpretation for components) *Let*

$$C = [\texttt{component } id \text{ uses } decl_1 \ldots decl_n \texttt{ end component}]$$

be a syntactical component definition. Let env be the environment defined by the previous derivation rules for the declaration statements $decl_1, \ldots, decl_n$ *and the uses statements. The interpretation for* C *is* $(I, O, S, next, out)$ *where* I*,* O*,* S*,* out_o*, and* $next$ *are defined by the previous definitions for* C *and* out *is defined in terms of* out_o*:*

$$\forall\ o \in O, \mathfrak{i} \in \mathfrak{I}, \mathfrak{s} \in \mathfrak{S}\colon\ out(\mathfrak{i}, \mathfrak{s})(o) = out_o(\mathfrak{i}, \mathfrak{s})$$

The above definition transforms the syntactically defined component to our formal component framework as stated in the next lemma. This includes in particular that *out* and *next* must be total functions.

Lemma 2.1.2 *Let* C *be a syntactical component definition. The interpretation* $(I, O, S, next, out)$ *defined above is a component.*

Proof. The syntactic constraints (see Appendix B) ensure that there is an output function for each output. Therefore, the defined *out* function is total. The next state function *next* is total by construction. □

2.2 Composition of Components

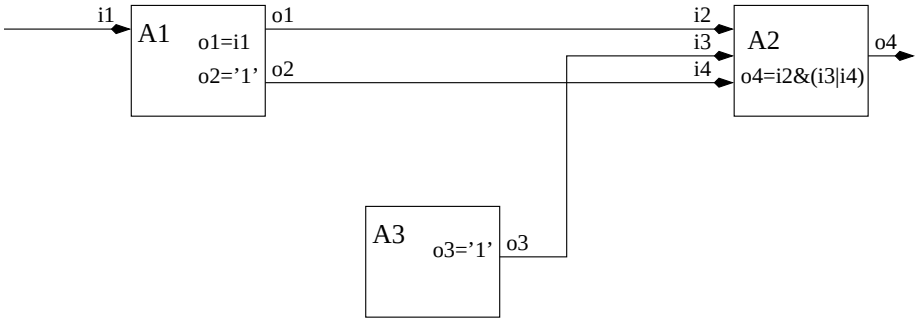
In this section we build new components based on already existing components. We first introduce the formal model for composing components and then show how component composition can be defined using our specification language.

2.2.1 Formal Definition

For a set of components we could build a new component by putting one interface (containing all inputs and outputs) around the basic components. The behavior of this new component would be the union of the single behaviors. However, we are more interested in building a new component where the behavior is defined by the behavior and interaction of the single components. Therefore, we connect inputs and outputs of the single components.

Figure 2.7 shows an example for a composition of the components A_1, A_2, A_3. In that example, i_2 is connected with o_1, i_3 with o_3, and i_4 with o_2.

Figure 2.7 Example for composition of components



To define the connections among inputs and outputs of the single components, we introduce a notion of *connection function*. For the example in Fig. 2.7, the connection function cf is defined as follows:

$$cf(i) = \begin{cases} o_1, & i = i_2 \\ o_3, & i = i_3 \\ o_2, & i = i_4 \end{cases}$$

If $cf(i) = id$, then the input value for i is determined by id. It follows the formal definition of a connection function:

Definition 2.2.1 (Connection function)
Let $C_1, \ldots, C_n$ be components. Let $C_i = (I_i, O_i, S_i, next_i, out_i)$. Without loss of generality, we require the sets I_i, O_i, S_i to be pairwise disjoint. A partial function

$$cf : (I_1 \cup \cdots \cup I_n) \nrightarrow (I_1 \cup \cdots \cup I_n \cup O_1 \cup \cdots \cup O_n)$$

connecting inputs and outputs or inputs and inputs is called a connection function for the components $C_1, \ldots, C_n$, if cf viewed as a relation is acyclic.

The definition of *connection function* raises two questions: (i) why do we allow connecting input with input and (ii) why do we restrict the set of valid connection functions? The answer to the first question is simple, because it's useful that in the composition several inputs get the same input value and we will use this feature later.

The restriction of the set of valid connection functions ensures that either an input is not connected, or it is connected only with an output, or with an input which is transitively not connected to the original input. This implies that for connected inputs we can always determine the source which determines the input value.

We introduce cf^* to denote the connection function obtained from cf where each input element is mapped to the identifier which eventually determines the

input value (cf^* can be viewed as the reflexive transitive closure of cf):

$$cf^*(i) = \begin{cases} cf^*(cf(i)), & cf(i) \in \mathrm{dom}(cf) \\ cf(f) & i \in \mathrm{dom}(cf) \\ i, & otherwise \end{cases}$$

The requirement, that cf must be acyclic does not prohibit cyclic definitions of outputs. In fact, there are situations where the behavior of a composition is not well-defined. For example, consider the composition in Fig. 2.7 with an additional connection from o_4 to i_1. Then we would have the following definitions:

$$o_1 = i_1, i_1 = o_4, o_4 = i_2 \& (i_3 \mid i_4), i_2 = o_1$$

If we unfold the definition of o_1, then we see that o_1 is defined recursively. We want to prohibit such compositions. Therefore, we analyze the resulting dependencies in the composed model by the following two definitions of *dependency function* and *dependency relation*.

The definition of *dependency function* below combines the dependency sets of the single outputs into one function. We will use it in the definition of a *dependency relation*.

Definition 2.2.2 (Dependency function)
Let $C_1, \ldots, C_n$ *be components with*

$$\begin{aligned} C_i &= (I_i, O_i, S_i, next_i, out_i) \\ I &= I_1 \cup \cdots \cup I_n \\ O &= O_1 \cup \cdots \cup O_n \end{aligned}$$

A function

$$I^{dep} \colon O \to \mathbb{P}(I)$$

is a called dependency function with respect to $C_1, \ldots, C_n$ *if for* $o \in O_i$, $I^{dep}(o)$ *is a dependency set for each* o *in component* C_i.

A dependency set $I^{dep}(o)$ contains the inputs, the output o at least depends on. If o is element of O_i, then $I^{dep}(o) \subseteq I_i$, i.e., the dependency set contains only inputs belonging to the same component. However, if an output o depends on an input i and i is connected with id, then o also depends on id.

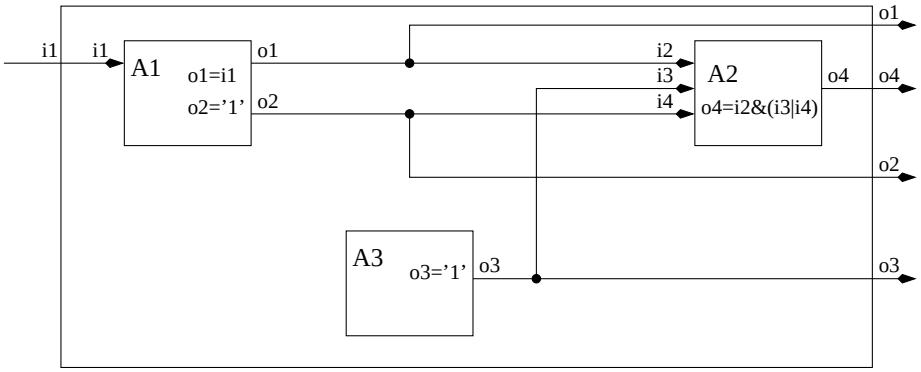
In the following definition of *dependency relation* we extend the dependencies in the dependency set with respect to the connections among the components. A pair (id_1, id_2) in the dependency relation means that the value of id_1 depends on the value of id_2:

Definition 2.2.3 (Dependency relation)
Let $C_1, \ldots, C_n$ *be components with*

$$C_i = (I_i, O_i, S_i, next_i, out_i)$$

Let cf *be a connection function for* $C_1, \ldots, C_n$. *Let* I^{dep} *be a dependency function with respect to* $C_1, \ldots, C_n$. *A smallest set* dep *fulfilling the following two*

Figure 2.8 Composed component



properties is called a dependency relation with respect to connection function cf and dependency function I^{dep}*:*

$\forall\ i \in \mathrm{dom}(cf)\colon\ (i, cf(i)) \in dep$
$\forall\ o \in O_1 \cup \ldots \cup O_n, i \in I^{dep}(o)\colon\ (o, i) \in dep$

If the dependency relation contains no cycle, then no output in the composed model could depend on its own value (instantaneously). This is a necessary condition to define the interpretation for the composition of the components.

The composition we define in the next theorem for components $C_1, \ldots, C_n$ and connection function cf is a component where

- the set of inputs is the union of the inputs of the single components without those inputs which are connected to other identifiers, i.e. which are in the domain of cf.
- the set of outputs is the union of the outputs of the single components. The same applies to the state elements.
- the output function for an output o is the output function of the corresponding component where o is defined. Since in the composed model we have an input state for the inputs of the composed model, we define the input state for the single components by using the input state for the composition, the connection function, and the output functions of the single components. In particular, if an input is connected to an output, then the input value is the result of the output function of the output it is connected to.
- the next state function for a state element s is the result of the next state function of the corresponding component where s is defined.

Figure 2.8 illustrates the component obtained from the composition shown in Fig. 2.7 according to the previous description of composition. In the figure, only i_1 is an input of the composed component, because all other inputs of the single components are connected to outputs. On the other hand, all outputs of

the single components are also outputs of the composition regardless whether they are connected or not.

The following theorem formalizes this composition principle in terms of the given single components according to the previous description and states that the composition is a component:

Theorem 2.2.1 (Composition of components)
Let $C_1, \ldots, C_n$ be components with

$$C_i = (I_i, O_i, S_i, next_i, out_i)$$

Let cf be a connection function for $C_1, \ldots, C_n$. The tuple $(I, O, S, next, out)$ with I, O, S, out, $next$ defined below is a component if there is a dependency function I^{dep} with respect to $C_1, \ldots, C_n$, such that the dependency relation dep with respect to cf and I^{dep} is acyclic:

$$\begin{aligned} I &= (I_1 \cup \cdots \cup I_n) \setminus \mathrm{dom}(cf) \\ O &= O_1 \cup \cdots \cup O_n \\ S &= S_1 \cup \cdots \cup S_n \end{aligned}$$

$$\frac{o \in O_i}{out(\mathfrak{i}, \mathfrak{s})(o) = out_i(\mathfrak{i}_i, S_i \lhd \mathfrak{s})(o)} \qquad \frac{s \in S_i}{next(\mathfrak{i}, \mathfrak{s})(s) = next_i(\mathfrak{i}_i, S_i \lhd \mathfrak{s})(s)}$$

$$\frac{cf^*(id) \in I, id \in I_i}{\mathfrak{i}_i(id) = \mathfrak{i}(cf^*(id))} \qquad \frac{cf^*(id) \in O_j, id \in I_i}{\mathfrak{i}_i(id) = out_j(\mathfrak{i}_j, S_j \lhd \mathfrak{s})(cf^*(id))}$$

Proof. We have to show that the output function *out* and the next state function *next* are well-defined. The crucial point is the definition of the input state $\mathfrak{i}_i$ for a component C_i depending on the global input state $\mathfrak{i}$. For the input state $\mathfrak{i}_i$ we have to show that it is well defined.

Let us first consider the definition $\mathfrak{i}_i(id) = out_j(\mathfrak{i}_j, S_j \lhd \mathfrak{s})(cf^*(id))$ in the deduction rule at right bottom. Since $I^{dep}(cf^*(id))$ is a dependency set for output $cf^*(id)$, the following property is satisfied for each input state v:

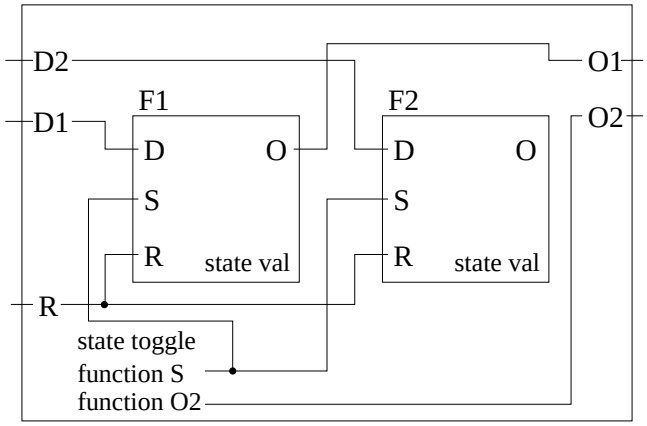
$$\begin{aligned} &I^{dep}(cf^*(id)) \lhd v = I^{dep}(cf^*(id)) \lhd \mathfrak{i}_i \Rightarrow \\ &\quad out_j(\mathfrak{i}_j, S_j \lhd \mathfrak{s})(cf^*(id)) = out_j(v, S_j \lhd \mathfrak{s})(cf^*(id)) \end{aligned}$$

This implies that we only need the input values $\mathfrak{i}_i(n)$ for $n \in I^{dep}(cf^*(id))$. The other input values can be chosen arbitrarily. Since the dependency relation *dep* is acyclic, the definition of $\mathfrak{i}_i(id)$ is acyclic, too. □

Remark 2.5 A composition C' in Theorem 2.2.1 where the set of outputs is a subset of O is a component, too. In such a case we have to restrict the domain of the output function *out* to the new domain.

Given the dependency sets for the single components, we can compute a dependency set for each output in the composition. An input i of the composition is in the dependency set for an output o, if the pair (o, i) is in the transitive closure of the corresponding dependency relation:

Figure 2.9 Graphical composition principle



Lemma 2.2.1 (Dependency set of composition)
Let $C_1, \ldots, C_n$ be components. Let cf be a connection function for $C_1, \ldots, C_n$, let I^{dep} be a dependency function with respect to $C_1, \ldots, C_n$, let dep be a dependency relation with respect to cf and I^{dep}, and let $C = (I, O, S, next, out)$ be the composition of $C_1, \ldots, C_n$ according to Theorem 2.2.1. Then the set I_o^{dep} defined below is a dependency set for output $o \in O$.

$$\forall\ o \in O, i \in I\colon\ i \in I_o^{dep} \Leftrightarrow (o, i) \in dep^+$$

where dep^+ is the transitive closure of dep.

Proof. We have to show that I_o^{dep} is a dependency set for output o. Given the dependency relation dep we first determine the dependency set $I_{j,o}^{dep}$ for an output o with respect to the corresponding component C_j where the output o is defined ($I_{j,o}^{dep}$ is a dependency set according to Def. 2.2.3).

$$\forall\, o \in O_j, i \in I_j\colon i \in I_{j,o}^{dep} \Leftrightarrow (o, i) \in dep$$

Consider now two input states u and v for the composition C. We have to show the following property:

$$I_o^{dep} \lhd u = I_o^{dep} \lhd v \ \Rightarrow\ out(u, \mathfrak{s})(o) = out(v, \mathfrak{s})(o)$$

Let output o be defined in component C_j and let u_j and v_j be the input states resulting from u and v according to Theorem 2.2.1 for component C_j. We now prove the following property which implies that the output functions for the input states u and v are equal.

$$I_o^{dep} \lhd u = I_o^{dep} \lhd v \ \Rightarrow\ I_{j,o}^{dep} \lhd u_j = I_{j,o}^{dep} \lhd v_j$$

For $i \in I_{j,o}^{dep}$ we have to distinguish three cases:

1. $i \in I$
   This implies that $i \in I_o^{dep}$, $u(i) = u_j(i)$, $v(i) = v_j(i)$, and therefore $u_j(i) = v_j(i)$.

2. $cf^*(i) \in I$
   Similar to the first case.

3. $cf^*(i) \in O$
   Now we prove that $I_{cf^*(i)}^{dep}$ is a dependency set for output $cf^*(i)$. This proof terminates, because in that proof there is no need to prove that I_o^{dep} is a dependency set for o, because the dependency relation dep in Theorem 2.2.1 is acyclic.

   By construction of dep, the following property holds:

   $$i \in I_o^{dep} \wedge cf^*(i) \in O \Rightarrow I_o^{dep} \supseteq I_{cf^*(i)}^{dep}$$

   With this property we can conclude our proof:

   $$\begin{array}{ll} & I_o^{dep} \lhd u = I_o^{dep} \lhd v \\ \Rightarrow & I_{cf^*(i)}^{dep} \lhd u = I_{cf^*(i)}^{dep} \lhd v \\ \Rightarrow & out(u, \mathfrak{s})(cf^*(i)) = out(v, \mathfrak{s})(cf^*(i)) \\ \Rightarrow & u_j(i) = v_j(i) \end{array}$$

□

Remark 2.6 The dependency set I_o^{dep} constructed in Lemma 2.2.1 for output o in the composition C is not necessarily minimal, even if the corresponding dependency sets for the single components are minimal.

2.2.2 Defining Composition

This subsection defines a syntax for the specification language which allows to define components by composition.

For the formal model, we defined composition by connecting single components as was illustrated by Fig. 2.8. Syntactically, we allow a more general composition technique, namely inclusion of components (similar to VHDL, e.g. in [6]). Figure 2.9 illustrates this composition technique. The figure shows a component M with inputs $D1, D2, R$, outputs $O1, O2$, state variable *toggle*, and functions $S, O2$. The component M includes the components $F1$ and $F2$ and defines connections among the different interfaces.

We first introduce the syntax to define such compositions and then we show how to reduce it to the formal composition model, namely to connect single components.

Syntax

Components can be included similarly to libraries. There is no need to include a library twice, but two instances of one component may differ from each other due to their state. Therefore, an inclusion statement for components consists of a component name (of the component to be included) and an alias name. The

Figure 2.10 Composition principle

```
component M
use library std_logic
use component FlipFlop as F1
use component FlipFlop as F2
interface {
  D1 : in  std_logic
  D2 : in  std_logic
  R  : in  std_logic
  O1 : out std_logic
  O2 : out std_logic
}
state { toggle : std_logic }
function O2 is F1:O and F2:O
```

```
connect {
  F1:D  = D1
  F2:D  = D2
  O1    = F1:O
  F1:R  = R
  F2:R  = R
  F1:S  = S
  F2:S  = S
}
rule M is {
  toggle := not(toggle)
}
function S is toggle and not(R)
end component
```

same component may be included several times with different alias names. We extend the *usedecl* statement for components.

$$usedecl ::= \texttt{use library}\ id \mid \texttt{use component}\ id\ \texttt{as}\ id$$

Figure 2.10 illustrates the composition in our specification language (Fig. 2.9 is the graphical representation of Fig. 2.10). In the example, the *FlipFlop* component is included as $F1$ and $F2$. For connecting identifiers we provide a connection section as can be seen in the figure. Additionally, the notation $alias : id$ allows to reference outputs of included components (the definition of function $O2$, e.g.); *alias* is a component C included with alias name *alias*, and id is an output in C.

We are now going to explain the syntactical constructs in detail. For connecting inputs and outputs we introduce the connection statement where we can define connections between the including and the included component and among included components. Therefore, we extend the *compdecl* statement with a connection section.

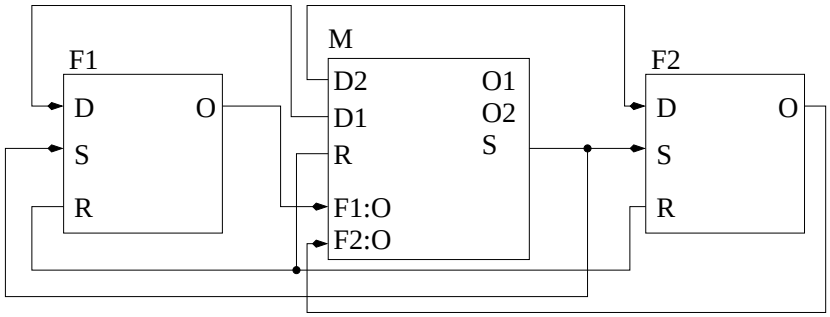
$$\begin{aligned} compdecl ::= {} & ruledecl \\ \mid {} & fundecl \\ \mid {} & typedecl \\ \mid {} & aliasdecl \\ \mid {} & \texttt{interface}\ \underline{\{}\ ifacedecl^{+}\ \underline{\}} \\ \mid {} & \texttt{state}\ \underline{\{}\ statedecl^{+}\ \underline{\}} \\ \mid {} & \texttt{connect}\ \underline{\{}\ conndecl^{+}\ \underline{\}} \end{aligned}$$

$$\begin{aligned} conndecl ::= {} & id = id : id \\ \mid {} & id : id = (id : id \mid id) \end{aligned}$$

Let C be an including component and let $C_1, \ldots, C_n$ be included components. There are four possibilities to connect among these components.

- Output id_1 of C with output id_2 of C_i $\quad (id_1 = C_i : id_2)$
- Input id_1 of C_i with input id_2 of C $\quad (C_i : id_1 = id_2)$
- Input id_1 of C_i with nullary function id_2 in C $\quad (C_i : id_1 = id_2)$
- Input id_1 of C_i with output id_2 of C_j $\quad (C_i : id_1 = C_j : id_2)$

Figure 2.11 Reduction to connecting single components



The value for the identifier on the lefthand side in a connection is determined by the identifier on the righthand side.

Interpretation

The following paragraphs define the interpretation of the introduced syntax fro composing components. Instead of defining a new framework for this composition technique, we reduce it to the composition of connecting single components.

Figure 2.11 illustrates the reduction principle for the composition shown in Fig. 2.9. We lift the included components $F1$ and $F2$ outside the including component M.Now we have three components connected to each other and we can apply the composition Theorem 2.2.1.

We are now going to describe this reduction in detail and we start with the interpretation for the component inclusion statement:

$$\overline{[\texttt{use component}\ c\ \texttt{as}\ id] \mapsto \{Use(c, id)\}}$$

For $Use(c, id) \in env$ and $C = (I, O, S, next, out)$ the interpretation of c, we introduce the notation $C_{id} = (I_{id}, O_{id}, S_{id}, next_{id}, out_{id})$ for a corresponding copy of C.

The notation $\overline{c:id}$ transforms the syntactical string $c:id$ into the semantical identifier id_c in component c. Note that we assume that all inputs, outputs, and state elements in different components are disjoint.

In the interpretation of the connection statement defined in Fig. 2.12 we have to distinguish the different connections as introduced in the syntax part. We use *OutOut* when connecting output with output, *InIn* for input with input, *InOut* for input with output, and *InFun* for input with function.

We can now define the interpretation of composition based on Theorem 2.2.1: If a component C uses the component instantiations $C_1, \ldots, C_n$, then we define a component C' and compose the components $C', C_1, \ldots, C_n$ according to Theorem 2.2.1. The component C' is obtained from C by eliminating the component inclusion statements and the connection section. Furthermore, referenced outputs of C_i in C (using the notation $C_i : id$ in terms) are treated as primary inputs of C', and nullary functions in C which appear as the righthand-side in the connection section of C are added as outputs in C'.

Figure 2.12 Connection declarations

$$\frac{id_1 \in O, \overline{c:id_2} \in O_c}{[id_1 = c : id_2] \mapsto \{OutOut(id_1, \overline{c:id_2})\}} \qquad \frac{id_2 \in I, \overline{c:id_1} \in I_c}{[c : id_1 = id_2] \mapsto \{InIn(id_2, \overline{c:id_1})\}}$$

$$\frac{\overline{c:id_1} \in I_c, Fun(id_2, \varepsilon, f) \in env}{[c : id_1 = id_2] \mapsto \{InFun(\overline{c:id_1}, id_2)\}}$$

$$\frac{\overline{c_1:id_1} \in I_{c_1}, \overline{c_2:id_2} \in O_{c_2}}{[c_1 : id_1 = c_2 : id_2] \mapsto \{InOut(\overline{c_1:id_1}, \overline{c_2:id_2})\}}$$

For the example in Fig. 2.11 this means that we modify component M to M' where we introduce two new inputs $F1:O$ and $F2:O$ and a new output S. We connect O in component $F1$ with the new input $F1:O$ in M' and O in $F2$ with the new input $F2:O$ in M', and S with $F1:S$ and $F2:S$. We now define this composition formally:

Definition 2.2.4 (Interpretation for composition) *Let*

$$C = [\texttt{component}\ id\ \textit{uses}\ decl_1 \ldots decl_n\ \texttt{end component}]$$

be a syntactically defined component. The interpretation of C is

$$(I, O, S, next, out)$$

where $I, S, out, next$ are defined as in Theorem 2.2.1 for the components

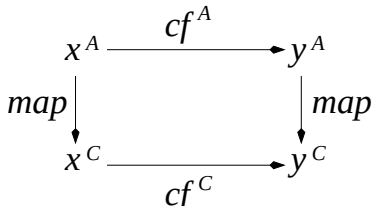
$$C'_0, C_1, \ldots, C_n$$

and connection function cf defined below. The set of outputs O of the new component is the set O_0 (the same outputs as in the definition of the interface in C). C_0 is the interpretation of C without the component inclusion and connection statements and where terms like $c:id$ are treated as inputs of C.

$$\{C_1, \ldots, C_n\} = \{C_{id} \mid Use(c, id) \in env\}$$
$$I' = \{c:id \mid c:id \in terms(C) \vee \exists\ o : OutOut(o, \overline{c:id}) \in env\}$$
$$O' = \{id_2 \mid InFun(id_1, id_2) \in env\}$$
$$C'_0 = (I_0 \cup I', O_0 \cup O', S_0, next_0, out')$$
$$out'(\mathfrak{i}, \mathfrak{s})(o) = \begin{cases} f(\mathfrak{i}, \mathfrak{s}, \bot), & Fun(o, \varepsilon, f) \in env \\ \mathfrak{i}(c:id), & OutOut(o, \overline{c:id}) \in env \end{cases}$$
$$cf(i) = \begin{cases} o, & InOut(i, o) \in env \vee InFun(i, o) \in env \\ i', & InIn(i, i') \in env \\ \overline{c:id}, & i = c:id \wedge (c:id \in terms(C) \vee \\ & \qquad \exists\ o : OutOut(o, \overline{c:id}) \in env) \end{cases}$$

We assume that the function $terms(C)$ computes the set of all basic expressions in the definitions of the syntactical component C.

Figure 2.13 Abstract connection function



2.3 Component based Verification

Automatic property verification (model checking, e.g.) seems to be possible for *small* components (small models). However, when reasoning about a composition of components, usually properties cannot be verified due to the complexity of the composed model.

In this section we introduce a verification technique which allows us to infer a property for a composed model from a (modified) property in a simplified composed model.

In the literature ([27, 62], e.g.), usually the user defines an abstraction function (with certain properties). The verification system automatically abstracts the model with respect to the given abstraction function and proves the property in the abstracted model. Often, the abstraction has to fulfill some properties such that each valid formula in the abstracted model holds in the original model, too. However, finding the right abstraction function is a difficult task.

We do not apply an abstraction function to a model to obtain an abstracted model. We assume that the model and its abstraction are given without knowing a functional relation between them. The disadvantage is that if the formula holds in the abstracted model, then we cannot conclude that the formula holds in the original model, too. This is obvious, since the abstraction may behave completely different from the original model. On the other hand it is easier to build such abstractions.

In contrast to other compositional verification proposals ([32, 5, 73], e.g.) we use the abstract models for the environment behavior.

Assume we have the components $C_1, \ldots, C_n$ together with connection function cf resulting in the composition C. Further, we want to prove formula φ for model C, but this is not possible due to complexity issues. Let us further assume that IO-abstractions A_i of C_i are given and A (similar to C) is the composition of $A_1, \ldots, A_n$. To conclude whether φ holds in C we check whether an extended version of φ holds in an extended model of A. The assumption is that automatic property proving (model checking) is possible in A extended with one component C_i. Thus, $A_1, \ldots, A_n$ have to reduce the complexity of $C_1, \ldots, C_n$; otherwise we would have the same problem as proving in C.

We connect the models $A_1, \ldots, A_n$ (to obtain A) according to $C_1, \ldots, C_n$; i.e., if there is a connection from id_1 to id_2 in the composition C, and both identifiers have a mapping from the abstract components, then the corresponding abstract identifiers are connected. This is illustrated in Fig. 2.13 and formalized in the following definition of cf^A:

Definition 2.3.1 (Abstract connection function) *Let cf^C be a connection function for the components $C_1, \ldots, C_n$. Let A_i be an IO-abstraction of C_i with respect to mapping function map_i. The connection function cf^A for $A_1, \ldots, A_n$ for the composition A is determined by*

$$\frac{cf^C(x^C) = y^C, map(x^A) = x^C, map(y^A) = y^C}{cf^A(x^A) = y^A}$$

where map is the union of the single map_i functions. Without loss of generality, we require the inputs, outputs, and state elements in $C_1, \ldots, C_n, A_1, \ldots, A_n$ to be pairwise disjoint.

We now extend the abstract composition A with all concrete components C_i and we connect each mappable input in the abstract model with the corresponding input in the concrete component. This implies that in such a composition no output of a concrete component is connected. We will use this extended composition as a first model for proving the formula φ.

Figure 2.14 shows an example for the abstract components A_1, A_2, A_3 and the concrete components C_1, C_2, C_3 with mapping functions map_1, map_2, map_3 and connection function cf^C, cf^A defined below.

$map_1(i_1) = i_5, map_1(o_1) = o_5, map_1(o_2) = o_6$
$map_2(i_2) = i_6, map_2(i_3) = i_7, map_2(i_4) = i_8, map_2(o_4) = o_8$
$map_3(o_3) = o_7$
$cf^C(i_6) = o_5, cf^C(i_7) = o_7, cf^C(i_8) = o_6$
$cf^A(i_2) = o_1, cf^A(i_3) = o_3, cf^A(i_4) = o_2$

The following definition formalizes this composition:

Definition 2.3.2 (Abstract-to-concrete composition) *Let cf^C be a connection function for the components $C_1, \ldots, C_n$. Let A_i be an IO-abstraction of C_i with respect to mapping function map_i. Let cf^A be the connection function defined by Def. 2.3.1. The component AC obtained by composing the components $A_1, \ldots, A_n, C_1, \ldots, C_n$ with respect to connection function cf defined below is called the* abstract concrete composition *of $A_1, \ldots, A_n, C_1, \ldots, C_n$ with respect to connection function cf^C.*

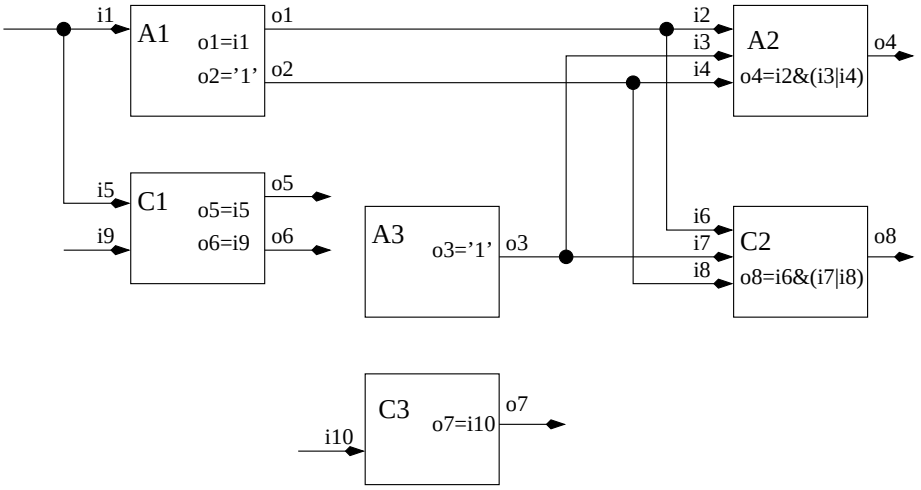
$$\frac{cf^A(x^A) = y^A}{cf(x^A) = y^A} \qquad \frac{x^A \in I_i^A, map_i(x^A) = x^C}{cf(x^C) = x^A}$$

The abstract-to-concrete composition is a component which contains all abstract and concrete components where the abstract components are connected as in the abstract composition and all inputs of the abstract components which can be mapped to concrete inputs are connected to them.

Our aim is to prove formula φ in C by proving a modified φ in the abstract composition A. The inputs and outputs in A and C are disjoint (by assumption) but we know their relation due to the mapping function. Hence, we can to substitute the inputs and outputs in formula φ to translate them to A and we first define what we mean with substitution:

Definition 2.3.3 (Formula substitution) *Let φ be a formula over input set I and output set O with respect to time period w. Let f be an injective function*

Figure 2.14 Example for an abstract concrete composition



from $I \cup O$ to any set. $\varphi[f]$ is the formula φ where all identifiers $id \in \mathrm{dom}(f)$ are substituted by $f(id)$.

Our concept of abstraction does not ensure that all inputs and outputs in the concrete composition are available in the abstract composition. This is natural, because A should be a simplification of C. The formulas for C which can be transformed to formulas for A are called *expressible in* A:

Definition 2.3.4 (Expressible formula) *Let C be the composition of the components $C_1, \ldots, C_n$ with respect to connection function cf^C. Let A_i be an IO-abstraction of C_i with respect to mapping function map_i and let φ be a formula over input set I^C and output set O^C of C with respect to time period w. Let A be the composition of $A_1, \ldots, A_n$ with cf^A as defined in Def. 2.3.1. A formula φ in the signature of C is called expressible in A if all identifiers in φ have a mapping from C:*

$$\forall\,(id, t) \in vars(\varphi)\colon\ id \in \mathrm{ran}(map)$$

where map is the union of map_i.

This implies that $\varphi[map^{-1}]$ is a formula over input set I^A and output set O^A with respect to time period w where map^{-1} is the inverse function of map (note that map is an injective function).

As stated before, we want to prove an extended version of formula φ in the abstract composition together with one concrete component. Before we come to this, we first prove an extended version of φ in the abstract composition together with *all* concrete components. If this extended formula holds, then φ holds in the composition C.

In the following theorem we extend formula φ by ψ, which states that every output in A which is connected to an input has the same value as the corresponding mapped output in the concrete component at each timepoint.

The idea here is that for a given property φ the components A_i and C_i behave in the same way with respect to their output values. Note that both components already get the same inputs by the constructed connection function:

Theorem 2.3.1 *Let AC be the abstract concrete composition of the concrete components $C_1, \ldots, C_n$ and the abstract components $A_1, \ldots, A_n$ with respect to connection function cf^C, let C be the composition of $C_1, \ldots, C_n$ with respect to cf^C and let A be the composition of $A_1, \ldots, A_n$ with respect to cf^A. Let A_i be an IO-abstraction of C_i with respect to mapping function map_i. Let φ be an expressible formula in A over input set I^C and output set O^C of C with respect to time period w. Then the following property holds:*

$$\frac{AC \models \varphi[map^{-1}] \wedge \psi}{C \models \varphi}$$

where ψ is defined as

$$\psi = \bigwedge_{1 \le i \le n} \bigwedge_{0 \le t \le w} \bigwedge_{o \in O_i^A \cap \mathrm{ran}(cf^A)} o^t = map_i(o)^t$$

and $a = b$ stands for $(a \wedge b) \vee (\neg a \wedge \neg b)$.

Proof by Contradiction. Let $AC \models \varphi[map^{-1}] \wedge \psi$ be valid and $C \models \varphi$ be not valid. This implies, that there is a sequence of input states us for the composition C and an internal state $\mathfrak{s}$, such that

$$\varphi^C(us, \mathfrak{s}) = \mathit{false}$$

where φ^C denotes the formula with respect to component C. On the other hand, there is a sequence of input states vs and an internal state $\mathfrak{s}'$ for the composition AC with the properties below such that $\varphi^{AC}(vs, \mathfrak{s}')$ holds.

$$S^C \lhd \mathfrak{s} = S^C \lhd \mathfrak{s}'$$
$$us = u^0 \cdot \ldots \cdot u^w$$
$$vs = v^0 \cdot \ldots \cdot v^w$$
$$v^t(i) = \begin{cases} u^t(cf^{*C}(i)) & i \in I_1^C \cup \ldots \cup I_n^C, cf^{*C}(i) \in I^C \\ out_C^t(us, \mathfrak{s})(cf^{*C}(i)) & i \in I_1^C \cup \ldots \cup I_n^C, cf^{*C}(i) \notin I^C \\ u^t(cf^{*C}(map(i))) & i \in \mathrm{dom}(map), cf^{*C}(map(i)) \in I^C \\ out_C^t(us, \mathfrak{s})(cf^{*C}(map(i))) & i \in \mathrm{dom}(map), cf^{*C}(map(i)) \notin I^C \\ \mathit{arbitrary} & \mathit{otherwise} \end{cases}$$

Since $\varphi^C(us, \mathfrak{s}) = \mathit{false}$, $\varphi^{AC}(vs, \mathfrak{s}') = \mathit{true}$, and the input values in us coincide with the input values in vs on the common subset, there must be a variable o^t in φ, such that $out_C^t(us, \mathfrak{s})(o) \neq out_{AC}^t(vs, \mathfrak{s}')(map^{-1}(o))$.

From ψ we know that $out_{AC}^t(vs, \mathfrak{s}')(map^{-1}(o)) = out_{AC}^t(vs, \mathfrak{s}')(o)$. Let $\mathfrak{s}_j$ and $\mathfrak{s}_j'$ be the internal states at time t for component C_j resulting from the input states us and vs, respectively.

$$\mathfrak{s}_j = S_j^C \lhd next_C^t(us, \mathfrak{s})$$
$$\mathfrak{s}_j' = S_j^C \lhd next_{AC}^t(vs, \mathfrak{s}')$$

Let u_j and v_j be the input states at time t for component C_j resulting from the sequence of input states us and vs according to Theorem 2.2.1, respectively. Let out_j be the output function for component C_j. From the previous discussion we know that $out_j(u_j, \mathfrak{s}_j)(o) \neq out_j(v_j, \mathfrak{s}'_j)(o)$ and we consider the two cases why these output values can be different:

1. $\mathfrak{s}_j = \mathfrak{s}'_j$
   The input states u_j and v_j must be different, because the corresponding output functions $out_j(u_j, \mathfrak{s}_j)(o)$ and $out_j(v_j, \mathfrak{s}_j)(o)$ are different. Hence, for $i \in \mathrm{dom}(u_j)$ there are seven cases to discuss:

   (a) $i \in I^C$. Contradiction, because $v_j(i)$ is also defined as $u_j(i)$.

   (b) $i \notin I^C, cf^{*C}(i) \notin I^C, i \in I^{AC}$. This implies $cf^{*C}(i) \in O^C$ and $u_j(i) = out^t_C(us, \mathfrak{s})(cf^{*C}(i))$. Contradiction, because $v_j(i)$ is defined as $out^t_C(us, \mathfrak{s})(cf^{*C}(i))$.

   (c) $i \notin I^C, cf^{*C}(i) \in I^C, i \in I^{AC}$. Contradiction, because $u_j(i)$ and $v_j(i)$ are both defined as $u^t(cf^{*C}(i))$.

   (d) $i \notin I^C, cf^{*C}(i) \in I^C, i \notin I^{AC}, cf^*(i) \in I^{AC}$. Let $i' = cf^*(i)$. Auxiliary statement (1) below implies $i' \in I^A$ and (2) implies $i' \in \mathrm{dom}(map)$. $u_j(i)$ is defined as $u^t(cf^{*C}(i))$ and $v_j(i)$ is defined as $v^t(cf^*(i)) = v^t(i') =_{def} u^t(cf^{*C}(map(i'))) =_{(3)} u^t(cf^{*C}(i))$. Contradiction.

   (e) $i \notin I^C, cf^{*C}(i) \in I^C, i \notin I^{AC}, cf^*(i) \notin I^{AC}$. Let $i' = cf^*(i)$. Hence, $i' \in O^A, i' \in \mathrm{dom}(map)$. This case is not possible, because (3) is violated (the connection in the concrete composition eventually leads to an input and the connection in the abstract composition eventually leads to an output).

   (f) $i \notin I^C, cf^{*C}(i) \notin I^C, i \notin I^{AC}, cf^*(i) \in I^{AC}$. This implies
   $u_j(i) = out^t_C(us, \mathfrak{s})(cf^{*C}(i))$,
   $v_j(i) = v^t(cf^*(i)) =_{(2)} out^t_C(us, \mathfrak{s})(cf^{*C}(map(cf^*(i))))$
   $=_{(3)} out^t_C(us, \mathfrak{s})(cf^{*C}(cf^{*C}(i))) = out^t_C(us, \mathfrak{s})(cf^{*C}(i))$.
   Contradiction.

   (g) $i \notin I^C, cf^{*C}(i) \notin I^C, i \notin I^{AC}, cf^*(i) \notin I^{AC}$. Let $o = cf^{*C}(i)$ and $o' = cf^*(i)$. (2) implies $o' \in \mathrm{dom}(map)$, (3) implies $map(o') = o$, because $cf^{*C}(id_1) = id_2 \Leftrightarrow id_1 = id_2$ for $o \in O^C$. We continue the proof for the output values of o and o'. The proof terminates, because the dependency relation dep in Theorem 2.2.1 is acyclic.

   Auxiliary statements:

   (1) $id' = cf(id) \Rightarrow id' \in I^A \cup O^A$

   (2) $id' = cf(id) \Rightarrow id' \in \mathrm{dom}(map)$

   (3) $id \in I^C_1 \cup \ldots \cup I^C_n$: $cf^{*C}(map(cf^*(i))) = cf^{*C}(i)$
   Proof: Let $i_1 = i, i_{k+1} = cf^C(i_k), j_1 = i, j_{k+1} = cf(j_k)$. Hence, $map(j_{k+1}) = i_k$ is an invariant.

2. $\mathfrak{s}_j \neq \mathfrak{s}'_j$
   The time point t must be greater than 0, because the initial state $\mathfrak{s}'$ coincide on the set S^C with the initial state $\mathfrak{s}$. Hence, there is a time

point t' with $0 \leq t' < t$, such that the corresponding states $\mathfrak{s}$ and $\mathfrak{s}'$ at time t' are equal and the input states are different. At that time we continue the proof with the first case. □

Theorem 2.3.1 seems to be useless, because the model AC (in which we prove the extended property of φ) is even larger than C. Further, we have to prove the equality of several outputs. Fortunately, in the proof of Theorem 2.3.1 we can reduce the number of equalities to prove and we can divide the proof in AC to n proofs in n smaller models.

We first analyze the set of outputs where it is sufficient to prove equality. Therefore, we define an extended version of a dependency set for an output which takes time into account.

A pair (i, t) is in a dependency set $I^{dep}_{o,w}$, if the output value of o at time w depends on the input value of i at time t. A set $I^{dep}_{o,w}$ is a dependency set for o with respect to time period w, if pairs (i, t) not in $I^{dep}_{o,w}$ can not influence the output value of o at time w:

Definition 2.3.5 (Dependency set)
Let $C = (I, O, S, next, out)$ be a component. A set

$$I^{dep}_{o,w} \subseteq I \times \mathbb{N}_w$$

is a dependency set for output $o \in O$ with respect to time period w if the following condition is true:

$$\forall\, \mathfrak{is}_1, \mathfrak{is}_2 \in \mathfrak{I}^{\geq w+1}, \mathfrak{s} \in \mathfrak{S}\colon \\ (\forall\, (i,t) \in I^{dep}_{o,w}\colon \mathfrak{is}_1^t(i) = \mathfrak{is}_2^t(i)) \Rightarrow out^w(\mathfrak{is}_1, \mathfrak{s})(o) = out^w(\mathfrak{is}_2, \mathfrak{s})(o)$$

Note that we use the notation $\mathfrak{is}^t$ where $\mathfrak{is}$ is a sequence of input states and $\mathfrak{is}^t$ is the t^{th} input state (starting counting at zero).

Similar to the dependency function in Def. 2.2.2 we define a dependency function which also takes time into account. We further add those dependencies which occur due to the connection function. For example, if an output value for o at time t depends on the input i_1 at t_1 and i_1 is connected to o_1, then we add all dependencies of o_1 at time t_1 to the dependencies of o at t. This is formalized in the following definition.

Definition 2.3.6 (Dependency function)
Let $C_1, \ldots, C_n$ be components with

$$\begin{aligned} C_i &= (I_i, O_i, S_i, next_i, out_i) \\ I &= I_1 \cup \cdots \cup I_n \\ O &= O_1 \cup \cdots \cup O_n \end{aligned}$$

Let cf be a connection function for $C_1, \ldots, C_n$. Let $I^{dep}_{o,t}$ be a dependency set for $o \in O_i$ in component C_i with respect to time period t where $0 \leq t \leq w$. A function

$$I^{dep}_w\colon O \times \mathbb{N}_w \to \mathbb{P}((I \cup O) \times \mathbb{N}_w)$$

satisfying the rules below, is called a dependency function with respect to $C_1, \ldots, C_n$*, connection function* cf*, and time period* w*.*

$$\frac{(i,t') \in I^{dep}_{o,t}}{(i,t') \in I^{dep}_{w}(o,t)} \qquad \frac{(i,t') \in I^{dep}_{w}(o,t),\, cf(i) = id}{(id,t') \in I^{dep}_{w}(o,t)}$$

$$\frac{(id_1,t_1) \in I^{dep}_{w}(o,t), (id_2,t_2) \in I^{dep}_{w}(id_1,t_1)}{(id_2,t_2) \in I^{dep}_{w}(o,t)}$$

For an output o and given time point t, this dependency function $I^{dep}_w(o,t)$ defines the set of inputs and outputs at certain time points upon which the value of o at t depends.

When we proof Theorem 2.3.1 we can see that it is sufficient to show the equality for those outputs and time points where the formula φ contains a subformula id^t and id^t depends on.

The function $vars(\varphi)$ in Def. 2.1.8 computes the set of formula variables in φ. For each variable o^t in this set, we compute the set of variables where the variable o^t depends on. The union of these sets is an upper bound for the equality check in Theorem 2.3.1.

Lemma 2.3.1 *For given dependency function* I^{dep}_w *with respect to components* $C_1, \ldots, C_n$*, connection function* cf*, and time period* w*, we can reduce the proof obligations in Theorem 2.3.1 to the following* ψ*:*

$$\psi = \bigwedge_{1 \le i \le n} \bigwedge_{0 \le t \le w} \bigwedge_{o \in O^A_i \cap \mathrm{ran}(cf^A), (o,t) \in deps} o^t = map_i(o)^t$$

where $deps = \bigcup \{I^{dep}_w(o,t) \cup \{(o,t)\} \mid (o,t) \in vars(\varphi)\}$.

Proof. Follows immediately from the proof of Theorem 2.3.1 by analyzing for which outputs the equation is needed. □

The abstract concrete composition AC contains all abstract and all concrete components. As can be seen in Def. 2.3.2, the outputs of the concrete components are not connected and therefore the concrete components cannot influence the behavior of AC. This is the reason why we can define reduced versions of AC where we use all abstract components and only one concrete component.

We define a model AC_i which we obtain from AC by removing the concrete components $C_{j \neq i}$ and the corresponding connections:

Definition 2.3.7 (Restricted abstract-to-concrete composition) *Let* cf^C *be a connection function for the components* $C_1, \ldots, C_n$*. Let* A_i *be an IO-abstraction of* C_i *with respect to mapping function* map_i*. Let* cf^A *be the connection function defined by Def. 2.3.1. The component* AC_i *obtained by composing the components* $C_i, A_1, \ldots, A_n$ *with respect to connection function* cf *(as defined in Def. 2.3.2 for given* i*) is called the* abstract concrete composition *of* $C_1, \ldots, C_n, A_1, \ldots, A_n$ *with respect to connection function* cf^C *restricted to component* C_i*.*

The proof obligations in Theorem 2.3.1 for the model AC can now be divided into n models where in each model we have only one concrete component. Since

the concrete components cannot influence the behavior of the AC model we can also split the formula ψ into n sub-formulas. This is stated in the following theorem:

Theorem 2.3.2 *Let AC_i be the abstract concrete composition of the concrete components $C_1, \ldots, C_n$ and the abstract components $A_1, \ldots, A_n$ with respect to connection function cf^C restricted to component C_i. Let C be the composition of $C_1, \ldots, C_n$ with respect to cf^C and let A be the composition of $A_1, \ldots, A_n$ with respect to cf^A. Let A_i be an IO-abstraction of C_i with respect to mapping function map_i. Let I_w^{dep} be a dependency function with respect to C_i, connection function cf^C, and time period w. Let φ be an expressible formula in A over input set I^C and output set O^C of C over time period w. Then the following property holds:*

$$\frac{AC_1 \models \varphi[map^{-1}] \wedge \psi_1, \ \ldots, \ AC_n \models \varphi[map^{-1}] \wedge \psi_n}{C \models \varphi}$$

where ψ_i is defined as

$$\psi_i = \bigwedge_{0 \leq t \leq w} \ \bigwedge_{o \in O_i^A \cap \mathrm{ran}(cf^A), (o,t) \in deps} o^t = map_i(o)^t$$

where $deps = \bigcup \{I_w^{dep}(o,t) \cup \{(o,t)\} \mid (o,t) \in vars(\varphi)\}$.

Proof. Follows immediately from Lemma 2.3.1 and the fact that the outputs of the concrete components are not connected. □

Usage of Theorem 2.3.2

Let φ be a formula for the composition C and expressible in A. We want to prove φ in C, but this is not possible due to complexity issues. We try to prove instead the modified version of φ in $AC_1, \ldots, AC_n$. If we succeed, then by Theorem 2.3.2 we know that φ holds in C. If the property fails in one of these models, then by model checking, we get an input state where the property fails. We analyze this counter example for AC_i, if it is a counter example for C, then we know that the property does not hold in C. If it is not a counter example in C, then we have to modify our abstract model or the concrete composition, since they behave differently from each other.

Remarks

For given concrete components C_i and abstract components A_i we can automatically apply the introduced verification technique for a formula φ, because the models AC_i and the extended version of φ can be generated automatically. The crucial point is how to define the abstract models. They have to contain all features which we are interested in proving and they also have to be simple enough, such that verification is possible.

This verification technique is in particular useful when the abstract models are already part of the specification for the designed hardware. In such a case high-level properties can be proved in the abstract models during the specification phase and the final implementation can be checked against the abstract models with the introduced verification technique. The implementation can then be seen as a refinement of the abstract models.

2.4 Related Work

Our component definition is similar to the definition of Mealy automata [21]. It is well-known that the composition of Mealy automata is not necessarily a Mealy automaton. In our work, we do not commit to any particular algorithm to detect whether the composition of given Mealy machines is a Mealy machine. Moreover, we use a notion of dependency set to decide whether the composition is a valid component. The Mealy composition problem is related to the detection of hazards in digital circuits. Some algorithms to detect hazards are described in [48].

We compose components by connecting inputs and outputs with a so-called connection function. Usually, in the literature [32, 50] automata are composed by name sharing, i.e. inputs and outputs with the same name are connected. We do not use the technique of name sharing, because we need a more flexible way of composition.

The most important difference between our proposal and others [40, 27] is the notion of abstraction. Our notion of abstraction does not necessarily imply any functional relation between the concrete and the abstract model. Roughly spoken, our concrete models are a refinement of the abstract models with respect to *some* properties.

The best studied approach to compositional verification is the assume-guarantee paradigm [58, 62, 5, 40], where component properties are verified contingent on properties that are assumed of the environment [62]. This is different from our approach, because we use abstract models to simulate the environment. Abstract models for the environment are also used in lazy compositional verification [62]. However, that approach uses for each component a model for the environment, whereas in our approach the environment is given by the composition of the abstract models. Additionally, the compositional verification is completely different from ours.

This is not the first proposal for a component concept for Abstract State Machines. For instance, [2] introduces XASM as a component based language. However, a major weakness of that language is the lack of support in modeling hierarchy which is of utmost importance for modeling complex architectures (see [3]).

The main contribution of this chapter is the introduction of a component concept for Abstract State Machines. This concept is especially useful for hardware design. We introduced a component based verification technique, where system properties about the composition of concrete models can be proven in a simpler model. There is no need to split system properties into component properties as needed for other compositional approaches (assume-guarantee, e.g.). Furthermore, if the concrete and abstract models are given, then our proof technique can be applied without user interaction.

Chapter 3

Execution of Abstract State Machines

In the previous chapters we introduced structuring and composition concepts for ASMs. We can use these concepts to specify systems, but up to now we can not execute them. Executable specifications are desirable for validation purpose (as can be seen in [64], e.g.).

The question arises how we can execute ASM specifications. There are at least two possibilities: (i) building a new tool from scratch or (ii) extending an existing one. Building a new tool and designing a new language is an interesting subject, but it needs a lot of man power. On the other hand, extending an existing tool is also very interesting and more effective, because we can use existing work. Hence, we decided to extend an existing tool.

We take the TkGofer [67] interpreter which is a conservative extension of Gofer [43] (a functional programming language similar to Haskell [66]) and extend it in order to make it suitable for executing Abstract State Machines.

In Section 3.1 we discuss the combination of functional programming and ASMs. As a result of this discussion we present in Section 3.2 a natural semantics for a functional programming language with lazy evaluation. Section 3.3 extends and modifies the semantics presented in Section 3.2 with respect to language features needed for executing ASMs. Eventually, Section 3.4 describes the semantics extension for the *seq* construct introduced in Section 1.2.

3.1 Functional Programming and ASMs

Combining functional programming and ASMs can be done by integrating ASMs into functional programming; in our case into TkGofer. The main question is how to integrate a notion of state in a functional language.

There is a standard answer to this question in the functional community: *use monads*. For an introduction and discussion about monads we refer the reader to [69]. Here, we assume the reader has basic knowledge of monads and we now discuss whether we can use them to represent the ASM state.

Consider an ASM specification with two nullary dynamic functions f and g. We can define a state monad [49] where the monad captures the two dynamic functions. We assume, that this monad is defined as the type *MyState a* where

a is the result type of the corresponding action of the monad. We assume, there are two functions for updating f and g and two functions for accessing their values. These functions have the following signatures where F and G are assumed to be the types of f and g, respectively.

$$\begin{array}{ll} \mathit{update\_f} & : F \to \mathit{MyState}\,() \\ \mathit{update\_g} & : G \to \mathit{MyState}\,() \\ \mathit{read\_f} & : \mathit{MyState}\,F \\ \mathit{read\_g} & : \mathit{MyState}\,G \end{array}$$

These functions are sufficient to represent an ASM with state functions f and g as a monadic expression. For instance, consider the following ASM rule:

$$f := f + g$$

It can now be represented using the *do-notation* for monads [41] as follows:

$$\begin{array}{l} \textbf{do}\ f \leftarrow \mathit{read\_f} \\ \quad\ g \leftarrow \mathit{read\_g} \\ \quad\ \mathit{update\_f}\ (f + g) \end{array}$$

The first line stores the value of the dynamic function f in the variable f; the second line is similar for g. In the last line, the function f is updated to the sum of the values of the variables f and g.

There is an obvious disadvantage with monads: we can not access the values of dynamic functions on the expression level. More precisely, we have to transform ASM rules by lifting dynamic function accesses to the monadic level as indicated in the above example. This is a general problem when using monads. Due to this, we do not use monads to represent state. Hence, the question is what else can we do?

There seems to be no appropriate solution in a pure functional language. Hence, we extend and modify an existing system, such that it can deal with dynamic functions and parallel updates.

The system we will extend is TkGofer; it is based on lazy evaluation. Instead of describing the implementation, we discuss the extension on the semantics level where we introduce new primitive expressions to support the ASM features.

The aim is to extend the semantics in a way, such that we can adapt the Gofer implementation [44] accordingly (and also TkGofer).

3.2 Lazy Evaluation

In this section we present a semantics for lazy evaluation. The semantics is taken from [47] where it is described in detail. We repeat the definitions here, because we will extend them in the next two sections.

The language considered in [47] is based on the lambda-calculus [4] extended with *let*-expressions and is normalized to a restricted syntax. This normalized syntax is characterized in [47] as: "normalized λ-expressions have two distinguishing features: all bound variables are distinct, and all applications are applications of an expression to a variable". The normalized λ-expressions are

Figure 3.1 Basic λ-expressions

$$\frac{}{\Gamma \vdash \lambda x.e \Downarrow_i^v \Gamma \vdash \lambda x.e} \quad \text{(lambda)}$$

$$\frac{\Gamma \vdash e \Downarrow_i^v \Gamma_1 \vdash \lambda y.e', \;\; \Gamma_1 \vdash e'[x/y] \Downarrow_i^v \Gamma_2 \vdash z}{\Gamma \vdash e\, x \Downarrow_i^v \Gamma_2 \vdash z} \quad \text{(apply)}$$

$$\frac{\Gamma \vdash e \Downarrow_i^0 \Gamma_1 \vdash z, \;\; upd(x)}{(\Gamma, x \mapsto e) \vdash x \Downarrow_i^0 (\Gamma_1, x \mapsto z) \vdash \hat{z}} \quad \text{(variable)}$$

$$\frac{(\Gamma, x_1 \mapsto e_1, \ldots, x_n \mapsto e_n) \vdash e \Downarrow_i^v \Gamma_1 \vdash z}{\Gamma \vdash \texttt{let}\; x_1 = e_1, \ldots, x_n = e_n \;\texttt{in}\; e \Downarrow_i^v \Gamma_1 \vdash z} \quad \text{(let)}$$

defined in [47] by the following syntax:

$$\begin{aligned} &x \in \mathit{Var} \\ &e \in \mathit{Exp} ::= \lambda x.e \\ &\qquad\quad \mid \;\; e\, x \\ &\qquad\quad \mid \;\; x \\ &\qquad\quad \mid \;\; \textbf{let}\; x_1 = e_1, \ldots, x_n = e_n \;\textbf{in}\; e \end{aligned}$$

The dynamic semantics is defined in [47] by deduction rules and is shown in Fig. 3.1 where the following naming conventions are used:

$$\begin{aligned} &\Gamma \quad \in \mathit{Heap} = \mathit{Var} \rightharpoonup \mathit{Exp} \\ &z, y \in \mathit{Val} \quad ::= \lambda x.e \end{aligned}$$

For the time being, ignore the super-script and sub-script values at the symbol $\Downarrow$ in Fig. 3.1 which are not present in the original definition. The symbol denotes reduction of terms and is explained in [47] as: "Judgments of the form $\Gamma \vdash e \Downarrow \Gamma_1 \vdash z$ are to be read, the term e in the context of the set of bindings Γ reduces to the value z together with the (modified) set of bindings Γ_1." Ignore also the predicate $upd(x)$ in the *variable* rule which is not present in the original definition, too. If z is an expression, then $\hat{z}$ denotes the expression where all bound variables are replaced by fresh variables ($\rightsquigarrow$ alpha-conversion).

The *lambda* rule in Fig. 3.1 states that λ-expressions are not further reduced. This is not surprisingly, because in lazy evaluation, expressions are evaluated only when necessary.

An application $e\,x$ is reduced by first reducing e to a lambda-expression $\lambda y.e'$, and then reducing the function body e' with the formal parameter y substituted by the argument x.

If a variable binding $x \mapsto e$ is in the heap, then we reduce a variable x by reducing the expression e (see *variable* rule). Let-expressions introduce new variable bindings (see *let* rule).

For more details about these definitions, we refer the reader to [47].

Figure 3.2 Nullary dynamic functions

$$\frac{\Gamma \vdash \mathit{init} \Downarrow_1^1 \Gamma_1 \vdash z, \quad (\Gamma_1, \mathit{init}(x) \mapsto z, \mathit{old}(x) \mapsto z) \vdash \texttt{dyn0}\, x \Downarrow_i^v \Gamma_2 \vdash y}{\Gamma \vdash \texttt{initVal}\, x\, \mathit{init} \Downarrow_i^v \Gamma_2 \vdash y} \quad \text{(init)}$$

$$\frac{}{\Gamma \vdash \texttt{dyn0}\, x \Downarrow_i^0 \Gamma \vdash \texttt{dyn0}\, x} \quad \text{(read)}$$

$$\frac{}{(\Gamma, \mathit{old}(x) \mapsto z) \vdash \texttt{dyn0}\, x \Downarrow_0^1 (\Gamma, \mathit{old}(x) \mapsto z) \vdash z} \quad \text{(read)}$$

$$\frac{}{(\Gamma, \mathit{init}(x) \mapsto z) \vdash \texttt{dyn0}\, x \Downarrow_1^1 (\Gamma, \mathit{init}(x) \mapsto z) \vdash z} \quad \text{(read)}$$

3.3 Lazy Evaluation and ASMs

Usually, one would semantically distinguish between expressions and rules. Extending a lazy evaluation semantics would then mean to define the ASM semantics on top of a semantics for expressions as it is usually done in the literature.

The problem arises when we want to adapt the TkGofer system in the same way as we adapted the semantics, because we have to implement the ASM semantics on top of expression evaluation. Since the TkGofer run-time system is heavily based on expressions, this would probably be nearly impossible.

Another possibility is to use expressions for representing the ASM features like parallel updates and rules. This has the advantage, that we stay in the functional world and therefore, it would be easier to extend the TkGofer system.

In the following subsections, we extend and modify the semantics for lazy evaluation in order to make it suitable for ASMs. In particular, we introduce new primitive expressions to define dynamic functions, to update them, and to execute rules.

3.3.1 Nullary Dynamic Functions

Nullary dynamic functions have an initial value and can be updated during a run. Let us first discuss how to define nullary dynamic functions and how to represent the function values in the heap structure.

We use an expression *initVal* to define a nullary dynamic function and we extend our expression syntax as follows:

$$\mathit{Exp} ::= \ldots \mid \texttt{initVal}\, x_1\, x_2$$

The expression $\texttt{initVal}\, x_1\, x_2$ can be viewed as a primitive function taking two arguments. The first is the name of the dynamic function and the second its initial value. Using this syntax, the following term is valid[1] and evaluates to 10.

let $f = \texttt{initVal f}\, 5$
in $f + f$

[1] We abstract from the fact that function arguments must be variables, see [47] for the transformation algorithm.

Figure 3.3 Firing rules

$$\frac{\Gamma \vdash \mathit{lhs} \Downarrow_i^0 \Gamma_1 \vdash \texttt{dyn0}\ n, \quad \Gamma_1 \vdash \mathit{rhs} \Downarrow_i^1 \Gamma_2 \vdash z}{\Gamma \vdash \texttt{update}\ \mathit{lhs}\ \mathit{rhs} \Downarrow_i^v (\Gamma_2, \mathit{new}(n) \mapsto z) \vdash \texttt{rule}} \quad \text{(upate)}$$

$$\frac{\Gamma \vdash r \Downarrow_i^v (\Gamma_1, \mathit{old} \mapsto o, \mathit{new} \mapsto n) \vdash \texttt{rule}}{\Gamma \vdash \texttt{fire1}\ r \Downarrow_i^v (\Gamma_1, \mathit{old} \mapsto o \oplus n) \vdash \texttt{io}} \quad \text{(fire)}$$

$$\frac{}{\Gamma \vdash \texttt{rule} \Downarrow_i^v \Gamma \vdash \texttt{rule}} \qquad \frac{}{\Gamma \vdash \texttt{io} \Downarrow_i^v \Gamma \vdash \texttt{io}} \quad \text{(rule, io)}$$

Figure 3.2 defines the semantics for initializing and reading nullary dynamic functions. At the end of this subsection we will explain the rules in detail.

We extend the universe of values by an element `dyn0` to represent nullary dynamic functions as values:

$$\mathit{Val} ::= \ldots \mid \texttt{dyn0}\ x$$

The argument to `dyn0` denotes the name of the dynamic function (the same as specified for `initVal`). We also extend our heap by two functions *init* and *old* for the current and the initialization values of dynamic functions:

$$\begin{aligned} \mathit{Heap} &= \ldots \cup \{\mathit{old}, \mathit{init}\} \\ \mathit{old} &: \mathit{Id} \nrightarrow \mathit{Val} \\ \mathit{init} &: \mathit{Id} \nrightarrow \mathit{Val} \end{aligned}$$

The heap function *old* assigns the current value to a dynamic function whereas *init* assigns the initial value. Initially, the current value and the initial value are the same.

We store the initial value, because it might be that we need the initial value of a dynamic function, but the function already contains a newer value. Consider the following scenario (see the example below) where the initial value of a dynamic function g depends on the initial value of a dynamic function f. Assume now, that f is evaluated and updated in the first step, but g is not evaluated (lazy evaluation). If g is evaluated in the second step, then we initialize the dynamic function g using the value of f. In our scenario, for this initialization we must use the intial value of f and not the current value. In the following example we use a not furthermore described function *ioseq* to denote sequential execution. Such a function can be implemented in Gofer using monads (see [45], e.g.).

$$\begin{aligned} &\textbf{let}\ \ f = \texttt{initVal f}\ 1, \\ &\qquad g = \texttt{initVal g}\ f \\ &\textbf{in}\ \ \ \texttt{ioseq}\,(\mathit{fire}1\,(f := f + 1))\,(\mathit{print}\ g) \end{aligned}$$

We will describe the update operator := and the *fire*1 rule in the next subsection. However, the value of g would be 2 and not 1 if we would not store the intial value of f.

Figures 3.1 and 3.2 use an annoted version of the reduction symbol $\Downarrow_i^v$ which is not present in the original semantics definition in [47]. The variables v and i denote in which context an expression is evaluated. If i is equal to 1, then we evaluate expressions with respect to initial values of dynamic functions. For instance, consider the reduction of the initial value for a dynamic function in Fig. 3.2 (rule *init*).

The variable v denotes whether a reduction should stop at `dyn0` x or whether we want to reduce this value to a basic value. By a basic value we mean a value different from `dyn0`. If v is 0, then dynamic functions are not evaluated. We need this feature for the update rule in Fig. 3.3.

Consider now the rules in Fig. 3.2. The *init* rule first evaluates the expression *init* to the value z. Now z is stored as the initial and current value of x and x is evaluated to y using one of the *read* rules.

The first *read* rule will be used in Fig. 3.3 where we need the name of the dynamic function on the lefthand-side of a function update. Therefore, the value `dyn0` x is not further reduced. The last two *read* rules are used to get the function value. The first of them returns the current value and the last returns the initial value.

3.3.2 Firing Rules

Similar to the definition of dynamic functions with *initVal*, we provide a primitive function *update* for function updates (see Fig. 3.3). Therefore, we extend the expression syntax:

$$\mathit{Exp} ::= \ldots \mid \texttt{update}\ x_1\ x_2$$

The arguments for the *update* function are the lefthand-side and righthand-side expressions of the update. Usually, an update is written as *lhs* := *rhs* instead of `update` *lhs rhs*.

Consider now the update definition in Fig. 3.3. We evaluate the lefthand-side expression to a dynamic function `dyn0` x. It is important to *evaluate* the righthand-side expression (in contrast to the usual lazy evaluation of arguments), because the expression might contain dynamic functions which have to be evaluated in the current state. We store the update itself in the heap using a function *new*:

$$\begin{aligned} \mathit{Heap} &= \ldots \cup \{\mathit{new}\} \\ \mathit{new} &: \mathit{Id} \nrightarrow \mathit{Val} \end{aligned}$$

In the definition of *update* in Fig. 3.3, we do not overwrite the value in *old*, because the value might be used in succeeding updates in the same state.

We introduce a special value `rule` to ensure that function updates, expressions, and firing of rules are nested correctly (rules can not be used in expressions, e.g.).

$$\begin{aligned} \mathit{Val} &::= \ldots \mid \texttt{rule} \mid \texttt{io} \\ \mathit{Exp} &::= \ldots \mid \texttt{fire1}\ x \end{aligned}$$

Rules can be fired with *fire*1 where we require that the argument of *fire*1 evaluates to value `rule`. In this case, the *fire*1 expression evaluates to value `io` which corresponds to the IO type in TkGofer (see rule *fire* in Fig. 3.3).

Figure 3.4 Unary dynamic functions

$$\frac{}{\Gamma \vdash \texttt{initFun}\, x \Downarrow_i^v \Gamma \vdash \texttt{dyn1}\, x} \quad \text{(init)}$$

$$\frac{}{\Gamma \vdash \texttt{dyn1}\, x \Downarrow_i^v \Gamma \vdash \texttt{dyn1}\, x} \quad \text{(read)}$$

$$\frac{\Gamma \vdash e \Downarrow_i^1 \Gamma_1 \vdash \texttt{dyn1}\, n, \quad \Gamma_1 \vdash x \Downarrow_i^1 (\Gamma_2, \mathit{old}(n, y) \mapsto z) \vdash y}{\Gamma \vdash e\, x \Downarrow_i^1 (\Gamma_2, \mathit{old}(n, y) \mapsto z) \vdash z} \quad \text{(apply)}$$

$$\frac{\Gamma \vdash e \Downarrow_i^0 \Gamma_1 \vdash \texttt{dyn1}\, n}{\Gamma \vdash e\, x \Downarrow_i^0 \Gamma_1 \vdash (\texttt{dyn1}\, n)\, x} \quad \text{(apply)}$$

$$\frac{\Gamma \vdash \mathit{lhs} \Downarrow_i^0 \Gamma_1 \vdash (\texttt{dyn1}\, n)\, x, \quad \Gamma_1 \vdash x \Downarrow_i^1 \Gamma_2 \vdash y, \quad \Gamma_2 \vdash \mathit{rhs} \Downarrow_i^1 \Gamma_3 \vdash z}{\Gamma \vdash \texttt{update}\, \mathit{lhs}\, \mathit{rhs} \Downarrow_i^v (\Gamma_3, \mathit{new}(n, y) \mapsto z) \vdash \texttt{rule}} \quad \text{(upd)}$$

Firing a rule means to make dynamic function updates visible. Thus, we take the values in *old* and overwrite them with the values stored in *new*. This is done by the operator $\oplus$ in Fig. 3.3:

$$o \oplus n = \{(f, z) \mid (f, z) \in o, f \notin \mathrm{dom}(n)\} \cup n$$

Our semantics definition of function updates is very convenient and powerful. For instance, we can define an alias to a dynamic function and update the alias instead of the dynamic function itself. The semantics is the same as directly updating the dynamic function.

Consider the following example, where the lefthand-side of the update is an *if-then-else* expression:

$$\begin{array}{lll} \textbf{let} & f & = \texttt{initVal f}\, 5, \\ & g & = \texttt{initVal g}\, 3, \\ & \mathit{useF} & = \ldots \\ \textbf{in} & \textbf{let} & a = \textbf{if}\ \mathit{useF}\ \textbf{then}\ f\ \textbf{else}\ g \\ & \textbf{in} & \texttt{update}\, (a := a + f) \end{array}$$

In general, we can use any term on the lefthand-side if it evaluates to a dynamic function. This is more useful when dealing with unary dynamic functions which can then be abbreviated by nullary variables. On the other hand, there is also a disadvantage with this semantics, because in general only at run-time, we can detect whether the lefthand-side of an update really evaluates to a dynamic function.

3.3.3 Unary Dynamic Functions

Similar to nullary dynamic functions, we define unary dynamic functions by using an expression $\texttt{initFun}\, x$ where x is the name of the dynamic function. Hence, we extend our expression syntax by this primitive function:

$$\mathit{Exp} ::= \ldots \mid \texttt{initFun}\, x$$

For unary dynamic functions we do not consider initialization in this semantics, because an initialization for a unary function would be a finite mapping; to represent such mappings we would have to deal with tuple and list expressions. However, the semantics can be extended for initializing unary dynamic functions similarly to the initialization of nullary dynamic functions.

As can be seen in Fig. 3.4, the `initFun` x expression evaluates to the value `dyn1` x which corresponds to `dyn0` x for nullary functions. Hence, we extend the universe of values:

$$\mathit{Val} ::= \ldots \mid \texttt{dyn1}\ x$$

Since a unary function can only be evaluated together with an argument, the *read* rule for `dyn1` x in Fig. 3.4 evaluates to itself (similar to the rule for lambda abstraction).

Let us now consider the unary function update. Usually, a function update looks syntactically as follows:

$$f\ a := t$$

This illustrates that the operator := is a function with two arguments of the same type and the lefthand-side expression of the operator must represent a unary dynamic function application. The problem is to split the function application into the function symbol and the function argument. This may look trivial, but in fact, it is a non-trivial problem and it may become clearer if we write the above example with the *update* function instead of the operator:

$$\texttt{update}\ (f\ a)\ t$$

Note that this expression is translated to a restricted syntax where only variables are allowed as function arguments (see [47] for the transformation algorithm):

$$\begin{array}{ll}\textbf{let} & x = f\ a \\ \textbf{in} & \texttt{update}\ x\ t\end{array}$$

This implies that the *update* function does not get the function application (the lefthand-side term of the update) as its argument and therefore we must evaluate the variable x in the above example to get the function symbol to be updated.

The crucial point is how much has to be evaluated; if we evaluate x completely, then we would return the result of the function application and not the application itself. Hence, we distinguish two cases for evaluating function applications. Both cases are shown in Fig. 3.4 by the two *apply* rules.

The first *apply* rule is used in the context of a reduction to a basic value ($v = 1$). The rule evaluates the expression e to a dynamic function, the argument x to a value, and returns the value which is stored in the heap function *old* (similarly to the *read* rule for nullary dynamic functions).

The second *apply* rule is used in the update rule where the lefthand-side expression of the update is reduced in context $v = 0$. The rule evaluates the expression e to a value `dyn1` n and returns the function application with its argument. As can be seen in the *upd* rule, this application is used to split the function symbol from the argument of the expression *lhs*. Afterwards, the

function argument and the righthand-side of the update are evaluated and the update is stored in the heap function *new* (similarly to nullary updates).

As already stated, this technique of function updates allows very flexible updates. For instance, consider the following example where the lefthand-side expression is an *if-then-else* expression.

$$\begin{array}{ll}\textbf{let} & f = \texttt{initFun}\, f, \\ & x = \ldots, \\ & a = \textbf{if}\ x = 5\ \textbf{then}\ f\,2\ \textbf{else}\ f\,3 \\ \textbf{in} & \texttt{update}\, a\, 7\end{array}$$

In the example, depending on the value of x, either $f\,2$ or $f\,3$ is updated.

The disadvantage is that we can not statically ensure, that the lefthand-side expression represents a unary function update.

3.3.4 Referential Transparency

Referential transparency is an important property in functional languages. It means that an expression always evaluates to the same value no matter when we evaluate the expression. Obviously, this property is not true in our semantics extension, because the value of a dynamic function depends on the current state. This leads to some problems in our definitions. For instance, consider the following example:

$$\begin{array}{ll}\textbf{let} & f = \texttt{initVal}\, f\, 1, \\ & g = f + 1 \\ \textbf{in} & \texttt{ioseq}\,(\texttt{fire1}\,(f := g))\,(\mathit{print}\, g)\end{array}$$

The expected output is 3, because in the first step, $g = f + 1 = 1 + 1 = 2$ is assigned to f and the value of $g = f + 1 = 2 + 1 = 3$ is printed in the second step. However, in our semantics up to now, the value 2 is printed, because in the first step the definition of g is set to 2 and not newly reevaluated in the second step. Indeed, the fact that variable definitions are overwritten by the evaluated value is the problem in our semantics (see rule *variable* in Fig. 3.1. Hence, one solution might be to prevent overwriting of definitions, but then we would have a very inefficient language, because expressions would not be shared.

A more useful strategy is to analyze for which definitions the referential transparency is violated. The remaining definitions can be safely overwritten by the evaluated value. The referential transparency is violated for a function definition if the definition transitively depends on a dynamic function, i.e., on `initVal` or `initFun`. We will come to this later.

Usually, a functional programming language supports top-level definitions and not only one functional expression as in our language. For instance, consider the following Gofer program:

$$\begin{array}{ll}\mathit{fac}\, n & = \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n * \mathit{fac}(n-1) \\ g & = \mathit{fac}\, 7 \\ h & = g + \mathit{fac}\, 2\end{array}$$

Such top-level definitions must be represented in our syntax by a top-level *let* expression. Furthermore, usually only top-level definitions are stored in a global

Figure 3.5 Variables and dynamic expressions

$$\frac{\Gamma \vdash e \Downarrow_i^v \Gamma_1 \vdash z, \quad \neg upd(x)}{(\Gamma, x \mapsto e) \vdash x \Downarrow_i^v (\Gamma_1, x \mapsto e) \vdash \hat{z}} \quad \text{(variable)}$$

$$\frac{\Gamma \vdash e \Downarrow_i^0 \Gamma_1 \vdash y, \quad \Gamma_1 \vdash y \Downarrow_i^1 \Gamma_2 \vdash z, \quad upd(x),}{(\Gamma, x \mapsto e) \vdash x \Downarrow_i^1 (\Gamma_2, x \mapsto y) \vdash \hat{z}} \quad \text{(variable)}$$

$$\frac{\Gamma \vdash \hat{e} \Downarrow_i^v \Gamma_1 \vdash z}{(\Gamma, x \mapsto e) \vdash \texttt{dynamic}\, x \Downarrow_i^v (\Gamma_1, x \mapsto e) \vdash z} \quad \text{(dynamic)}$$

environment table. Hence, due to technical reasons, we have to restrict our analysis (dependence on dynamic functions) to the variable definitions in the top-level *let* expression. On the other hand, the evaluation machine with this restriction is more efficient. Consider the following example:

let $f = \texttt{initVal}\, f\, 1,$
$\quad h = \lambda x.x + x,$
$\quad g = h\,(f + 2)$
in $\texttt{fire1}\,(f := g)$

The expression $h\,(f + 2)$ would be translated to the restricted syntax

let $x = f + 2$ **in** $h\, x$

If we do not restrict the dependence analysis to top-level definitions, then we would not overwrite the definition of x, because x depends on the dynamic function f. This implies that in the definition of

$h = \lambda x.x + x$

we would evaluate the variable x twice. With the restriction to top-level definitions in the analysis, we overwrite the definition of x after the first evaluation and therefore evaluate it only once.

We assume there is a predicate $upd(x)$ which is *false* if x is a top-level definition depending (transitively but not directly) on a dynamic function. Otherwise $upd(x)$ is required to be *true*. It is very important, that $upd(x)$ is *true* if x depends directly on a dynamic function. This seems to be strange, but consider the following example:

let $f = \texttt{initVal}\, f\, 1$
in $f + f$

If $upd(x)$ would be *false*, then each time accessing variable f, we would create the dynamic function f.

As can be seen in Fig. 3.1, the *variable* rule there applies only if $upd(x)$ is *true*. The other *variable* rules are shown in Fig. 3.5. The first *variable* rule in that figure is similar to that in Fig. 3.1 except that the definition of x in the heap is not modified. The second *variable* rule in Fig. 3.5 considers the case that a variable x should be evaluated to a basic value (different from `dyn0`).

Consider now the case that f in the above expression $f + f$ should be evaluated to a basic value ($v = 1$). According to the second *variable* rule in Fig. 3.5 we first evaluate f to a value in the context $v = 0$. The result y (in our example $\texttt{dyn0}\, f$) of this evaluation is used to overwrite the definition of our dynamic function f. Eventually, we evaluate the intermediate value y to a basic value ($v = 1$) and return the result z with fresh bound variables (value 1 in our example).

This treatment of variables implies a restriction of the usage of `initVal` and `initFun`: They are allowed only in top-level definitions and must not appear as function arguments. In particular, examples like the following definition for g make no sense:

$$\begin{array}{ll}\textbf{let} & f = \texttt{initVal}\, f\, 1, \\ & g = f + \texttt{initVal}\, g\, 2 \\ \textbf{in} & \ldots\end{array}$$

Now our semantics works fine except for the case where the argument for `fire1` is not a top-level definition. Consider the following example:

$$\begin{array}{ll}\textbf{let} & f \quad = \texttt{initVal}\, f\, 1, \\ & \mathit{fire} = \lambda n.\, \lambda r.\, \textbf{if}\ n = 1\, \textbf{then}\, \texttt{fire1}\, r \\ & \qquad\qquad\qquad\qquad \textbf{else}\, \texttt{ioseq}\, (\texttt{fire1}\, r)\, (\mathit{fire}\, (n-1)\, r) \\ \textbf{in} & \mathit{fire}\, 2\, (f := f + 1)\end{array}$$

The intended behavior if $\mathit{fire}\, n\, n$ is to execute n steps of the rule r. Hence, the expected result for f after executing the program is the value 3. However, f is updated twice to 2, because the expression $\mathit{fire}\, 2\, (f := f + 1)$ is replaced by

$$\begin{array}{ll}\textbf{let} & x_1 = \textbf{let}\ x_2 = f + 1\ \textbf{in}\ f := x_2 \\ \textbf{in} & \mathit{fire}\, 2\, x_1\end{array}$$

and x_2 is evaluated once. This problem occurs very rarely. For instance consider the program below similar to the previous one where the function f is in fact updated to 3.

$$\begin{array}{ll}\textbf{let} & f \quad = \texttt{initVal}\, f\, 1, \\ & \mathit{fire} = \lambda n.\, \lambda r.\, \textbf{if}\ n = 1\, \textbf{then}\, \texttt{fire1}\, r \\ & \qquad\qquad\qquad\qquad \textbf{else}\, \texttt{ioseq}\, (\texttt{fire1}\, r)\, (\mathit{fire}\, (n-1)\, r), \\ & \mathit{act} = f := f + 1 \\ \textbf{in} & \mathit{fire}\, 2\, \mathit{act}\end{array}$$

However, we can fix our semantics, such that both programs compute the same result and the same heap as expected. We do this by introducing another primitive function `dynamic`:

$$\mathit{Exp} ::= \ldots \mid \texttt{dynamic}\, x$$

The reduction rule is defined in Fig. 3.5 by the rule *dynamic*. This rule first creates a copy of the expression and then evaluates the copy. We use this expression to define our *fire*1 rule:

$$\begin{array}{ll}\textbf{let} & f \quad\ = \texttt{initVal}\, f\, 1, \\ & \mathit{fire}1 = \lambda r.\, \texttt{fire1}\, (\texttt{dynamic}\, r), \\ & \mathit{fire} \ = \lambda n.\, \lambda r.\, \textbf{if}\ n = 1\, \textbf{then}\, \mathit{fire}1\, r \\ & \qquad\qquad\qquad\qquad \textbf{else}\, \texttt{ioseq}\, (\mathit{fire}1\, r)\, (\mathit{fire}\, (n-1)\, r) \\ \textbf{in} & \mathit{fire}\, 2\, (f := f + 1)\end{array}$$

Note that we provide prelude definitions for the functions $\mathit{fire}1, \mathit{fire}, :=, \ldots$ and therefore there is no need for the user to take care of this problem:

This concludes the extension of the semantics for lazy evaluation and we state the following informal theorem.

Theorem 3.3.1 (Soundness) *With the described restriction for* `initVal` *and* `initFun` *and with the predefined functions* $\mathit{fire}1, :=,$ `ioseq`*, an expression evaluates to its* expected *value with respect to the semantics of lazy evaluation and ASMs.*

To formulize and prove this theorem, one first must translate a program in our syntax (where rules and expressions are represented as expressions) to a syntax where we syntactically distinguish among rules, functions, and dynamic functions. Then one could use the usual ASM semantics [33] for the theorem.

Additionally, one must restrict our syntax, because we know no ASM semantics, where expressions like the following could be represented without transforming the definitions:

$$\begin{array}{ll} \textbf{let} & f \quad = \texttt{initVal}\, f\, 1, \\ & x \quad = \ldots, \\ & a \quad = \textbf{if}\ x = 7\ \textbf{then}\ f\, 2\ \textbf{else}\ f\, 3, \\ & \mathit{act} = a := a + 1 \\ \textbf{in} & \texttt{fire1}\ \mathit{act} \end{array}$$

In fact, this feature in our semantics is useful to introduce an alias for a location which may be more expressive than the original expression.

In summary, proofing this theorem would be really difficult and we leave the proof for further investigation.

3.4 Sequential Execution of Rules

In Section 1.2 we introduced the concept of sequential execution of rules. The sequential execution of two rules is to fire the second rule in the intermediate state established by executing the first rule in the initial state. On the level of update sets, the semantical concept is intuitive. However, it is difficult to integrate the concept in our semantics for lazy evaluation: we have the same problem as in the previous section, namely to represent the sequential execution of two rules as a functional expression.

Let us first consider an example to illustrate why the *seq* concept is problematic in our semantics:

$$\begin{array}{l} \texttt{seq}\ (f := 5)\ \begin{pmatrix} f := 6 \\ h := h + f \end{pmatrix} \\ g := f \end{array}$$

The first of the two rules is the sequential execution of $f := 5$ and the two rules on the righthand-side. In parallel to this sequential execution, we have the update $g := f$. This update needs the initial value of f, but the update $h := h + f$ needs the value of f from the update $f := 5$.

In our semantics, we execute all updates sequentially with the same effect as if they would be executed in parallel. As demonstrated in the example above,

Figure 3.6 Sequential execution

$$\frac{\Gamma \vdash r_1 \Downarrow^{v}_{i,l+1} (\Gamma_1, new_{l+1} \mapsto n_1) \vdash \texttt{rule} \quad (\Gamma_1, old_{l+1} \mapsto n_1) \vdash r_2 \Downarrow^{v}_{i,l+1} (\Gamma_3, new_l \mapsto n, new_{l+1} \mapsto n_2) \vdash \texttt{rule}}{\Gamma \vdash \texttt{seq}\, r_1\, r_2 \Downarrow^{v}_{i,l} (\Gamma_3, new_l \mapsto n \cup (n_1 \oplus n_2), old_{l+1} \mapsto \varnothing) \vdash \texttt{rule}} \quad \text{(Seq)}$$

this implies that we have to work with different values for f depending on where the function is accessed.

We parametrize our heap functions *old* and *new* with an additional level index l and add this index to our previous reduction rules to the reduction symbol $\Downarrow$. The notation $\Downarrow^{v}_{i,l}$ means to reduct in context v, i with level index l. Furthermore, we modify the *read* and *update* rule for dynamic functions. For instance, consider the modified *read* rule for nullary dynamic functions:

$$\frac{\nexists\, k' \colon k < k' \leq l \colon \Gamma = (\Gamma', old_{k'} \mapsto y)}{(\Gamma, old_k(x) \mapsto z) \vdash \texttt{dyn0}\, x \Downarrow^{1}_{0,l} (\Gamma, old_k(x) \mapsto z) \vdash z}$$

The rule implies that for given index l, we use old_k where k is maximal.

When updating a nullary dynamic function in the context of index level l, then we store the update in the function new_l:

$$\frac{\Gamma \vdash lhs \Downarrow^{0}_{i,l} \Gamma_1 \vdash \texttt{dyn0}\, n, \quad \Gamma_1 \vdash rhs \Downarrow^{1}_{i,l} \Gamma_2 \vdash z}{\Gamma \vdash \texttt{update}\, lhs\, rhs \Downarrow^{v}_{i,l} (\Gamma_2, new_l(n) \mapsto z) \vdash \texttt{rule}}$$

The *read* and *update* rules for unary functions are modified similarly and not further discussed here.

To summarize, the previous reduction rules pass the given level index to the sub-reductions. When reading a dynamic function in level l, then we use a maximal k such that old_k is defined and when updating a dynamic function in level l, then we use new_l.

We increment the level index for the evaluation of the arguments of the `seq` construct. The `seq` construct is also a basic expression and takes two rules as its arguments:

$$Exp ::= \ldots \mid \texttt{seq}\, x_1\, x_2$$

The semantics of `seq` is defined in Fig. 3.6, where for given level l we evaluate the rule r_1 in level $l + 1$. This leads to an update set n_1 which is copied to old_{l+1} to evaluate the rule r_2 (also in level $l + 1$) to get the update set n_2. The result of the sequential execution is now a heap where we merge the updates n (we already have for level l) and the updates n_1 and n_2 for level $l + 1$. Since r_2 is executed after r_1, the updates in n_2 overwrite updates in n_1 (denoted by $n_1 \oplus n_2$). The updates n and $n_1 \oplus n_2$ have to be applied in parallel and therefore we build the union of both update sets.

Note that we did not consider inconsistent update sets in our semantics. However, the semantics could be easily extended.

Now, the soundness theorem could be extended for the `seq` construct using the ASM semantics for `seq` as described in Section 1.2.

3.5 Related Work

There are already some tools for executing ASMs. For instance XASM [2] and the ASM Workbench [30]. In the next paragraphs we will briefly describe them and then we will discuss related work with respect to updatable functions.

The ASM Workbench is an interpreter written in ML. Its language syntax is also inspired by ML. There is no sequence operator. However, an advantage of this tool is that it can translate finite ASMs into the language for the SMV model checker for proving properties. The tool provides a graphical user interface to debug an ASM run.

XASM is an ASM compiler written in *C*. Static functions can be defined in *C*, too. The tool has some interesting features; for instance, an ASM with return value can itself be used as a function, or for given grammar rules, the tool generates the corresponding parser and graphically shows the control flow while executing the rules on the syntax tree. In fact, the tool is especially useful with respect to attributed grammars.

In [53], updatable variables are introduced in a simple functional language with lazy evaluation. To guarantee confluence and referential transparency, the author introduces a static criterion based on abstract interpretation which checks that any side-effect (variable update) remains invisible. This criterion is too restrictive for combining ASMs and functional programming.

Another proposal for updatable variables is given in [55, 25] where the authors show that their new calculus is a conservative extension of the classical lambda calculus. This proposal is similar to using monads except that state transformers (used for updatable variables) are introduced in the semantics instead of defined in terms of basic lambda expressions. Therefore, the introduced language has the same disadvantages as described in the beginning of this chapter when discussing monads.

Updatable functions are introduced in [54], where the functional language *Fun* is described. The formal semantics is not completely defined. Therefore, it is difficult to compare the work with our proposal. However, it seems that *Fun* is a functional language with strict evaluation and the mutable variables are similar to references in Standard ML [71, 57].

The above mentioned proposals use the term representation to store variable/function updates instead of using an explicit store as in our work. Furthermore, all updates in these proposals are not applied simultaneously and therefore not suitable for combining ASMs and functional programming. The same applies for Standard ML.

The main contribution of this chapter is the extension of the lambda calculus (with lazy evaluation semantics) for simultaneous function updates. This extended calculus can be used to represent ASMs. With this work, it is straightforward to extend an implementation of a functional system with lazy evaluation. For instance, the next chapter introduces the AsmGofer system which extends Gofer in that way.

Chapter 4

The AsmGofer System

AsmGofer [60] is an advanced Abstract State Machine (ASM) [16] programming system. It is an extension of the functional programming language Gofer [43] which is similar to Haskell [66]. More precisely, AsmGofer introduces a notion of state and parallel updates into TkGofer [67] and TkGofer extends Gofer to support graphical user interfaces.

AsmGofer has been used successfully for several applications. For instance, *Java and the Java Virtual Machine* [64], the *Light Control Case Study* [18], *Simulating UML Statecharts* [23].

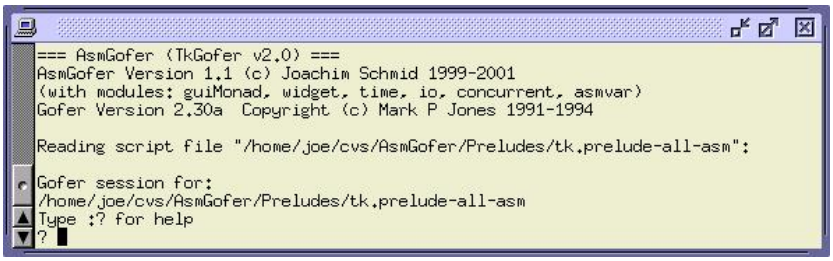
In this chapter we introduce the AsmGofer programming environment. We assume basic knowledge in functional programming, like algebraic data types, functional expressions, and pattern matching. For an introduction into functional programming we refer the reader to [7, 68]. This chapter is not a reference manual for AsmGofer. Unfortunately, up to now, there is no such manual.

In Section 4.1 we briefly explain the Gofer command line interface for loading, editing, and running examples. Section 4.2 and Section 4.3 introduce the notations and constructs for sequential and distributed ASMs in AsmGofer. In Section 4.4 we use sequential ASMs to define the well-known *Game of Life* example. For this example, we show in Section 4.5 how our GUI generator works, and in Section 4.6 we define a customized GUI for the game.

4.1 The Interpreter

Gofer and AsmGofer are interpreters. In this section we describe the command line interface of Gofer and AsmGofer. After starting AsmGofer, the output looks as in Fig. 4.1. At startup the system reads a prelude file where all library functions (*map*, *head*, *tail*, *fst*, ...) are defined. We use the prelude `tk.prelude-all-asm`. The Gofer system is ready for new commands when the `?` prompt occurs. One can now evaluate arbitrary expressions, load and edit files, load projects, and find function and type definitions. In the following subsections we explain these features of the command line.

Figure 4.1 AsmGofer system at startup



4.1.1 Expression Evaluation

Expressions can be evaluated in Gofer using the command line. For instance, typing `5+2` and pressing the *enter* key results in `7` being displayed.

```
? 5+2
7
(3 reductions, 7 cells)
?
```

The Gofer system evaluates the expression and reports the result together with information about the used cells[1] and reduction steps. This information gives an impression about the complexity and size of the evaluation. Another short example is the sum of squares of numbers from one to ten.

```
? sum (map (\x -> x*x) [1..10])
385
(124 reductions, 185 cells)
?
```

When the user enters an expression, Gofer first tries to typecheck it. If the expression has a unique type, then Gofer evaluates the expression, otherwise a type error is reported. For example, consider the wrongly typed expression `5+"Hello"`.

```
? 5+"Hello"
ERROR: Type error in application
*** expression     : 5 + "Hello"
*** term           : 5
*** type           : Int
*** does not match : [Char]
?
```

4.1.2 Dealing with Files

The command line can be used to evaluate expressions, but not to provide definitions. Functions and types can be defined only in files (also called scripts). The command `:l filename.gs` loads the file *filename.gs*. File names do not

[1] A cell is a synonym for head element.

have to end with `.gs`, but this is a usual extension for gofer scripts. The `:l` command removes all definitions of previously loaded files except the definitions in the prelude file. To append definitions contained in another file *file*2.*gs*, the command `:a file2.gs` can be used.

If a file is modified, then `:r` reloads all dependant files. In case there is a syntax or a type error, the interpreter shows the file name and the line number where the error occurred. Typing `:e` opens an editor with the corresponding file at the corresponding line number. The editor used by Gofer can be determined in the environment variables `EDITOR` and `EDITLINE`.

Especially for many files, it is cumbersome to load files with the `:a` command. Therefore Gofer supports so-called project files; each line in a project file contains a file name. The files in *file*.*p* are loaded by `:p file.p` in the order in which they appear in `file.p`. The reload command `:r` works for projects, too. Additionally, an arbitrary file can be edited by typing `:e filename.gs`.

4.1.3 Other Commands

As already mentioned, Gofer typechecks every expression. Furthermore, the interpreter supports a command to print the type of a given expression. Consider the input `:t "Hello"`.

```
? :t "Hello"
"Hello" :: String
?
```

The command `:t` determines the type and prints it to standard output. The operator `::` separates the type information from the expression. In our case the type is *String*. Additional information will be displayed by typing `:i String`.

```
? :i String
-- type constructor
type String = [Char]
?
```

Now Gofer tells us, that *String* is an alias for [*Char*] which is a list of characters.

Gofer remembers the file name and line number for each function and type definition. Entering `:f sum` instructs Gofer to open the editor with the corresponding file and to jump to the position where *sum* is defined.

The command to quit the interpreter is `:q`.

```
? :q
[Leaving Gofer]
joe:  >
```

Gofer supports several other commands. Typing `:?` prints a list of known commands and short descriptions. For further information we refer the reader to the Gofer manual which is included in the AsmGofer distribution [60].

4.2 Sequential ASMs

Encoding ASMs in a purely functional programming language like Gofer seems to be a contradiction on terms, because a pure functional language has no side-effects, whereas each ASM update has a main-effect on the global state. In fact,

AsmGofer is not Gofer with additional definitions in Gofer to support ASMs. Rather, AsmGofer modifies the evaluation machine in the Gofer run-time system to support a notion of state (see Section 3.1 for a discussion about this topic). On the other hand, we do not change the Gofer syntax and therefore we have to represent the ASM features as expressions. We provide special functions and operators to define dynamic functions and to perform updates, as we are going to explain in this section.

4.2.1 Nullary Dynamic Functions

Since we do not provide any special ASM syntax, we have to represent a dynamic function as an ordinary functional term. For this purpose, the prelude contains a function *initVal* with the following signature (see below for an explanation of *Eq*):

$$initVal :: Eq\ a \Rightarrow String \rightarrow a \rightarrow a$$

With this function we can create a 0-ary dynamic function f by the following definition:

$$f = initVal\ \texttt{"name"}\ init$$

The first argument `"name"` is a name for the dynamic function which is used only in error messages. Due to technical reasons in the Gofer implementation, we cannot access the function name f as given by the lefthand-side expression of the function definition. The second argument *init* is the initial value for f. The *initVal* function is defined in such a way, that the return type of the function is equal to the type of the initial value, as can be seen from the signature declaration above.

In Gofer there is no need to define function signatures for function definitions. Usually, Gofer can deduce the types. However, we suggest to define the signatures anyway, because this enhances the readability of type error reports by Gofer. To improve readability we further suggest to write signatures for dynamic functions with the type alias *Dynamic*:

type $Dynamic\ t = t$

Note that this definition of *Dynamic* implies that there is *no* semantic difference between a type A and $Dynamic\ A$. *Dynamic* is added only for better readability. A declaration of a dynamic function may then look as follows:

$f :: Dynamic\ Int$
$f = initVal\ \texttt{"f"}\ 0$

The notation $Eq\ a \Rightarrow \ldots$ in the signature for *initVal* means that the type a must be an instance of type class *Eq*. We refer the reader to [38] for a discussion of type classes. In the following we use the term *class* as an alias for type class. Being an instance of *Eq* implies that the equality operator == is defined for that type. AsmGofer needs this operator for the consistency check of updates, namely to determine whether two values are equal. All basic types in AsmGofer are already defined as an instance of class *Eq* and for all user-defined types, the user has to provide the corresponding definition. Alternatively, one can define

a type to be an instance of the class *AsmTerm* (defined in the prelude), which makes it an instance of class *Eq* using the type class features of Gofer. For more information about type classes in Gofer, we refer the reader to [42].

data *MyType* = ...

instance *AsmTerm MyType*

For AsmGofer, a dynamic function behaves similarly to other functions. For instance, we can enter *f* in the command line to evaluate the function:

```
? f
0
(5 reductions, 19 cells)
?
```

In our definition of *f* we defined zero as the initial value for the dynamic function. It is also possible to use the special predefined expression *asmDefault*, which corresponds to an undefined value for each type, defined as an instance of class *AsmTerm*:

asmDefault :: *AsmTerm a* ⇒ *a*

We can now use this undefined value to define the dynamic function *f*:

f :: *Dynamic Int*
f = *initVal* `"f"` *asmDefault*

This *undefined* value is not an implementation of the value `undef` of the *Lipari Guide* [33]. In particular, AsmGofer can not use this undefined value in computations. Therefore, whenever an expression evaluates to *asmDefault*, AsmGofer stops the computation and reports an error message. Note that Gofer uses lazy evaluation [51] which implies that expressions are evaluated only when necessary.

```
? f
(5 reductions, 18 cells)
ERROR: evaluation of undefined 'asmDefault'
*** dynamic function: "f"
?
```

However, it is possible to specify for any type an own *undefined* value. For example we could define 0 as the undefined value for expressions of type *Int*, as in the following instance definition.

instance *AsmTerm Int* **where**
  asmDefault = 0

This definition implies that 0 and *asmDefault* are treated as equal. This will be become more interesting for unary dynamic functions as discussed in the next subsection.

The above kind of *undefined* is more flexible than the *undefined* in the *Lipari Guide*, where *undefined* is treated like an ordinary value which can be used in computations. In AsmGofer one can use the predefined *asmDefault* expression where a computation is abrupted whenever this expression occurs, but one can also define an element which should be used instead of *undefined*.

4.2.2 Unary Dynamic Functions

We provide a function *initAssocs* which can be used to define unary dynamic functions.

$$initAssocs :: (AsmOrd\ a, Eq\ b) \Rightarrow String \rightarrow [(a, b)] \rightarrow Dynamic(a \rightarrow b)$$

The first argument is the name for the dynamic function, which is used only in error messages. The second argument is an initialization list. If this list is empty, then the dynamic function is undefined (in the sense above) for each argument. In the type signature for *initAssocs* we can see that type a must be an instance of class *AsmOrd* and b an instance of class *Eq*. Requiring b to be an instance of *Eq* allows one to determine whether two values are equal when checking the consistency of updates. Requiring a to be an instance of *AsmOrd* is used to compare two arguments. The equality operator would be sufficient, but we can implement unary dynamic functions more efficiently using binary search, if there is an ordering on the argument type. The class *AsmOrd* is defined as follows.

class *AsmOrd a* **where**
  $asmCompare :: a \rightarrow a \rightarrow Int$

The function *asmCompare* returns for two arguments either -1, 0, or 1 depending on whether the first argument is less than, equal to, or greater than the second argument. If we define a type as an instance of *AsmTerm*, then it automatically becomes an instance of class *AsmOrd*.

Similarly to nullary dynamic functions we can introduce unary dynamic functions, but using *initAssocs* instead of *initVal*.

$g :: Dynamic(Int \rightarrow Int)$
$g = initAssocs$ `"g"` $[(0, 1), (1, 1)]$

The function g can be used like other unary functions, except when the function should be evaluated for an argument which is not in the domain of the function. In that case, AsmGofer uses the expression *asmDefault* already introduced above.

```
? g 1
1
(6 reductions, 16 cells)
? g 2
(4 reductions, 15 cells)
ERROR: evaluation of undefined 'asmDefault'
*** dynamic function: "g"
?
```

Additionally, AsmGofer supports some other functions to determine the domain and the range of unary dynamic functions, to check whether an expression is in the domain of a function, and to compute the current association list (function represented as a finite mapping).

$dom :: Ord\ a \Rightarrow Dynamic(a \rightarrow b) \rightarrow \{a\}$
$ran :: Ord\ b \Rightarrow Dynamic(a \rightarrow b) \rightarrow \{b\}$
$inDom :: a \rightarrow Dynamic(a \rightarrow b) \rightarrow Bool$
$assocs :: Dynamic(a \rightarrow b) \rightarrow [(a, b)]$

The predefined Gofer class *Ord* defines the operators $<, \leq, >, \geq$. The type $\{a\}$ is the type corresponding to the power set of type a similar to the list type $[a]$ except that there are no duplicate values. The requirements on class *Ord* are used to sort the expressions in a set. The domain of a dynamic function only contains those expressions which are mapped to a value different from the *undefined* element for the corresponding type.

4.2.3 Update Operator

Up to now we introduced nullary and unary dynamic functions. Now the question arises how we can update dynamic functions. As usual in ASMs we provide the := operator.

$(:=) :: AsmTerm\ a \Rightarrow a \rightarrow a \rightarrow Rule\ ()$

The operator takes two arguments of the same type (which must be an instance of *AsmTerm*) and returns something of the special type *Rule* (). We use this type to represent rules. Additionally, we use the *do notation* for monads [66] in Gofer to denote parallel execution of rules. The *do notation* and monads [69] are not described in this chapter, because this would explode this introduction. Roughly spoken, the *do notation* for rules in AsmGofer can be viewed as taking a set of rules and combining them to one rule as in the example below for *someUpdate*.

```
someUpdate :: Rule ()
someUpdate = do
  f   := 5
  g 2 := 7 + f
```

The other basic rule is the *skip* rule which has the empty set as update set.

```
skip :: Rule ()
```

This *skip* rule is especially useful in *if-then-else* expressions when no else part is needed.

```
someOtherUpdate :: Rule ()
someOtherUpdate =
  if f == 2 then
    g 2 := 7 + f
  else skip
```

4.2.4 N-ary Dynamic Functions

We do not provide syntax for dynamic functions with arity greater than one. However, such dynamic function can be represented as a unary dynamic function by using a tuple for the arguments. Additionally, one can define an auxiliary dynamic function as illustrated in the following example for a 2-ary dynamic function g:

```
g_aux :: Dynamic((Int, String) -> String)
g_aux = initAssocs "g" some_init

g :: Int -> String -> String
g i s = g_aux(i, s)
```

The function g is a 2-ary function. We can use g to access the values of g_aux. On the other hand, we can also use g to update the dynamic function g_aux, because $g\,i\,s$ and $g\_aux(i, s)$ are treated equally by AsmGofer.

```
my_updates :: Rule ()
my_updates = do
  g 5 "great" := "hello"
  g 7 "other" := g 3 "strange"
```

4.2.5 Execution of Rules

In the previous subsections we defined dynamic functions and updates to them. The question is how to execute the updates and especially, what to do with the type *Rule* ()? Expressions of type *Rule* () correspond to rules and have a side-effect on the global state. Gofer supports *IO actions* [45] of type *IO* (), which are used to perform input-output-operations like printing a string on standard output. Printing a string is a side-effect. In Gofer, this side-effect is implemented by a primitive function which on evaluation prints the corresponding string on standard output. The monad ensures that the function must be evaluated in order to proceed. We use the same technique to define primitive functions having a side-effect on a global state to implement dynamic functions. However, to distinguish in the type system between IO actions and rules, we use an abstract type *Rule* and we provide the following functions to transform expressions of type *Rule* () into IO actions which can be executed by the Gofer interpreter.

```
fire      :: Int → Rule () → IO ()
fire1     :: Rule () → IO ()
fireWhile :: Bool → Rule () → IO ()
fireUntil :: Bool → Rule () → IO ()
fixpoint  :: Rule () → IO ()
```

The first function *fire* takes two arguments. The first argument is the number of steps to execute the rule specified by the second argument.

```
? fire 2 someUpdate
2 steps
(102 reductions, 228 cells)
?
```

The *fire*1 function is a specialization of *fire* where the number of steps to execute is fixed to 1. The *fireWhile* and *fireUntil* functions take a condition and a rule as arguments and fire while or until the condition holds. The *fixpoint* function fires its argument as long as the resulting update set is not empty.

Execution of rules is possible only if the corresponding update set is consistent. An update set is inconsistent if it contains updates to assign two different values to the same location.

```
? fire1 (do f := 1; f := 2)
(22 reductions, 60 cells)
ERROR: inconsistent update
*** dynamic function : "f"
*** expression (new) : 2
*** expression (old) : 1
?
```

4.2.6 Rule Combinators

The *Lipari Guide* [33] introduces several rules like *import*, *extend*, *choose*, and *var over*. We have implemented *forall*, *choose*, and *create* where *forall* corresponds to *var over* and *create* to the *import* rule in the *Lipari Guide*. The result type of the rule combinators in this subsection is always *Rule* ().

Our *forall* rule takes a range constraint similar to list comprehension in Gofer and a rule to execute.

forall $i \leftarrow dom(g)$ **do**
  $g\ i := g\ i + 1$

In this example the rule body is executed for each i in the domain of g. It is also possible to loop over several variables.

forall $i \leftarrow dom(g), j \leftarrow \{1..10\}$ **do**
  $h(i, j) := g\ i + j$

On the other hand, the *choose* rule below chooses one i in the domain of g and executes the body. If the domain of g is empty, then the rule is equivalent to *skip*.

choose $i \leftarrow dom(g)$ **do**
  $f := f + g\ i$

We provide an alternative *choose* rule where we can determine with an *ifnone* clause what should happen when the range constraint is empty.

chooseIfNone $i \leftarrow dom(g)$ **do**
  $f := f + g\ i$
ifnone
  $f := 0$

In this example the update $f := 0$ is performed if the domain of g is empty, otherwise an element in the domain is chosen as in the *choose* rule above.

The *Lipari Guide* [33] supports an *import* rule which takes anonymous elements from a special universe *reserve*. Imported elements are no longer in that universe and no element of *reserve* is an element of any other universe. In AsmGofer we want to support something similar. However, it is difficult to implement the semantics of the *import* rule according to the *Lipari Guide* in a functional language with algebraic data types. Therefore, we provide a *create* rule which can be used to deal with anonymous elements. The rule only ensures that a "created" expression was never created previously by a *create* rule, and if two *create* rules are executed in parallel, then both elements are different. Consider the following example for creating heap references.

create *ref* **do**
  $heap(ref) := Object(..)$

The *create* rule works for expressions of type *Int*. When the necessity arises to use the *create* rule for a type different from *Int*, then we have to define this type as an instance of the following type class *Create*.

class *Create a* **where**
  $createElem :: Int \rightarrow a$

When defining a type a as an instance of class *Create* we must provide an implementation for the *createElem* function. This function expects an integer value as its argument and transforms it to an expression of the corresponding type a. It is important, that the definition of *createElem* is an injective function. Otherwise the *create* rule does not "create" always different elements of the corresponding types.

instance *Create MyType* **where**
  createElem i = ...

Sometimes it is useful to choose one rule among a set of rules. For that reason we provide the *choose among* rule.

choose among
  $f := f + 1$
  $f := f + 2$
  $f := f + 3$

In Section 1.2 we introduced the concepts of sequential execution and iteration of rules. Both concepts are implemented in AsmGofer by the functions *seq* and *iterate*.

seq :: *Rule* () → *Rule* () → *Rule* ()
iterate :: *Rule* () → *Rule* ()

The result of *seq* is the sequential execution of the argument rules. The second rule is executed in the intermediate state established by the first rule. The *iterate* construct is similar to the *fixpoint* function in the previous subsection, except that the result is a rule and not an IO action. Note also, that intermediate states are not visible to other rules. Both constructs are atomic and executed in one step.

4.3 Distributed ASMs

In the previous section we described constructs for sequential ASMs. In the *Lipari Guide* [33] there is also a definition for distributed (or multi agent) ASMs. In sequential ASMs there is one agent firing always the same set of rules. In a distributed ASM there are several agents firing rules. Furthermore, the set of active agents might be dynamic.

In our implementation of multi agent ASMs we can define for each agent a rule to execute. Such a rule gets as its first argument (*self* in the example in Fig. 4.2) the agent which executes the rule. For instance, consider the definitions for the well known *Dining Philosophers* problem in Fig. 4.2 where each philosopher is an agent. In the figure the functions *fork* and *mode* are dynamic functions parametrized over a philosopher. It is important that we parametrize the *Up* constructor over a philosopher, too. Otherwise we do not know which fork is used by which philosopher. The dynamic function *phils* assigns to a philosopher the *exec* rule. In our case each philosopher executes the same rule.

We provide a special function *multi* with the following signature to execute agents.

multi :: *Dynamic*(*a* → *AgentRule a*) → *Rule* ()

Figure 4.2 Dining philosophers

```
type AgentRule a = a → Rule ()
type Philosopher = Int
data Fork        = Up(Philosopher) | Down
data Mode        = Think | Eat

instance AsmTerm Fork where
  asmDefault = Down

instance AsmTerm Mode where
  asmDefault = Think

fork :: Dynamic(Philosopher → Fork)
fork = initAssocs "fork" []

mode :: Dynamic(Philosopher → Mode)
mode = initAssocs "mode" []

phils :: Dynamic(Philosopher → AgentRule Philosopher)
phils = initAssocs "phils" [(ph1, exec), (ph2, exec), ...]

exec :: AgentRule Philosopher
exec self =
  if mode(self) == Think ∧ lfork == Down ∧ rfork == Down then do
    lfork      := Up(self)
    rfork      := Up(self)
    mode(self) := Eat
  else if mode(self) == Eat then do
    lfork      := Down
    rfork      := Down
    mode(self) := Think
  else skip
where lfork = fork(self)
      rfork = fork(right)
      right = (self + 1) `mod` card(dom(phils))
```

This function takes as its argument a dynamic function like the function *phils* in Fig. 4.2. The function result is a rule. The implementation of *multi* chooses non-deterministically a subset of the domain of *phils* and executes in parallel for each element in this subset the corresponding rule. This could be described by the following pseudo rule.

```
multi actions =
  forall act ← some_subset(dom(actions)) do
    (actions act)(act)
```

Note that not necessarily all agents execute the same rule as in our example. It is important that *multi* never chooses a subset which leads to an inconsistent update. This is useful in particular for our example, because the rule

```
multi phils
```

never chooses a set of philosophers where a fork is shared by two philosophers, because then we would get an inconsistent update for the dynamic function *fork*.

4.4 An Example: Game of Life

In this section we briefly introduce Conway's well-known *Game of Life* and then we show how to formulate the static and dynamic semantics in AsmGofer. Figure 4.3 shows a typical pattern of this game. The game consists of a $n \times m$ matrix; each cell is either alive or dead. The rules for survival, death and birth are as follows (see [70]):

- *Survival:* each living cell with two or three alive neighbors survives until the next generation.
- *Death:* each living cell with less than two or more than three neighbors dies.
- *Birth:* each dead cell with exactly three living neighbors becomes alive.

In the following two subsections we illustrate the use of AsmGofer by means of the *Game of Life* example. We define the static and dynamic semantics.

4.4.1 Static Semantics

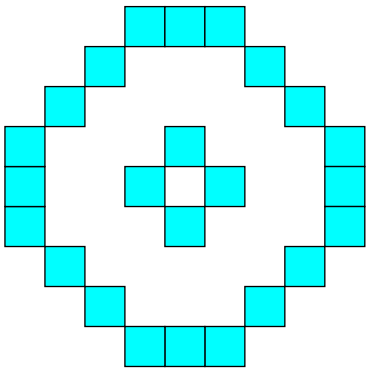
The definitions in this subsection are ordinary Gofer definitions. In the next subsection when presenting the dynamic semantics we use constructs which are available only in AsmGofer.

Each cell in the matrix is either alive or dead and therefore we define a type *State* consisting of the two constructors *Dead* and *Alive*.

```
data State = Dead | Alive
```

Both constructors can be viewed as nullary functions creating elements of type *State*. Note that in Gofer, type names and constructor names must always start with an upper case letter. Functions on the other hand with a lower case letter.

Figure 4.3 Conway's Game of Life



For the representation of cells we use pairs of integer values and we define a type *Cell* as an alias for them.

type $Cell = (Int, Int)$

Further, we define a nullary function cN to denote the number of columns and rows. In order to loop later through all possible cells, we define a function computing such a list. In the definition below we use the concept of list comprehension. The function result is a list of pairs (i, j) where i and j range from 0 to $cN-1$. The order in the list is as follows: $[(0,0),(0,1),\ldots,(0,cN-1),(1,0),\ldots]$.

$$cN :: Int$$
$$cN = 8$$

$$cells :: [Cell]$$
$$cells = [(i,j) \mid i \leftarrow [0..cN-1], j \leftarrow [0..cN-1]]$$

In the rules for *Game of Life* we need for each cell the number of alive neighbors. A cell has at most 8 neighbors. The following definition computes a list of tuples (i', j') where (i', j') is a neighbor different from (i, j) and a valid position in the game.

$$neighbors :: Cell \to [Cell]$$
$$neighbors(i,j) = [(i',j') \mid i' \leftarrow [i-1..i+1], j' \leftarrow [j-1..j+1], (i,j) \neq (i',j'), valid\ i', valid\ j']$$
where $valid\ i = i \in [0..cN-1]$

With the definition of *neighbors*, we can define the number of alive neighbors. This is the length of the list *neighbors* restricted to those elements which are alive.

$$aliveNeighbors :: Cell \to Int$$
$$aliveNeighbors\ cell = length([c \mid c \leftarrow neighbors\ cell, status\ c == Alive])$$

For the time being, let us assume there is a function *status* with the signature below. In the next subsection we will define *status* as a dynamic function, since

it may change its value during execution.

```
status :: Cell → State
```

4.4.2 Dynamic Semantics

In this subsection we describe the dynamic semantics of the game in terms of ASMs. To do this, we first have to store for each cell whether it is alive or dead. Therefore we use a unary dynamic function *status* initialized with state *Dead* for each cell.

```
status :: Dynamic(Cell → State)
status = initAssocs "status" [(c, Dead) | c ← cells]

instance AsmTerm State
```

Now we can define two rules *letDie* and *letLive* for the behavior of a cell with n alive neighbors. Both rules correspond to the last two rules (*Death* and *Birth*) at the beginning of this section. The first rule *Survival* is automatically established by ASM semantics. Since everything must be an expression in Gofer (as already stated), we can not omit the *else* branch in the following definitions.

```
letDie :: Cell → Int → Rule ()
letDie cell n =
  if status cell == Alive ∧ (n < 2 ∨ n > 3) then
    status cell := Dead
  else skip

letLive :: Cell → Int → Rule ()
letLive cell n =
  if status cell == Dead ∧ n == 3 then
    status cell := Alive
  else skip
```

It remains to execute both rules for all possible cells. For this purpose we use the *forall* rule of AsmGofer.

```
gameOfLife :: Rule ()
gameOfLife =
  forall cell ← cells do
    let n = aliveNeighbors cell
    letDie cell n
    letLive cell n
```

These definitions are sufficient for the dynamic behavior of the game. We can use the command line to validate our specification. For instance, consider following scenario:

```
? status (2,3)
Dead
(66 reductions, 175 cells)
? fire1 gameOfLife
```

Figure 4.4 Configuration for Game of Life

```
@FUNS
status
@TERMS
status (2,3)
@RULES
gameOfLife
initField
@CONDS
False
@FILES
output          guiMain.gs
@GUI
main            gmain
title           Game of Life
@DEFAULT
history
update
```

```
0 steps
(126277 reductions, 259144 cells, 2 garbage collections)
? status (2,3)
Dead
(66 reductions, 175 cells)
? fire1 (do status (2,3) := Alive; status (3,4) := Alive)
1 steps
(131 reductions, 352 cells)
? status (2,3)
Alive
(66 reductions, 176 cells)
? assocs status
[((0,0),Dead), ((0,1),Dead), ...,  ((2,3),Alive),...]
(386 reductions, 2182 cells)
?
```

AsmGofer reports zero execution steps in the above output for the first *fire*1 expression, because no update was performed.

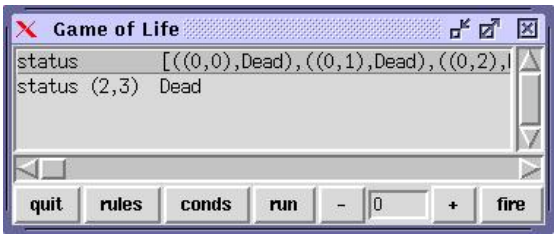
Obviously, this kind of executing rules and debugging is very time consuming. Fortunately, AsmGofer extends TkGofer whose features can be used to define graphical user interfaces as we will describe in the next two sections.

4.5 Automatic GUI Generation

A useful feature of AsmGofer is the automatic generation of a GUI. Obviously, the generated GUI is independent of the application domain, but it can be used to debug and validate a specification at early stages.

The GUI generator is written in AsmGofer itself and reads some configuration information from a file called `gui.config` which must be located in the current working directory. The configuration file provides information about

Figure 4.5 Generated GUI for Game of Life



dynamic functions to display, expressions to display, rules the user may select in the GUI to execute, and some conditions for iterative execution of rules. With this information, the generator creates a new file containing the corresponding GUI definitions.

Figure 4.4 shows a configuration file for our example. The file is divided into several sections which we are now going to explain. Section `FUNS` contains dynamic function names which should be displayed in the GUI. Additionally the GUI displays expressions which are defined in the `TERMS` section. In our case we display only the expression *status* $(2,3)$. The GUI generator creates a list box where the user can select a rule to execute. All names in the `RULES` section are listed in that box. Obviously these names must be of type *Rule* $()$. We add the following rule *initField* to our definitions for the game to initialize the matrix.

```
initField :: Rule ()
initField = do
  forall c ← cells do
    if c ∈ init then
      status c := Alive
    else
      status c := Dead
where init = [(3,4),(4,4),(5,4),(4,3),(5,5)]
```

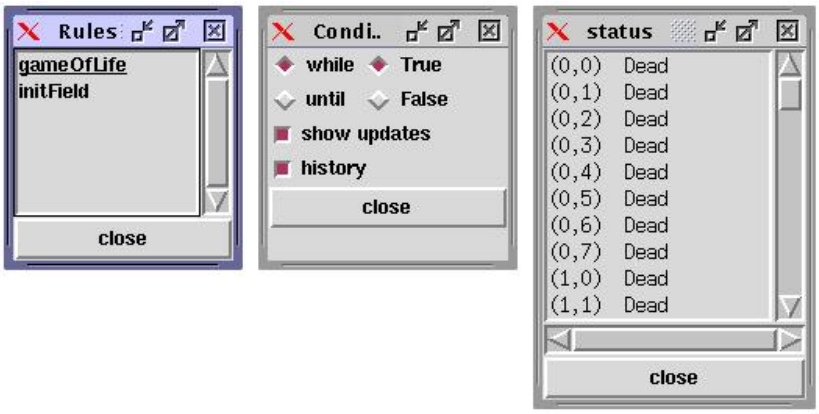
Selected rules in the list box can be executed step by step and while or until a certain condition holds. The conditions which can be selected are defined in the `CONDS` section. All expressions in this section must be of type *Bool*.

The *output* entry in the section `FILES` denotes the file in which the generator should create the corresponding GUI definitions. Additionally, the *main* and *title* entries define the function name to start the resulting GUI and the title for the main window, respectively. In our case we can run the generated GUI with the function *gmain*.

In the `DEFAULT` section we can specify whether we want a history for the execution steps to move forward and backwards and whether the GUI should be updated during execution of a rule while or until a condition holds.

With the configuration file in Fig. 4.4 we can type `genGUI` in the unix shell. This command calls the GUI generator, reads the information in the configuration file and writes the file `guiMain.gs`. If the file *game.gs* contains the AsmGofer definitions introduced in the previous chapter, then we define the following project file `game.p` for Gofer.

Figure 4.6 Rule and condition window, Dynamic function *status*




```
game.gs
guiMain.gs
```

We now load this project file by typing `:p game.p` in the Gofer command line. The expression *gmain* starts the generated GUI and the main window in Fig. 4.5 appears. Clicking on the `rules` button displays the list box containing the rules specified in the `RULES` section of the configuration file. The same applies for the box displaying the conditions. Figure 4.6 shows both windows. Note that the condition *True* is always present.

The content of dynamic functions is displayed in the main window. Double clicking on such a dynamic function opens a new window which displays only the values for that dynamic function. Figure 4.6 also shows this window for the dynamic function *status*.

The `quit` button in Fig. 4.5 is self explaining. Pressing the `run` button executes the rule selected in the rule window while or until the selected condition in the condition window holds. If the check button `show updates` is enabled, then the GUI will be updated in each step during iterative execution with `run`; otherwise only after the iterative execution.

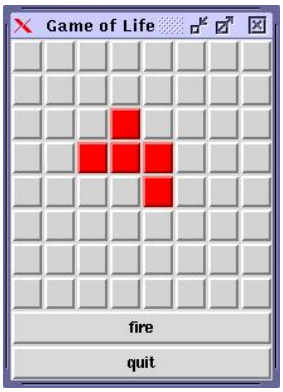
If the history button is enabled, then AsmGofer stores the complete history of execution steps. The buttons `+` and `-` move backwards and forwards in the history. Finally, the `fire` button executes one step of the selected rule.

4.6 User defined GUI

This section describes how to fire rules from a GUI. For this reason, we define a graphical user interface for Conway's *Game of Life* using the features of TkGofer. In this section, we assume the reader is familiar with TkGofer [67].

As already mentioned, the function *fire*1 transforms a rule into an IO action and is of type $Rule\,() \rightarrow IO\,()$. On the other hand, TkGofer provides a function *liftIO* transforming an IO action into a GUI action. This implies that the following definition for *oneStep* is a GUI action which first executes one step of the rule *gameOfLife* and then updates each field in the list *fields* containing

Figure 4.7 Graphical user interface



pairs of buttons and cells. GUI actions are always executed sequentially from top to bottom (see [67] for more information).

```
oneStep :: Fields → GUI ()
oneStep(fields) = do
  liftIO (fire1 gameOfLife)
  seqs [cset f (background(color c)) | (f, c) ← fields]

type Field  = (Button, Cell)
type Fields = [Field]
```

We are now going to define the remaining functions for the GUI (see Fig. 4.7) and we start with a function yielding a color to represent a cell depending on its status.

```
color :: Cell → String
color cell = if status cell == Dead then "lightgrey" else "red"
```

Since we do not want to initialize the dynamic function *status* by entering commands on the command line we define each field in the matrix in Fig. 4.7 in such a way, that the status of the corresponding cell toggles whenever we click on the field. Therefore we define a *toggle* action similarly to the *oneStep* action above. It first switches the status for the given cell and then updates the background color of the corresponding button in the GUI.

```
toggle :: Field → GUI ()
toggle(b, cell) = do
  liftIO (fire1(if status cell == Dead then
                  status cell := Alive
                else
                  status cell := Dead))
  cset b (background(color cell))
```

A Gofer program is usually started by a function called *main*. Hence we define *main* as creating the main window and the elements as shown in Fig. 4.7.

All functions used in this definition are already introduced or described in [67] except *controlWindow*.

```
main :: IO ()
main = start $ do
  w      ← window [title "Game of Life"]
  q      ← button [text "quit", command quit] w
  bfire  ← button [text "fire"] w
  fields ← binds [button[background(color c), font "courier"] w
                  | c ← cells]
  let fields' = zip fields cells
  seqs [cset f (command(toggle(f, c))) | (f, c) ← fields']
  cset bfire (command(oneStep fields'))
  pack (matrix cN fields ^^ fillX bfire ^^ fillX q)
  controlWindow(fields')
```

The function *controlWindow* creates an additional window (see Fig. 4.8) to control the execution. This window contains a `run` button to start automatic execution (iterative execution of *oneStep*) and a `cancel` button to stop it.

```
controlWindow :: Fields → GUI ()
controlWindow(fields) = do
  w  ← window [title "speed control"]
  t  ← timer [initValue 1000, active False, command(oneStep(fields))]
  bc ← button [text "cancel", active False] w
  br ← button [text "run"] w
  s  ← hscale [scaleRange (0, 3000), tickInterval 1000,
               text "delay in ms", height 200, initValue 1000] w
  pack (flexible(flexible s^^bc << br))
  cset s  (command( do v ← getValue s; setValue t v))
  cset br (command(animation(t, s, bc, br)))
  cset bc (command( do cset t (active False)
                       cset br (active True)
                       cset bc (active False)))
```

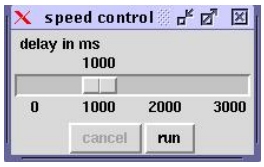
The definition creates a new window with a timer object, two buttons `cancel` and `run`, and a scaler. Whenever the timer expires, the *oneStep* action is executed. In our definition, the timer expires every second (assuming the timer is active).

When clicking on the `run` button, the action *animation* is executed which disables the `run` button, enables the `cancel` button, and activates the timer. This implies that then *oneStep* is executed every second.

```
animation :: (Timer, Scale, Button, Button) → GUI ()
animation(t, s, bc, br) = do
  cset br (active False)
  cset bc (active True)
  cset t  (active True)
```

Loading the source code (downloadable at [60]) in AsmGofer and typing *main* starts the GUI. We can initialize the dynamic function *status* by clicking

Figure 4.8 Control execution



on the corresponding fields to establish the glider pattern as shown in Fig. 4.7. The `fire` button in the main window executes one step of the *gameOfLife* rule. On the other hand, the `run` button in the control window executes *gameOfLife* iteratively. Pressing the `quit` button terminates the GUI and the interpreter is ready for further commands.

Chapter 5

Applications

This chapter describes some applications and case studies where the introduced refinement and implementation techniques were applied. Additionally, Section 5.4 describes the FALKO application where ASMs are compiled to *C++*.

5.1 The Light Control System

The *Light Control System* was a case study of the Dagstuhl seminar about *Requirements Capture, Documentation, and Validation* [15]. For this case study, E. Börger, E. Riccobene, and J. Schmid used ASMs to capture the informally stated requirements. The following paragraph is the abstract of the published paper [18].

We show how to capture informally stated requirements by an ASM (Abstract State Machine) model. The model removes the inconsistencies, ambiguities and incomplete parts in the informal description without adding details which belong to the subsequent software design. Such models are formulated using application-domain-oriented terminology and standard software engineering notation and bridge the gap between the application-domain and the system design views of the underlying problem in a reliable and practical way, avoiding any formal overhead. The basic model architecture reflects the three main system parts, namely for the manual and automatic light control and for handling failures and services. We refine the ground model into a version that is executable by AsmGofer and can be used for high-level simulation, test and debugging purposes.

5.2 Java and the Java Virtual Machine

The following paragraphs are part of the introduction of *Java and the Java Virtual Machine* [64] written by R. F. Stärk, J. Schmid, and E. Börger.

The book provides a structured and high-level description, together with a mathematical and an experimental analysis, of Java and of the Java Virtual Machine (JVM), including the standard compilation of Java programs to JVM code and the security critical bytecode verifier component of the JVM. The description is structured into modules (language layers and machine components), and its abstract character implies that it is truly platform-independent.

It comes with a natural refinement to executable machines on which code can be tested, exploiting in particular the potential of model-based high-level testing. The analysis brings to light in what sense, and under which conditions, legal Java programs can be guaranteed to be correctly compiled, to successfully pass the bytecode verifier, and to be executed on the JVM correctly, i.e., faithfully reflecting the Java semantics and without violating any run-time checks. The method we develop for this purpose, using Abstract State Machines which one may view as code written in an abstract programming language, can be applied to other virtual machines and to other programming languages as well.

We provide experimental support for our analysis, namely, by the validation of the models in their AsmGofer executable form. Since the executable AsmGofer specifications are mechanically transformed into the LaTeX code for the numerous models which appear in the text, the correspondence between these specifications is no longer disrupted by any manual translation. AsmGofer is an ASM programming system, extending TkGofer to execute ASMs which come with Haskell definable external functions. It provides a step-by-step execution of ASMs, in particular of Java/JVM programs on our Java/JVM machines, with GUIs to support debugging. The appendix which accompanies the book contains an introduction to the three graphical AsmGofer user interfaces: for Java, for the compiler from Java to bytecode, and for the JVM. The Java GUI offers debugger features and can be used to observe the behavior of Java programs during their execution. As a result, the reader can run experiments by executing Java programs on our Java machine, compiling them to bytecode and executing that bytecode on our JVM machine.

5.3 Hardware Verification

The introduced specification language and verification technique in Chapter 2 was experimentally used for two projects at Siemens Corporate Technology. Since these are industrial and thus highly confidential projects, we can only touch the experiments.

We now report about the first experiment. When the experiment started, the verbal specification was already given. Almost at the same time, for some components in the specification we built abstract models and the hardware designers started encoding in VHDL based on the verbal specification. Since the specification could be interpreted differently, some of the abstract models behaved differently from the models designed by the hardware experts. However, the introduced verification technique could be applied to show that one small component was buggy. For this component, the specification was exact enough, such that the abstract model and the concrete model behave similarly. We formalized a high-level property (about the composition of the studied components) which was valid in the abstract composition but the equality check of Theorem 2.3.2 failed due to a bug in the concrete implementation of that small component.

In the second experiment the VHDL code for some components was already given. The problem was, that important properties could not be proved in the composed model, because this model was too complex. Therefore, the introduced specification language was used to build abstract models with respect to the property to be proved. The composed abstract model was *simple* enough

such that the desired properties could be proven. However, the introduced verification technique was not applied, since it is not yet supported enough due to insufficient tool support. On the other hand, this experiment shows, that abstract models can be built which are *simple* enough but contain the functionality needed for interesting properties.

For further experiments it would be important that the abstract models are developed during the specification phase and then used as a pattern for the final implementation. Otherwise, if somebody (not familiar with the designed hardware, e.g.) tries to build an abstract model and hardware designers build independently their VHDL implementation, then probably the abstract models behave differently from the concrete models and it would be difficult to apply the introduced verification technique.

5.4 FALKO

FALKO [17] is a software system for railway simulation. The software consists of three components, namely the train supervision, interlocking system, and the process simulator. The first two components are manually encoded in *C++* and the process simulator is designed using ASM-SL [29, 30]. The specification is detailed enough such that it can be executed using the ASM-Workbench [30] with an additional oracle for the external functions in the specification.

The ASM-Workbench was used to debug the specification for small test scenarios. We compiled this specification into *C++* using our compilation scheme introduced in [61]. The generated code is used successfully since January 1999 in the final product release. Until now (April 2001) no bugs in the compilation scheme have been discovered and only two *specification bugs* occurred. When the specification bugs were discovered, the team which implemented the other components fixed the bugs directly in the generated code, because they were not familiar with the ASM specification and the provided tool environment. They also introduced a new feature in the compiled code. This illustrates that the generated code is readable enough so that people not familiar with the compilation scheme can fix and extend the produced code. Meanwhile, the bugs have been fixed in the specification, the new feature was introduced, and the specification was recompiled into *C++* to prevent inconsistencies between the specification and the code.

For more information about the FALKO specification and the generated code like size and effort, we refer the reader to [17].

Conclusions and Outlook

This thesis introduced several structuring and composition principles for Abstract State Machines. For instance, sequential execution of machines, iteration of machines, parametrized machines, or components. These concepts can be used to structure single machines (by using parametrized machines, sequential execution, iteration, e.g.) as well as to divide large specifications into simpler parts by using the introduced component concept.

Most of these concepts are applied in several applications and some of them are implemented in AsmGofer. In the following paragraphs we will summarize the chapters of this thesis.

Submachines. In the Lipari Guide [33] ASMs are defined inductively mainly by guarded function updates. In particular, there is no concept of named machines and no concept of sequential execution; essential features for structuring machines. In this work, we introduced parametrized machines (possibly recursive), sequential execution of two machines, fixpoint iteration of a machine, local state inside machines, machines with return values, and error handling. These concepts are useful to structure machines. On the other hand, to divide a specification into independent parts, the defined component concept is appropriate.

Components. We defined a notion of ASM components inspired by digital hardware circuits. Based on the defined formal semantics, we introduced a composition principle where inputs and outputs of single components are connected to each other in order to define the interaction among the components. Furthermore, we discuss the conditions under which the resulting composition is a well-defined component.

Usually, properties have to be proven about the composition of components (system properties), but due to complexity issues, this can not be done automatically; this is known as the *state explosion problem* in the literature. Therefore, we introduced a component-based verification technique which allows us to prove a desired system property in a less complex model. For this verification technique, we assume that for each concrete component there is a corresponding abstract component. However, we do not require a (known) functional relation between the models as it is usually required in the literature. The above mentioned less complex model is the composition of all abstract components together with one concrete component. Therefore, we have to apply the proof for each concrete component. The abstract components in this composition can be regarded as the simulation of the environment for the corresponding concrete component.

The aim is to show that if a desired system property holds in the less complex models, then the property also holds in the composition of the concrete components (without explicitly proving this). To ensure this, we must prove a stronger property in the less complex models.

In order to evaluate this ASM component concept, we built a tool to compile such components into VHDL code. This had the advantage that we could use existing VHDL tools (a model checker for VHDL, e.g.) to apply the introduced component based verification technique for automatically proving system properties.

Executing ASMs. We discussed how to extend and modify a semantics for lazy evaluation in order to support dynamic functions and simultaneous function updates which are essential features in ASMs.

Usually, in the literature, an ASM semantics is defined on top of a given semantics for terms. This means that new syntactic constructs and their interpretation are defined for ASM rules, dynamic functions, and function updates. Our semantics is not defined on top of a semantics for terms, because our aim is to extend an existing system for lazy evaluation (lazy evaluation is a semantics for λ-terms) according to the semantics extension. If we would define an ASM semantics on top of lazy evaluation, then we would have to introduce ASM rules, dynamic functions, and function updates in the corresponding implementation which implies to build a new tool based on another one. Hence, we represent all ASM features as λ-terms. We do this by defining new functional expressions and by modifying the lazy evaluation semantics in order to behave as expected in ASMs.

One main problem is the fact that referential transparency of expressions is no longer a property of the extended language. However, we adapted the lazy evaluation semantics in order to behave correctly according to ASM semantics.

We applied this semantics extension on the functional programming system TkGofer (based on lazy evaluation) and obtained AsmGofer—an advanced ASM programming system.

AsmGofer. AsmGofer is different from other ASM tools in a sense that rules are treated as expressions. In particular, one can define expressions on top of firing rules, because firing a rule is an expression, too. This allows to build graphical user interfaces to observe the current state and to execute arbitrary ASM rules. Usually, in other tools, rules can be executed only by the run-time system.

AsmGofer is usually too slow for performance critical applications (it is an interpreter), but it is rather suitable for executable specifications and their validation. Although there are some other ASM tools (ASM-Workbench [30], XASM [2], e.g.), the AsmGofer system is probably the most powerful and flexible tool with respect to the supported language features. For instance, multi agent ASMs are supported, non-deterministic selection of elements and rules, sequential execution and iteration of rules, higher-order functions, recursive definitions for rules and functions, constructor classes, graphical user interfaces, etc. Of course, some of these features come from TkGofer, but this illustrates the usefulness of combining functional programming and ASMs.

Applications. AsmGofer was extensively used in the book *Java and the Java Virtual Machine* [64] where the ASM models in the book for Java, the Java compiler, and the Java Virtual Machine are executable with this tool. The executable models come with a graphical user interface to observe the execution steps. In the models, recursive machines and machines parametrized over machines are used in order to get compact and intuitive definitions. Probably, the book contains the most complex application considered with ASMs in such a detailed way. Without AsmGofer, much more effort would had been necessary in order to get executable models.

In an industrial software project for train simulation (FALKO), we have demonstrated that in principle it is possible to compile structured ASMs into efficient *C++* code. In fact, in that project the generated code is used successfully in the final product release.

Outlook

The introduced component concept is focused on hardware systems. For software systems, one could think about a more general component concept. For instance, the possibility to use functions and rules in interface definitions. This would complicate the composition notion and the corresponding component-based verification technique.

Not all submachine and composition concepts are integrated into the lazy evaluation semantics and also not in the AsmGofer system. Some of them should be easy to integrate (catching inconsistent update sets, e.g.) and some others might be really hard (local state, e.g.), because we have to use the framework provided by Gofer.

AsmGofer allows to define expressions on top of rules (graphical user interfaces, e.g.). However, in the semantics extension we did not take into account this feature. This is a very interesting research area, because it leads to something like programming on a meta level for ASMs.

In the industrial application for railway simulation, the ASM specification was embedded into an HTML documentation. The implemented ASM compiler extracts from this documentation the formal parts and generates the *C++* code. This enabled seamless transition from the documentation to the executable code and was a major reason for the success of the project.

The procedure for the book *Java and the Java Virtual Machine* was similarly to the mentioned industrial application. This patency was very helpful for the design and implementation. However, the developed tools in this thesis are experimental. One useful continuation of this work could be to implement the introduced composition concepts in a tool suitable for industrial software design supporting seamless transition from the documentation til the executable code.

Zusammenfassung

Gurevich führt in [33] den Begriff der Abstrakten Zustandsmachinen (ASMs) ein. Diese Maschinen wurden in zahlreichen Anwendungen für den Entwurf und für die Analyse von Software- und Hardware-Systemen erfolgreich angewandt (einen Überblick gibt [16]). Ein wesentlicher Aspekt für diesen Erfolg ist ein allgemeiner Zustandsbegriff (eine mathematische Struktur) mit der notwendigen Abstraktionsmöglichkeit in Verbindung mit einer Zustandsübergangsfunktion (Maschinenschritt) als der simultanen Ausführung von atomaren Aktionen.

Die simultane Ausführung von atomaren Aktionen im gleichen Zustand hat jedoch ihren Preis: Bekannte Kompositions- und Strukturierungsprinzipien (notwendig für die Beschreibung von größeren Systemen) werden nicht unterstützt und können nur mit Hilfe von Verfeinerungsschritten verwendet werden. Ebenso hat auch die Möglichkeit zur Datenabstraktion, welche die Beschreibung eines Systems auf unterschiedlichen Ebenen ermöglicht, ihren Preis: In einem weiteren Verfeinerungsschritt müssen diese Abstraktionen in eine ausführbare Form gebracht werden, so daß eine Validierung der Spezifikation durch Simulation ermöglicht wird.

Die vorliegende Arbeit erweitert das Spektrum der Einsatzmöglichkeiten von ASMs für das Software Engineering hinsichtlich der folgenden Punkte:

- Definition von drei wichtigen Kompositionskonzepten für ASMs: Komponenten, Parametrisierte Untermaschinen und Iteration.

- Entwicklung eines Werkzeuges für die Ausführung der erweiterten ASMs welches eine grafische Benutzeroberfläche für die experimentielle Analyse ermöglicht.

Parametrisierte Untermaschinen und Iteration erlauben die Strukturierung von einzelnen atomaren Aktionen; das Komponentenkonzept hingegen ermöglicht die Aufteilung einer Spezifikation in unabhängige kleinere Teile, welche zusammen das Gesamtsystem beschreiben. Basierend auf diesem Komponentenkonzept stellt die Arbeit eine Verifikationsmethode vor, die sich für Korrektheitsbeweise von Komponenten und deren Komposition eignet.

Die Grundlage für das entwickelte Werkzeug zur Ausführung der erweiterten ASMs bildet eine Semantikbetrachtung für eine funktionale Sprache mit verzögerter Auswertung und deren Erweiterung im Hinblick auf die Anforderungen zur Ausführung von ASMs (beispielsweise dynamische Funktionen und simultane Funktionsaktualisierungen). Das entstandene Werkzeug *AsmGofer* ist eine konservative Erweiterung des Gofer Systems.

Die Anwendbarkeit der vorgestellten Kompositionskonzepte und deren Implementierungen wurde für den Entwurf und die Analyse von folgenden Anwendungen evaluiert:

- eine aus der Literatur bekannte Fallstudie und eine industrielle Anwendung mittlerer Größe für die Simulation von Zugfahrplänen. In diesem Projekt entstand auch ein Compiler von ASMs nach *C++*. Die entsprechende ASM-Spezifikation wurde von Peter Päppinghaus und Joachim Schmid gemacht. Das Compilations-Schema wurde entworfen und implementiert von Joachim Schmid.

- *Java and the Java Virtual Machine*. Ein Buch, das die Definition, Verifikation und die Validierung von Java und der Virtuellen Maschine mit Hilfe von ASMs behandelt. Die Beweise wurden von Robert Stärk und Egon Börger entwickelt. Der Bytecode-Verifier wurde von Robert Stärk und Joachim Schmid entworfen und Robert Stärk zeigt, dass der Verifier jedes legale Java-Programm — erzeugt nach dem vorgestellten Compilations-Schema — akzeptiert. Die ausführbaren Modelle mit AsmGofer wurden von Joachim Schmid entwickelt.

- ein industrielles ASIC Projekt mit formaler Verifikation. In diesem Projekt entstand ein Compiler von ASMs nach VHDL welcher bereits von Giuseppe Del Castillo für ein Verifikationsprojekt eingesetzt wurde.

In der industriellen Anwendung für die Simulation von Zugfahrplänen wurde die ASM-Spezifikation in eine HTML-Dokumentation eingebettet. Ausgehend von der HTML-Dokumentation konnte der *C++* Code erzeugt werden. Dies ermöglichte den nahtlosen Übergang von der Dokumentation bis hin zum ausführbaren Modell und trug wesentlich zum Erfolg des Projektes bei.

Ähnlich wurde auch beim Buch *Java and the Java Virtual Machine* vorgegangen. Diese Durchgängigkeit hat sich als sehr hilfreich und nützlich erwiesen. Allerdings ist die Werkzeugunterstützung (entstanden in dieser Arbeit) noch experimentell. Eine mögliche Fortführung dieser Arbeit wäre die Integration der vorgestellten Kompositionskonzepte in einem Werkzeug, das industriellen Anforderungen genügt, und welches die Durchgängigkeit von der Dokumentation bis hin zum ausführbaren Modell unterstützt.

Teile der Arbeit und deren Anwendung sind veröffentlich in [17] (industrielle Anwendung), [18] (Fallstudie aus der Literatur), [19] (Kompositions- und Strukturierungsprinzipien), [61] (Kompilierung von ASMs nach *C++*) und [64] (Java und die Virtuelle Machine).

Appendix A

Submachine Concept

A.1 Deduction Rules for Update Sets

The following rules provide a calculus for computing the semantics of standard ASMs and for the constructs introduced in this paper.

We use R, R_i, and S for rules, f for functions, x for variables, s and t for expressions, p for predicates (boolean expressions), and u and v for semantical values and update sets.

Function and variable evaluation.

$$\frac{\forall\ i : [\![t_i]\!]^{\mathfrak{A}}_{\zeta} = v_i}{[\![f(t_1, \ldots, t_n)]\!]^{\mathfrak{A}}_{\zeta} = f^{\mathfrak{A}}(v_1, \ldots, v_n)} \qquad \frac{}{[\![x]\!]^{\mathfrak{A}}_{\zeta} = \zeta(x)}\ \mathit{variable}(x)$$

Skip rule.

$$\frac{}{[\![\textbf{skip}\ ]\!]^{\mathfrak{A}}_{\zeta} = \varnothing}$$

Guarded rules.

$$\frac{[\![t]\!]^{\mathfrak{A}}_{\zeta} = \mathrm{true}^{\mathfrak{A}},\ [\![R]\!]^{\mathfrak{A}}_{\zeta} = u}{[\![\textbf{if}\ t\ \textbf{then}\ R\ \textbf{else}\ S]\!]^{\mathfrak{A}}_{\zeta} = u} \qquad \frac{[\![t]\!]^{\mathfrak{A}}_{\zeta} = \mathrm{false}^{\mathfrak{A}},\ [\![S]\!]^{\mathfrak{A}}_{\zeta} = u}{[\![\textbf{if}\ t\ \textbf{then}\ R\ \textbf{else}\ S]\!]^{\mathfrak{A}}_{\zeta} = u}$$

Function update.

$$\frac{\forall\ i : [\![t_i]\!]^{\mathfrak{A}}_{\zeta} = v_i,\ [\![s]\!]^{\mathfrak{A}}_{\zeta} = u}{[\![f(t_1, \ldots, t_n) := s]\!]^{\mathfrak{A}}_{\zeta} = \{(f\langle v_1, \ldots, v_n\rangle, u)\}}$$

Parallel combination and *let* rule.

$$\frac{\forall\ i : [\![R_i]\!]^{\mathfrak{A}}_{\zeta} = u_i}{[\![\{R_1, \ldots, R_n\}]\!]^{\mathfrak{A}}_{\zeta} = u_1 \cup \cdots \cup u_n} \qquad \frac{[\![t]\!]^{\mathfrak{A}}_{\zeta} = v,\ [\![R]\!]^{\mathfrak{A}}_{\zeta\frac{x}{v}} = u}{[\![\textbf{let}\ x = t\ \textbf{in}\ R]\!]^{\mathfrak{A}}_{\zeta} = u}$$

Quantified rules.

$$\frac{V = \{v_1, \ldots, v_n\},\ \forall\ i : [\![R]\!]^{\mathfrak{A}}_{\zeta \frac{x}{v_i}} = u_i}{[\![\textbf{forall}\ x\ \textbf{with}\ p\ \textbf{do}\ R]\!]^{\mathfrak{A}}_{\zeta} = u_1 \cup \cdots \cup u_n}\ V = \{v \mid [\![p]\!]^{\mathfrak{A}}_{\zeta \frac{x}{v}} = \text{true}^{\mathfrak{A}}\}$$

$$\frac{[\![p]\!]^{\mathfrak{A}}_{\zeta \frac{x}{v}} = \text{true}^{\mathfrak{A}},\ [\![R]\!]^{\mathfrak{A}}_{\zeta \frac{x}{v}} = u}{[\![\textbf{choose}\ x\ \textbf{with}\ p\ \textbf{do}\ R]\!]^{\mathfrak{A}}_{\zeta} = u}$$

$$\frac{}{[\![\textbf{choose}\ x\ \textbf{with}\ p\ \textbf{do}\ R]\!]^{\mathfrak{A}}_{\zeta} = \varnothing}\ \nexists\ v : [\![p]\!]^{\mathfrak{A}}_{\zeta \frac{x}{v}} = \text{true}^{\mathfrak{A}}$$

Sequential composition.

$$\frac{[\![R]\!]^{\mathfrak{A}}_{\zeta} = u,\ [\![S]\!]^{fire_{\mathfrak{A}}(u)}_{\zeta} = v}{[\![R\ \textbf{seq}\ S]\!]^{\mathfrak{A}}_{\zeta} = u \oplus v}\ consistent(u)$$

$$\frac{[\![R]\!]^{\mathfrak{A}}_{\zeta} = u}{[\![R\ \textbf{seq}\ S]\!]^{\mathfrak{A}}_{\zeta} = u}\ inconsistent(u)$$

Iteration.

$$\frac{[\![R^n]\!]^{\mathfrak{A}}_{\zeta} = u}{[\![\textbf{iterate}(R)]\!]^{\mathfrak{A}}_{\zeta} = u}\ n \geq 0, inconsistent(u)$$

$$\frac{[\![R^n]\!]^{\mathfrak{A}}_{\zeta} = u,\ [\![R]\!]^{fire_{\mathfrak{A}}(u)}_{\zeta} = \varnothing}{[\![\textbf{iterate}(R)]\!]^{\mathfrak{A}}_{\zeta} = u}\ n \geq 0, consistent(u)$$

Parameterized Rules with local state. Let R be a named rule as in Section 1.4.1:

$$\frac{[\![(\{Init_1, \ldots, Init_k\}\ \textbf{seq}\ body)[a_1/x_1, \ldots, a_n/x_n]]\!]^{\mathfrak{A}}_{\zeta} = u}{[\![R(a_1, \ldots, a_n)]\!]^{\mathfrak{A}}_{\zeta} = u \setminus Updates(f_1, \ldots, f_k)}$$

Error Handling.

$$\frac{[\![R]\!]^{\mathfrak{A}}_{\zeta} = u}{[\![\textbf{try}\ R\ \textbf{catch}\ f(t_1, \ldots, t_n)\ S]\!]^{\mathfrak{A}}_{\zeta} = u}\ \begin{array}{l}\nexists\ v_1 \neq v_2 : (loc, v_1) \in u \wedge (loc, v_2) \in u \\ where\ \ loc = f\langle [\![t_1]\!]^{\mathfrak{A}}_{\zeta}, \ldots, [\![t_n]\!]^{\mathfrak{A}}_{\zeta}\rangle\end{array}$$

$$\frac{[\![R]\!]^{\mathfrak{A}}_{\zeta} = u,\ [\![S]\!]^{\mathfrak{A}}_{\zeta} = v}{[\![\textbf{try}\ R\ \textbf{catch}\ f(t_1, \ldots, t_n)\ S]\!]^{\mathfrak{A}}_{\zeta} = v}\ \begin{array}{l}\exists\ v_1 \neq v_2 : (loc, v_1) \in u \wedge (loc, v_2) \in u \\ where\ \ loc = f\langle [\![t_1]\!]^{\mathfrak{A}}_{\zeta}, \ldots, [\![t_n]\!]^{\mathfrak{A}}_{\zeta}\rangle\end{array}$$

Remark A.1 The second rule for *choose* reflects the decision in [33] that an ASM does nothing when there is no choice. Obviously also other decisions could be formalized in this manner, e.g. yielding instead of the empty set an update set which contains an error report.

Remark A.2 The rule for *forall* is formulated as finitary rule, i.e. it can be applied only for quantifying over finite sets. The set theoretic formulation in Section 1.1 is more general and can be formalized by an infinitary rule. It would be quite interesting to study different classes of ASMs, corresponding to different finitary or infinitary versions of the *forall* construct.

Appendix B

Component Concept

B.1 Syntax

Type definition. A type is either an enumeration of type constructors or an array type; the range for array types is optional.

typedef ::= $\underline{\{}$ *type_constructor* ($\underline{,}$ *type_constructor*)* $\underline{\}}$
| `array of` *type*
| `array[` *range* `] of` *type*
range ::= *constant* (`to` | `downto`) *constant*
type_constructor ::= *id* | *char*

Type instantiation. Array types can be restricted to ranges. Additionally, *integer* is a valid type.

type ::= *id* ([*range*])* | `integer` | $\underline{(}$ *type* $\underline{)}$ ([*range*])*

Term syntax. Terms can be defined using *if-then-else*, binary operators, unary operators, function applications, and constants.

term ::= *if_then_else_term*
if_then_else_term ::= `if` *term* `then` *term* `else` *term* | *or_xor_term*
or_xor_term ::= *and_term* ((`or` | `xor`) *and_term*)*
and_term ::= *relational_term* (`and` *relational_term*)*
relational_term ::= *rotate_term* ((= | ≠ | < | ≤ | > | ≥) *add_sub_term*)?
add_sub_term ::= *mul_term* ((+ | − | &) *mul_term*)*
mul_term ::= *not_term* (∗ *not_term*)*
not_term ::= `not` *not_term* | *unary_term*
unary_term ::= + *not_term* | − *not_term* | *postfix_term*
postfix_term ::= *primary_term*([*term* | *constant* (`to` | `downto`) *constant*])*
primary_term ::= *funterm* | *basic_constant* | $\underline{(}$ *term* $\underline{)}$ | *id* ′ $\underline{(}$ *term* $\underline{)}$

Basic constants. Integers, characters, and bit-vectors (sequence of 0 and 1) are constants.

basic_constant ::= *integer* | *char* | ″ (0 | 1)$^+$ ″

Constants. Expressions containing only constant expressions, are treated as con-

stants, too.

$$
\begin{aligned}
&\mathit{constant} ::= \mathit{constadd}\\
&\mathit{constadd} ::= \mathit{constmul}\,((+ \mid -)\,\mathit{constmul})^*\\
&\mathit{constmul} ::= \mathit{constexp}\,(*\ \mathit{constexp})^*\\
&\mathit{constexp} ::= \mathit{integer} \mid \mathit{id} \mid \underline{(}\mathit{constant}\underline{)}
\end{aligned}
$$

Basic Universes. *ID* denotes identifiers, *INT* denotes integer values, and *CHAR* denotes characters:

$$
\begin{aligned}
&\mathit{id} ::= \mathit{ID}\\
&\mathit{integer} ::= \mathit{INT}\\
&\mathit{char} ::= \mathit{CHAR}
\end{aligned}
$$

B.2 Semantics

Values. A value is either an integer, a type constructor, or a sequence of values (used for arrays).

$$\mathit{Val} = \mathit{Integer} \mid \mathit{Constructor} \mid \mathit{Sequence}(\mathit{Val}^*)$$

To represent the interpretation of types, we use the following definition of *Type*:

$$
\begin{aligned}
&\mathit{Type} = \mathit{Integer} \mid \mathit{Boolean} \mid \mathit{Inst}(\mathit{Type}, \mathit{Range})\\
&\mid \mathit{Array}(\mathit{Type}) \mid \mathit{Constructors}((\mathit{Char} \mid \mathit{id})^*) \mid \mathit{Rule}
\end{aligned}
$$

Type definitions.

$$
\begin{aligned}
&[\![\underline{\{}\mathit{cons}_1\underline{,}\ldots\underline{,}\mathit{cons}_n\underline{\}}]\!] &&= \mathit{Constructors}(\mathit{cons}_1, \ldots, \mathit{cons}_n)\\
&[\![\texttt{array of}\ \mathit{type}]\!] &&= \mathit{Array}([\![\mathit{type}]\!])\\
&[\![\texttt{array[}\ \mathit{range}\ \texttt{]of}\ \mathit{type}]\!] &&= \mathit{Inst}(\mathit{Array}([\![\mathit{type}]\!]), [\![\mathit{range}]\!])
\end{aligned}
$$

$$
\begin{aligned}
&[\mathit{typedef}] \mapsto \mathit{def} \Leftrightarrow [\![\mathit{typedef}]\!] = \mathit{def}\\
&[\mathit{type}] \mapsto t \Leftrightarrow [\![\mathit{type}]\!] = t
\end{aligned}
$$

Types. The following lines define the interpretation of types. The notion i_1 `to` i_2 and i_1 `downto` i_2 are range specifications for arrays.

$$
\begin{aligned}
&[\![\texttt{integer}]\!] &&= \mathit{Integer}\\
&[\![\mathit{type}\ [\ \mathit{range}\ ]]\!] &&= \mathit{Inst}([\![\mathit{type}]\!], [\![\mathit{range}]\!])\\
&[\![\underline{(}\mathit{type}\underline{)}]\!] &&= [\![\mathit{type}]\!]\\
&[\![\mathit{id}]\!] &&= \mathit{normalize}(\mathit{id})\\
&[\![i_1\ \texttt{to}\ i_2]\!] &&= \mathit{To}(i_1, i_2)\\
&[\![i_1\ \texttt{downto}\ i_2]\!] &&= \mathit{DownTo}(i_1, i_2)
\end{aligned}
$$

The function *normalize* is defined inductively on types and substitutes identifiers by the corresponding type definitions.

$$
\begin{aligned}
&\mathit{normalize}(\mathit{id}) = \begin{cases} \mathit{normalize}(\mathit{type}), & \mathit{Alias}(\mathit{id}, \mathit{type}) \in \mathit{env}\\ \mathit{normalize}(\mathit{type}), & \mathit{TypeDef}(\mathit{id}, \mathit{type}) \in \mathit{env} \end{cases}\\
&\mathit{normalize}(\mathit{Array}(\mathit{type})) &&= \mathit{Array}(\mathit{normalize}(\mathit{type}))\\
&\mathit{normalize}(\mathit{Inst}(\mathit{type}, \mathit{range})) &&= \mathit{Inst}(\mathit{normalize}(\mathit{type}), \mathit{range})\\
&\mathit{normalize}(\mathit{Integer}) &&= \mathit{Integer}\\
&\mathit{normalize}(\mathit{Boolean}) &&= \mathit{Boolean}\\
&\mathit{normalize}(\mathit{Constructors}(c_1, \ldots, c_n)) &&= \mathit{Constructors}(c_1, \ldots, c_n)
\end{aligned}
$$

If-then-else expressions.

$$[\![\texttt{if } cond \texttt{ then } term_1 \texttt{ else } term_2]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta} = \begin{cases} [\![term_1]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta}, & [\![cond]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta} = true \\ [\![term_2]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta}, & [\![cond]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta} = false \end{cases}$$

Binary operators.

$$[\![^{\alpha} term_1 \; bop \; ^{\beta} term_2]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta} = [\![bop_{\mathcal{T}(\alpha),\mathcal{T}(\beta)}]\!]([\![term_1]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta}, [\![term_2]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta})$$

Unary operators.

$$[\![uop \; ^{\alpha} term]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta} = [\![uop_{\mathcal{T}(\alpha)}]\!]([\![term]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta})$$

Identifiers.

$$[\![f]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta} = \begin{cases} \zeta(f), & f \text{ is formal parameter} \\ \mathfrak{i}(f), & f \text{ is input identifier} \\ \mathfrak{s}(f), & f \text{ is state variable} \\ f_{ext}, & f \text{ is library function} \\ g, & f \text{ is function definition,} \\ & g(p_1, \ldots, p_n) = [\![body]\!]^{\mathfrak{i},\mathfrak{s}}_{id_1 \to p_1, \ldots, id_n \to p_n} \text{ and} \\ & Fun(f, \langle id_1, \ldots, id_n \rangle, body) \in env \end{cases}$$

Function application.

$$[\![f \; \underline{(} term_1 \underline{,} \ldots \underline{,} term_n \underline{)}]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta} = [\![f]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta}([\![term_1]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta}, \ldots, [\![term_n]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta})$$

Array indexing and slicing.

$$\begin{aligned} &[\![^{\alpha} term \; [\; index \;]]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta} = val_i \\ &\quad \textbf{where } Sequence(val_1, \ldots, val_n) = [\![term]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta} \\ &\qquad\quad Inst(\_, range) = \mathcal{T}(\alpha) \\ &\qquad\quad i = select(range, [\![index]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta}) \\ &[\![^{\alpha} term \; [\; range_1 \;]]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta} = Sequence(val_i, \ldots, val_j) \\ &\quad \textbf{where } Sequence(val_1, \ldots, val_n) = [\![term]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta} \\ &\qquad\quad Inst(\_, range_2) = \mathcal{T}(\alpha) \\ &\qquad\quad (i, j) = select(range_2, [\![range_1]\!]) \end{aligned}$$

Basic terms.

$$\begin{aligned} [\![integer]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta} &= integer \\ [\![char]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta} &= char \\ [\![vector]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta} &= Sequence(vector) \end{aligned}$$

Type cast.

$$[\![id \; ' \; \underline{(} term \underline{)}]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta} = [\![term]\!]^{\mathfrak{i},\mathfrak{s}}_{\zeta}$$

Given a range specification and an index or another range specification, the function *select* computes the corresponding element number(s) in a sequence of values.

$$\begin{aligned} &select(To(i_1, i_2), j) = 1 + j - i_1 \\ &select(DownTo(i_1, i_2), j) = 1 + i_1 - j \\ &select(To(i_1, i_2), To(j_1, j_2)) = (1 + j_1 - i_1, 1 + j_2 - i_1) \\ &select(DownTo(i_1, i_2), DownTo(j_1, j_2)) = (1 + i_1 - j_1, 1 + i_1 - j_2) \end{aligned}$$

B.3 Type system

Type compatibility is denoted by the symbol $\preceq$. If $t_1 \preceq t_2$, then the type t_1 may be used when t_2 is expected. This is implies, that t_2 is more general than t_1. The relation is defined as follows:

$$\begin{array}{lll} \mathit{Inst}(t_1, r_1) & \preceq \mathit{Inst}(t_2, r_2) & = t_1 \preceq t_2 \wedge \mathit{size}(r_1) = \mathit{size}(r_2) \\ \mathit{Inst}(t_1, r) & \preceq \mathit{Array}(t_2) & = t_1 \preceq t_2 \\ \mathit{Array}(t_1) & \preceq \mathit{Array}(t_2) & = t_1 \preceq t_2 \\ t & \preceq t & = \mathit{True} \end{array}$$

In the remaining part of this section, we list inference rules to deduce the type of terms and rules. $\mathcal{T}(t)$ denotes the type of t.

If-then-else expressions.

$$\frac{\mathcal{T}(\mathit{term}_1) \preceq \mathcal{T}(\mathit{term}_2),\ \mathcal{T}(\mathit{cond}) = \mathit{Boolean}}{\mathcal{T}(\texttt{if}\ \mathit{cond}\ \texttt{then}\ \mathit{term}_1\ \texttt{else}\ \mathit{term}_2) = \mathcal{T}(\mathit{term}_2)}$$

$$\frac{\mathcal{T}(\mathit{term}_2) \preceq \mathcal{T}(\mathit{term}_1),\ \mathcal{T}(\mathit{cond}) = \mathit{Boolean}}{\mathcal{T}(\texttt{if}\ \mathit{cond}\ \texttt{then}\ \mathit{term}_1\ \texttt{else}\ \mathit{term}_2) = \mathcal{T}(\mathit{term}_1)}$$

Binary operators.

$$\frac{\mathcal{T}(\mathit{term}_1) = t_1,\ \mathcal{T}(\mathit{term}_2) = t_2,\ \mathcal{T}(\mathit{bop}_{t_1,t_2}) = t_{bin}}{\mathcal{T}(\mathit{term}_1\ \mathit{bop}\ \mathit{term}_2) = t_{bin}}$$

Unary operators.

$$\frac{\mathcal{T}(\mathit{term}) = t,\ \mathcal{T}(\mathit{uop}_t) = t_{un}}{\mathcal{T}(\mathit{uop}\ \mathit{term}) = t_{un}}$$

Local variables. We use $\mathcal{T}_{n \cdot id}$ to denote the type of parameter id in function or rule n.

$$\frac{\mathit{id}\text{ is formal parameter, } n\text{ is current function or rule}}{\mathcal{T}(\mathit{id}) = \mathcal{T}_{n \cdot id}}$$

Input and Output identifiers.

$$\frac{\mathit{In}(\mathit{id}, \mathit{type}) \in \mathit{env}}{\mathcal{T}(\mathit{id}) = \mathit{type}} \qquad \frac{\mathit{Out}(\mathit{id}, \mathit{type}) \in \mathit{env}}{\mathcal{T}(\mathit{id}) = \mathit{type}}$$

State variables.

$$\frac{\mathit{State}(\mathit{id}, \mathit{type}, \langle t_1, \ldots, t_n \rangle) \in \mathit{env},\ \mathcal{T}(\mathit{term}_i) \preceq t_i}{\mathcal{T}(f(\mathit{term}_1, \ldots, \mathit{term}_n)) = \mathit{type}}$$

Library function signatures.

$$\frac{\mathit{Sig}(\mathit{id}, \mathit{type}, \langle t_1, \ldots, t_n \rangle) \in \mathit{env},\ \mathcal{T}(\mathit{term}_i) \preceq t_i}{\mathcal{T}(f(\mathit{term}_1, \ldots, \mathit{term}_n)) = \mathit{type}}$$

Function signatures.

$$\frac{\mathit{Fun}(f, \langle \mathit{id}_1, \ldots, \mathit{id}_n \rangle, \mathit{body}) \in \mathit{env},\ \mathcal{T}(\mathit{term}_i) \preceq \mathcal{T}_{f \cdot \mathit{id}_i},\ \mathcal{T}(\mathit{body}) = t}{\mathcal{T}(f(\mathit{term}_1, \ldots, \mathit{term}_n)) = t}$$

Array indexing and slicing.

$$\frac{\mathcal{T}(\mathit{term}) = \mathit{Inst}(\mathit{type}, \_),\ \mathcal{T}(\mathit{index}) = \mathit{Integer}}{\mathcal{T}(\mathit{term}\ [\ \mathit{index}\ ]) = \mathit{type}}$$

$$\frac{\mathcal{T}(term) = Inst(type, \_)}{\mathcal{T}(term\ [\ range\ ]) = Inst(type, [\![range]\!])}$$

Integers.

$$\mathcal{T}(integer) = Integer$$

Type constructors.

$$\frac{TypeDef(id, Constructors(c_1, \ldots, c_n)) \in env}{\mathcal{T}(c_i) = normalize(id)}$$

Vectors.

$$\frac{\begin{array}{l} TypeDef(id, t) \in env,\ t = Constructors(c_1, \ldots, c_n) \\ vector = v_1 \cdot \ldots \cdot v_k,\ v_i \in \{c_1, \ldots, c_n\} \end{array}}{\mathcal{T}(vector) = Inst(Array(t)), DownTo(k-1, 0)}$$

Type casts.

$$\frac{normalize(id) = t, t \preceq \mathcal{T}(term)}{\mathcal{T}(id\ '\ \underline{(}term\underline{)}) = t}$$

Skip rule.

$$\mathcal{T}(\texttt{skip}) = Rule$$

Parallel rules.

$$\frac{\mathcal{T}(rule_i) = Rule}{\mathcal{T}(\underline{\{}rule_1 \ldots rule_n\underline{\}}) = Rule}$$

Function update.

$$\frac{State(f, type, \varepsilon) \in env,\ \mathcal{T}(term) \preceq type}{\mathcal{T}(f := term) = Rule}$$

$$\frac{State(f, type, \langle t_1, \ldots, t_n\rangle) \in env,\ \mathcal{T}(term_i) \preceq t_i,\ \mathcal{T}(term) \preceq type}{\mathcal{T}(f\underline{(}term_1\underline{,}\ldots\underline{,}term_n\underline{)} := term) = Rule}$$

Call rule.

$$\frac{Rule(r, \varepsilon, body) \in env,\ \mathcal{T}(body) = Rule}{\mathcal{T}(r) = Rule}$$

$$\frac{Rule(r, \langle id_1, \ldots, id_n\rangle, body) \in env,\ \mathcal{T}(term_i) \preceq \mathcal{T}_{r \cdot id_i},\ \mathcal{T}(body) = Rule}{\mathcal{T}(r\underline{(}term_1\underline{,}\ldots\underline{,}term_n\underline{)}) = Rule}$$

Conditional rule.

$$\frac{\mathcal{T}(cond) = Boolen,\ \mathcal{T}(rule_i) = Rule}{\mathcal{T}(\texttt{if}\ cond\ \texttt{then}\ rule_1\ \texttt{else}\ rule_2) = Rule}$$

B.4 Constraints

Constraints for $C = [\texttt{component}\ id\ \textit{uses}\ decl_1 \ldots decl_n\ \texttt{end component}]$

Main rule. The component contains a rule having the same name as the component:

$$\exists\ Rule(id, \varepsilon, body) \in env \colon \mathcal{T}(id) = Rule$$

Outputs. For each output id_1, there is either a nullary function with the same name, or id_1 is connected to an output id_2. The corresponding types must be compatible.

$$\begin{aligned}\forall\ Out(id_1, type) \in env \colon \exists\ Fun(id_1, \varepsilon, body) \in env \colon \mathcal{T}(id_1) \preceq type \vee \\ \exists\ OutOut(id_1, id_2) \in env \colon \mathcal{T}(id_2) \preceq type\end{aligned}$$

Type compatibility of connections.

$$\begin{aligned}\forall\ OutOut(id_1, id_2) &\in env \colon \mathcal{T}(id_2) \preceq \mathcal{T}(id_1) \\ \forall\ InOut(id_1, id_2) &\in env \colon \mathcal{T}(id_2) \preceq \mathcal{T}(id_1) \\ \forall\ InIn(id_1, id_2) &\in env \colon \mathcal{T}(id_2) \preceq \mathcal{T}(id_1) \\ \forall\ InFun(id_1, id_2) &\in env \colon \mathcal{T}(id_2) \preceq \mathcal{T}(id_1)\end{aligned}$$

References

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

[2] M. Anlauff. XASM – An extensible, component-based Abstract State Machines language. In Gurevich et al. [36], pages 69–90.

[3] M. Anlauff, D. Fischer, P. W. Kutter, J. Teich, and R. Weper. Hierarchical microprocessor design using XASM. In Moreno-Dìaz and Quesanda-Arencibia [52], pages 271–274. Extended Abstract.

[4] H. Barendregt. *The Lambda Calculus. Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1981.

[5] S. Berezin, S. Campos, and E. M. Clarke. Compositional reasoning in model checking. In W.-P. de Roever, H. Langmaack, and A. Pnueli, editors, *Compositionality. The Significant Difference*, volume 1536 of *Lecture Notes in Computer Science*, pages 81–102. Springer-Verlag, 1998.

[6] J. Bergé, O. Levia, and J. Rouillard. *Hardware Component Modeling*. Kluwer Academic Publishers, 1996.

[7] R. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 1998.

[8] C. Böhm and G. Jacopini. Flow diagrams, Turing Machines, and languages with only two formation rules. *Communications of the ACM*, 9(5):366–371, 1966.

[9] E. Börger. Why use evolving algebras for hardware and software engineering? In M. Bartosek, J. Staudek, and J. Wiederman, editors, *Proceedings of SOFSEM'95, 22nd Seminar on Current Trends in Theory and Practice of Informatics*, volume 1012 of *Lecture Notes in Computer Science*, pages 236–271. Springer-Verlag, 1995.

[10] E. Börger. High level system design and analysis using Abstract State Machines. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Current Trends in Applied Formal Methods (FM-Trends 98)*, number 1641 in Lecture Notes in Computer Science, pages 1–43. Springer-Verlag, 1999.

[11] E. Börger, A. Cavarra, and E. Riccobene. An ASM semantics for UML Activity Diagrams. In T. Rust, editor, *Algebraic Methology and Software*

Technology, number 1816 in Lecture Notes in Computer Science. Springer-Verlag, 2000.

[12] E. Börger, A. Cavarra, and E. Riccobene. Modeling the dynamics of UML state machines. In Gurevich et al. [36], pages 223–241.

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

[14] E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem*. Perspectives in Mathematical Logic. Springer-Verlag, 1997.

[15] E. Börger, B. Hörger, D. L. Parnas, and D. Rombach, editors. *Requirements Capture, Documentation and Validation*, Dagstuhl-Seminar-Report 242, 1999. Web pages at http://www.iese.fhg.de/Dagstuhl/seminar99241.html.

[16] E. Börger and J. Huggins. Abstract State Machines 1988–1998. Commented ASM bibliography. *Bulletin of EATCS*, 64:105–127, February 1998. Updated bibliography available at http://www.eecs.umich.edu/gasm/.

[17] E. Börger, P. Päppinghaus, and J. Schmid. Report on a practical application of ASMs in software design. In Gurevich et al. [36], pages 361–366. Online available at http://www.tydo.de/files/papers/.

[18] E. Börger, E. Riccobene, and J. Schmid. Capturing requirements by Abstract State Machines. the Light Control case study. *Journal of Universal Computer Science*, 6(7), 2000.

[19] E. Börger and J. Schmid. Composition and submachine concepts. In P. G. Clote and H. Schwichtenberg, editors, *Computer Science Logic (CSL 2000)*, number 1862 in Lecture Notes in Computer Science, pages 41–60. Springer-Verlag, 2000.

[20] E. Börger and W. Schulte. Modular design for the Java Virtual Machine architecture. In E. Börger, editor, *Architecture Design and Validation Methods*, pages 297–357. Springer-Verlag, 2000.

[21] W. Brauer. *Automatentheorie*. B. G. Teubner, 1984.

[22] 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 Lecture Notes in Computer Science, pages 198–236. Springer-Verlag, 1984.

[23] A. Cavarra and E. Riccobene. Simulating UML statecharts. In Moreno-Dìaz and Quesanda-Arencibia [52], pages 224–227. Extended Abstract.

[24] K. C. Chang. *Digitial Design and Modeling with VHDL and Synthesis*. IEEE Computer Society Press, 1997.

[25] K. Chen and M. Odersky. A type system for a lambda calculus with assignment. In *Theoretical Aspects of Computer Science*, number 789 in Lecture Notes in Computer Science, pages 347–364. Springer-Verlag, 1994.

[26] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, pages 343–354, 1992.

[27] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[28] M. Davis. *The Universal Computer. The Road from Leibniz to Turing.* W.W. Norton, New York, 2000.

[29] G. Del Castillo. The ASM Workbench. an open and extensible tool environment for Abstract State Machines. In *Proceedings of the* 28^{th} *Annual Conference of the German Society of Computer Science*. Technical Report, Magdeburg University, 1998.

[30] G. Del Castillo. *The ASM-Workbench. A tool environment for computer aided analysis and validation of ASM models.* PhD thesis, University of Paderborn, 2000.

[31] A. Dold. A formal representation of Abstract State Machines using PVS. *Verifix Report Ulm/6.2*, 1998.

[32] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, 1994.

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

[34] Y. Gurevich. May 1997 draft of the ASM guide. Technical Report CSE-TR-336-97, University of Michigan EECS Department, 1997.

[35] Y. Gurevich. Sequential Abstract State Machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1(1), 2000.

[36] Y. Gurevich, M. Odersky, and L. Thiele, editors. *Abstract State Machines, ASM 2000*, number 1912 in Lecture Notes in Computer Science. Springer-Verlag, 2000.

[37] Y. Gurevich and M. Spielmann. Recursive Abstract State Machines. *Journal of Universal Computer Science*, 3(4):233–246, 1997.

[38] C. Hall, K. Hammond, S. P. Jones, and P. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, 1996.

[39] U. Heinkel. *VHDL reference. A pratical guide to computer-aided integrated circuit design, including VHDL-ASM.* John Wiley Ltd, 2000.

[40] T. A. Henzinger, S. Qadeer, S. K. Rajamani, and S. Tasıran. An assume-guarantee rule for checking simulation. *Formal Methods in Computer-Aided Design*, 1997.

[41] G. Hutton and E. Meijer. Monadic parser combinators. Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.

[42] M. P. Jones. An introduction to Gofer, 1991. Available in the Gofer distribution at http://www.cse.ogi.edu/~mpj/goferarc/index.html.

[43] M. P. Jones. Gofer. Functional programming environment, 1994. Web page at http://www.cse.ogi.edu/~mpj/goferarc/index.html.

[44] M. P. Jones. The implementation of the Gofer functional programming system. *Research Report YALEU/DCS/RR-1030*, 1994.

[45] S. P. Jones and P. Wadler. Imperative functional programming. In *Conference record of the* 20^{th} *Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina*, pages 71–84, 1993.

[46] S. C. Kleene. *Introduction to Metamathematics*. D. van Nostrand, Princeton, New Jersey, 1952.

[47] J. Launchbury. A natural semantics for lazy evaluation. In *Conference Record of the* 20^{th} *Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 144–154, 1993.

[48] L. Lavagno and A. Sangiovanni-Vincentelli. *Algorithms for Synthesis and Testing of Asynchronous Circuits*. Kluwer Academic Publishers, 1993.

[49] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 333–343, 1995.

[50] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. Technical Report MIT/LCS/TM-373, M.I.T. Laboratory for Computer Science, 1988.

[51] J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. *Journal of Functional Programming*, 8(3), 1994.

[52] R. Moreno-Dìaz and A. Quesanda-Arencibia, editors. *Formal Methods and Tools for Computer Science, Eurocast*. Universidad de Las Palmas de Gran Canaria, 2001.

[53] M. Odersky. How to make destructive updates less destructive. In *Proceedings,* 18^{th} *ACM Symposium on Principles of Programming Languages*, pages 25–26, 1991.

[54] M. Odersky. Programming with variable functions. *ICFP'98, Proceedings of the* 3^{rd} *ACM SIGPLAN International Conference on Functional Programming*, 34(1):105–116, 1998.

[55] M. Odersky, D. Rabin, and P. Hudak. Call-by-name, assignment, and the lambda calculus. In *Conference record of the* 20^{th} *Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina*, pages 43–56, 1993.

[56] D. L. Parnas. Information distribution aspects of design methodology. In *Information Processing 71*, pages 339–344. North Holland Publishing Company, 1972.

[57] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1996.

[58] A. Pnuelli. In transition from global to modular temporal reasoning about programs. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, Computer and System Science, pages 123–144. Springer-Verlag, 1985.

[59] G. Schellhorn. *Verifikation abstrakter Zustandsmaschinen*. PhD thesis, University of Ulm, 1999. For an english version see http://www.informatik.uni-ulm.de/pm/kiv/papers/verif-asms-english.pdf.

[60] J. Schmid. Executing ASM specifications with AsmGofer. Web pages at http://www.tydo.de/AsmGofer/, 1999.

[61] J. Schmid. Compiling Abstract State Machines to *C++*. In Moreno-Dìaz and Quesanda-Arencibia [52], pages 298–300. Extended Abstract.

[62] N. Shankar. Lazy compositional verification. In H. Langmaack, A. Pnueli, and W.-P. de Roever, editors, *Compositionality. The Significant Difference*, volume 1536 of *Lecture Notes in Computer Science*, pages 541–564. Springer-Verlag, 1998.

[63] R. F. Stärk and J. Schmid. Java bytecode verification is not possible. In Moreno-Dìaz and Quesanda-Arencibia [52], pages 232–234. Extended Abstract.

[64] R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine. Definition, Verification, Validation*. Springer-Verlag, 2001. See web pages at http://www.inf.ethz.ch/~jbook/.

[65] E. D. Thomas and R. P. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 2000.

[66] S. Thompson. *Haskell. The Craft of Functional Programming*. Addison-Wesley, second edition, 1999.

[67] T. Vullinghs, W. Schulte, and T. Schwinn. An introduction to TkGofer, 1996. Web pages at http://pllab.kaist.ac.kr/seminar/haha/tkgofer2.0-html/user.html.

[68] P. Wadler. The essence of functional programming. In *Proceedings of the* 19^{th} *Symposium on Principles of Programming Languages*, pages 1–14, 1992.

[69] P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, 1^{st} *International Spring School on Advanced Functional Programming Techniques*, pages 24–52. Springer-Verlag, 1995.

[70] J. Walton. Conway's Game of Life, 2000. Web pages at http://www.reed.edu/~jwalton/.

[71] A. Wikström. *Functional programming using Standard ML.* Prentice Hall, 1987.

[72] K. Winter. Model checking for Abstract State Machines. *Journal of Universal Computer Science*, 3(5):689–701, 1997.

[73] Q. Xu and M. Swarup. Compositional reasoning using the assumption-commitment paradigm. In W.-P. de Roever, H. Langmaack, and A. Pnueli, editors, *Compositionality. The Significant Difference*, volume 1536 of *Lecture Notes in Computer Science*, pages 565–583. Springer-Verlag, 1998.

[74] A. Zamulin. Object-oriented Abstract State Machines. In *Proceedings of the* 28^{th} *Annual Conference of the German Society of Computer Science.* Technical Report, Magdeburg University, 1998.