University of Groningen **Faculty of Mathematics and Physical Sciences**

Department of Computing Science

AFSK-D: A Specification Language with Object-Oriented State-Based Semantics

Victor Bos

begeleiders: G.R. Renardel de Lavalette
E.H. Saaman

december 1995

RuG

Contents

1 Introduction

In this report the specification language AFSK-D is described. AFSK-D is an abbreviation for "Almost Formal Specification Kernel - Dynamic". In a formal specification language one cannot use natural language. In AFSK-D one *can* use natural language and therefore it is called *Almost* formal. In this report the natural language features of AFSK-D are not discussed, so it is probably better to read AFSK-D as "A Formal Specification Kernel - Dynamic". AFSK-D is a kernel language and this means that it contains only a few different language constructs. The user language that corresponds with AFSK-D is called AFSL-D, i.e., *kernel* is replaced by *language*. At the moment, there is not yet a definition of AFSL-D, but it will be much more user-friendly than AFSK-D. AFSK-D is dynamic in the sense that it has constructs which can change the state.

The development of AFSK-D is part of the research done by the FSA (Formal System Analysis) project group at the university of Groningen. In this project several closely related languages are developed. The first language is called AFSL-0. This is a specification language based on first order predicate logic. AFSK-D is a successor of the kernel language AFSK-0 of AFSL-0. Another extension of AFSL is called AFSL-M. In this language a module mechanism is incorporated. One of the goals of the FSA project is to develop a general purpose dynamic specification language with a module mechanism.

For the development of AFSK-D the following guidelines where used:

Simple concepts: With this we mean that the underlying concepts of AFSL-D (and therefore also of AFSK-D) must be simple to learn. This is important, because in the end all languages developed in the FSA project are meant to be usable in practice.

Object-oriented: The methodology used in the FSA project is object-oriented. This means that the user languages, AFSL-0, AFSL-D and AFSL-M should have facilities to support the writing of object oriented specifications.

Programming Language: In the user language AFSL-D one must be able to give an *executable* specification. This means that a subset of the language should be a programming language.

As a consequence of the first guideline AFSK-D has only one syntactical class of terms. So, there are no expressions, formulas, sentences etc. However, when using AFSK-D, we noticed we defined different kinds of terms in our specifications, especially different kinds of operations (or: AFSK-D procedures). According to the differences between these operations we will describe a formal classification of operations, see chapter 6.

The semantics of AFSK-D gives a theoretical framework, which corresponds with the informal descriptions of object-orientation found in literature. Objects in AFSK-D specifications all have a local state and this state is a black box to the outside world. However, in the terminology of [Boo94] we cannot call AFSK-D object-oriented, for there is no typing and no inheritance in AFSK-D.

The third guideline is less important for AFSK-D, because it is a kernel language. However, the final user language, AFSL-D, should contain usual programming constructs like an assignment, a conditional and a loop-construct.

It was not our purpose to design a completely new language, but we have tried to reuse methods which had proven to be succesfull in other languages. The language COLD [FJ92, FJM94] has had great influence on the development of AFSK-D and, in fact, on all other languages of the FSA project. Other languages which have influenced AFSK-D are QDL (Quantified Dynamic Logic [Har84]), MLCM (Modal Logic of Creation and Modification [GdL93]), FLEA (Formal Language for Evolving Algebra [GdL95]), and, of course, AFSK-0.

AFSK-D is a kind of prototype of AFSL-D. This means that the development of AFSL-D could lead to modifications of AFSK-D. For example, the logic of AFSK-D, as described in this report, is three-valued, however, it is very likely that the logic of AFSL-D will be a two-valued logic.

The contents of this report is divided up into seven chapters. In the second chapter some preliminaries are described. This chapter is not very interesting, but necessary for rest of the report. Chapter 3 introduces AFSK-D in an informal way. For every language construct the intended meaning is described and a small example is given at the end of that chapter. The formal semantics of AFSK-D is given in chapter 4. That chapter can be seen as the main part of this report. In the last section of chapter 4 we compare AFSK-D with several languages which we have mentioned above. In chapter 5 we give some examples of specifications. This chapter illustrates how AFSK-D can be used. In chapter 6 we give a formal classification of operations. This chapter is less related to AFSK-D than the other chapters, but since the classification is a consequence of the way in which we have set up the semantics of AFSK-D, it is an interesting topic. In the last chapter we draw some conclusions and give some suggestions for further research.

2 Preliminaries

This chapter contains some definitions and some cpo-theory. For a more thorough treatment of cpo-theory we refer to [dB80].

Def 2.1 *Sequences are finite or infinite strings of elements.*

- *We denote a sequence* $x_0, x_1, \ldots$ *by the expression* $\overline{x}$.
- *Let* l *be a sequence, then* $(a : l)$ *is the sequence with head* a *and tail* l.
- *The empty sequence is denoted by* ϵ
- *Let* S *be a set. The set of all sequences over* S *is denoted by* S^ℓ.

Def 2.2 *Let* C *be an arbitrary set. A partial order* $\sqsubseteq$ *on* C *is a relation on* C *for which the following holds:*

- $\forall x \in C\ (x \sqsubseteq x)$ *(reflexivity)*
- $\forall x, y \in C\ (x \sqsubseteq y \wedge y \sqsubseteq x \ \Rightarrow\ x = y)$ *(antisymmetry)*
- $\forall x, y, z \in C\ (x \sqsubseteq y \wedge y \sqsubseteq z \ \Rightarrow\ x \sqsubseteq z)$ *(transitivity)*

The pair $(C, \sqsubseteq)$ *is called a partially orded set.*

Def 2.3 *Let* $(C, \sqsubseteq)$ *be a partially ordered set and* $X \subseteq C$.

- *If the element* $z \in C$ *defined by:*
  - $\forall x \in X\ (x \sqsubseteq z)$,
  - $\forall y \in C\ ((\forall x \in X\ (x \sqsubseteq y)) \ \Rightarrow\ z \sqsubseteq y)$

  exists, we can show that it is unique and we call z *the least upperbound of* X. *The least upper bound of* X *is denoted by:* $\sqcup X$.
- *Let* $\overline{x}$ *be a sequence* $x_0, x_1, \ldots$. *Define* $X = \{x \mid \exists i \in 0, 1, \ldots\ (x_i = x)\}$. *Then if* $\sqcup X$ *exists we define the least upperbound of* $\overline{x}$ *by:* $\sqcup \overline{x} = \sqcup X$

A special class of partially ordered sets are the *complete partially ordered sets.* In order to define this class, we have to introduce the concept of a *chain* on a partially orderd set.

Def 2.4 *Let $(C, \sqsubseteq)$ be a partially ordered set. A chain on $(C, \sqsubseteq)$ is a sequence $\overline{x} = x_0, x_1, \ldots$ such that: for $i = 0, 1, \ldots$ we have $x_i \sqsubseteq x_{i+1}$*

Lemma 2.5 *Let $\overline{x}$ be an infinite chain on $(C, \sqsubseteq)$ with $\sqcup\overline{x}$ as its least upperbound. Define for some $k \in \mathbb{N}$ the sequence $\overline{y}$ by: $y_i = x_{i+k}$. Then $\sqcup\overline{y} = \sqcup\overline{x}$.*

Proof
First we have:

$$\begin{array}{ll} & y_i \sqsubseteq \sqcup\overline{x} \\ = & \{ \text{ definition of } y_i \} \\ & x_{i+k} \sqsubseteq \sqcup\overline{x} \\ = & \\ & \text{True} \end{array}$$

So, $\sqcup\overline{x}$ is an upperbound of $\overline{y}$. Next we show that $\sqcup\overline{x}$ is the least upperbound of $\overline{y}$. Let z be some upperbound of $\overline{y}$, then:

$$\begin{array}{ll} & \forall i(y_i \sqsubseteq z) \\ = & \{ \text{ let } k \in \mathbb{N} \text{ and use definition of } y_i \} \\ & \forall i(x_{i+k} \sqsubseteq z) \\ = & \{ \text{ property of chains: } (i \leq j) \Rightarrow (x_i \sqsubseteq x_j) \} \\ & \forall x_i(x_i \sqsubseteq z) \\ = & \\ & \sqcup\overline{x} \sqsubseteq z \end{array}$$

This completes the proof.

Def 2.6 *A complete partially ordered set (cpo) is a partially ordered set $(C, \sqsubseteq)$ for which:*

- *There is a least element with respect to $\sqsubseteq$, i.e. an element $\bot_C \in C$ such that $\bot_C \sqsubseteq x$ for all $x \in C$.*
- *For each chain $\overline{x}$ in C the least upper bound $\sqcup\overline{x} \in C$ exists .*

Lemma 2.7 *Let C_1 be a set and $(C_2, \sqsubseteq_2)$ be a cpo. Define the partial order $\sqsubseteq$ on the set of total functions $C_1 \to C_2$ by: $(f \sqsubseteq g \Leftrightarrow (\forall x \in C_1(f(x) \sqsubseteq_2 g(x))))$ Then $(C_1 \to C_2, \sqsubseteq)$ is a cpo.*

Proof
First we show that $\sqsubseteq$ is reflexive, antisymmetric and transitive. Let $f \in C_1 \to C_2$, then:

$$
\begin{array}{ll}
 & f \sqsubseteq f \\
= & \\
 & \forall x \in C_1(f(x) \sqsubseteq_2 f(x)) \\
= & \{ \sqsubseteq_2 \text{ is reflexive}\} \\
 & \text{True}
\end{array}
$$

Let $f, g \in C_1 \to C_2$, then:

$$
\begin{array}{ll}
 & (f \sqsubseteq g) \wedge (g \sqsubseteq f) \\
= & \\
 & \forall x \in C_1((f(x) \sqsubseteq_2 g(x)) \wedge (g(x) \sqsubseteq_2 f(x))) \\
= & \{ \sqsubseteq_2 \text{ is antisymmetric}\} \\
 & \forall x \in C_1(f(x) = g(x)) \\
= & \\
 & f = g
\end{array}
$$

Let $f, g, h \in C_1 \to C_2$ and $f \sqsubseteq g$ and $g \sqsubseteq h$, then:

$$
\begin{array}{ll}
 & f \sqsubseteq h \\
= & \\
 & \forall x \in C_1(f(x) \sqsubseteq_2 h(x)) \\
= & \{f \sqsubseteq g \text{ and } g \sqsubseteq h\} \\
 & \forall x \in C_1((f(x) \sqsubseteq_2 h(x)) \wedge (f(x) \sqsubseteq_2 g(x)) \wedge (g(x) \sqsubseteq_2 h(x))) \\
= & \{\sqsubseteq_2 \text{ is transitive}\} \\
 & \forall x \in C_1((f(x) \sqsubseteq_2 g(x)) \wedge (g(x) \sqsubseteq_2 h(x))) \\
= & \\
 & \text{True}
\end{array}
$$

Define $f_0 \in C_1 \to C_2$ by: $f_0(x) = \bot_{C_2}$. f_0 is the least element in $C_1 \to C_2$ with respect to $\sqsubseteq$, because for any $g \in C_1 \to C_2$ we have:

$$
\begin{array}{ll}
 & f_0 \sqsubseteq g \\
= & \\
 & \forall x \in C_1(f_0(x) \sqsubseteq_2 g(x)) \\
= & \{ \text{ definition of } f_0\} \\
 & \forall x \in C_1(\bot_{C_2} \sqsubseteq_2 g(x)) \\
= & \{ \bot_{C_2} \text{ is least element in } C_2\} \\
 & \text{True}
\end{array}
$$

Let $\overline{f} = f_0, f_1, \ldots$ be a chain in $C_1 \to C_2$. Then for any $x \in C_1$ $f_0(x), f_1(x), \ldots$ is a chain in C_2. Denote this chain by $\overline{f_i(x)}$ and the least upperbound of this chain by $\sqcup\overline{f_i(x)}$. Define $\mathbf{f} \in C_1 \to C_2$ by: $\mathbf{f}(x) = \sqcup\overline{f_i(x)}$. Then we have:

$$
\begin{array}{ll}
 & f_i \sqsubseteq \mathbf{f} \\
= & \\
 & \forall x \in C_1(f_i(x) \sqsubseteq_2 \mathbf{f}(x)) \\
= & \{ \text{ definition of } \mathbf{f} \ \} \\
 & \forall x \in C_1(f_i(x) \sqsubseteq_2 \sqcup\overline{f_i(x)}) \\
= & \{ \text{ definition of } \sqcup\overline{f_i(x)}\} \\
 & \text{True}
\end{array}
$$

Furthermore, if $g \in C_1 \to C_2$ and for all f_i in the sequence $\overline{f}$ we have $f_i \sqsubseteq g$ then we also have:

$$
\begin{array}{ll}
 & \mathbf{f} \sqsubseteq g \\
= & \\
 & \forall x \in C_1(\mathbf{f}(x) \sqsubseteq_2 g(x)) \\
= & \{ \text{ definition of } \mathbf{f}\} \\
 & \forall x \in C_1(\sqcup\overline{f_i(x)} \sqsubseteq_2 g(x)) \\
= & \{ \forall f_i \ (f_i \sqsubseteq g)\} \\
 & \text{True}
\end{array}
$$

This shows that $\mathbf{f} = \sqcup\overline{f_i}$.

Lemma 2.8 *Let $\overline{f} = f_0, f_1, \ldots$ be a chain of functions in $(C_1 \to C_2, \sqsubseteq)$. Define the chain $\overline{f_i(x)}$ as in lemma 2.7. Then $\forall x \in C_1((\sqcup\overline{f})(x) = \sqcup f_i(x))$.*

Proof
This follows directly from definition of $\mathbf{f}$ as in the proof of lemma 2.7 and the result $\mathbf{f} = \sqcup\overline{f_i}$ of lemma 2.7.

3 Introduction to AFSK-D

In this chapter we will give the syntax of AFSK-D. Besides that we will say something about the intended meaning of the language constructs. To illustrate this we will give some small examples.

In section 3.1 the syntax of AFSK-D will be given in BNF-like notation. Section 3.2 is about the intended meaning of the language constructs. In that section we will informally describe some concepts which will be formalized in chapter 4. Finally, in section 3.3 we will give an example of a simple specification.

3.1 Syntax

A specification[1] is a set of terms. Every specification has a signature. This is the set of user-defined procedure and function names.

The set of all procedure and function names is called PROC. Examples of procedure names are `Assign` (see section 5.1) and `Read` (see section 5.3). Arbitrary elements of PROC are denoted by p, p_1, p_2,... PROC has a subset FUNC of all *function names*. Examples of function names are `Val` (see section 5.1) and `Fptr` (see section 5.3). Arbitrary elements of FUNC are denoted by f, f_1, f_2,... We have to mention that there is no syntactical difference between function and procedure names. Informally, however, we think of procedures names as denoting operations with side-effects, whereas function names denote operations without side-effects.

We assume there is a set VAR of variables. Elements of VAR are *logical* variables. Logical variables should not be confused with *programming* variables. One could say that logical variables are *dummies*, since they are used at places where one needs an arbitrary object. We will denote elements of VAR by d. In section 5.1 we give a specification of programming variables. There we will say more about the difference between dummies and programming variables.

In definition 3.1 we have given the syntax of AFSK-D. It follows from this definition that TERM $\subseteq$ PROG. So, terms and dynamic terms belong to the same syntactical class. In fact, there is only one syntactical class in AFSK-D. The reason for this is that it simplifies the semantics of AFSK-D, since we only need to describe the meaning of one sort of syntactic entities. The reason why we have defined two sets PROG and TERM, instead of only one set PROG, is that it is very natural to distinguish dynamic terms from "normal", or static, terms. The same argument can be used to explain why we have introduced both function and procedure names, whereas syntactically there is no difference between them.

[1] With a specification we mean an AFSK-D-specification unless explicitly stated otherwise.

Def 3.1 **The syntax of AFSK-D** *In this definition we have* $n \in \mathbb{N}$.

- TERM *is a set of static* AFSK-D*-terms. Elements of* TERM *are denoted by the metavariable* t*, possibly indexed. The syntax of terms is defined as follows:*

$$
\begin{array}{rcl}
t & ::= & d \\
& | & \texttt{TRUE} \\
& | & \texttt{FALSE} \\
& | & \texttt{UNDEF} \\
& | & !\tau \\
& | & (\tau_1 = \tau_2) \\
& | & \texttt{NOT } (t) \\
& | & (t_1 \texttt{ OR } t_2) \\
& | & (\texttt{EXISTS } d\ t) \\
& | & f(t_1, \ldots, t_n) \\
& | & \texttt{LET } d = t_1 \texttt{ IN } t_2 \texttt{ END} \\
& | & t' \\
& | & [d := \tau]\ t \\
& | & (\tau_0 \texttt{ MODSTATE } \tau_1, \ldots, \tau_n) \\
& | & (\tau \texttt{ MODRESULT } p_1, \ldots, p_n) \\
& | & (\tau_0 \texttt{ DEPSTATE } \tau_1, \ldots, \tau_n) \\
& | & (\tau \texttt{ DEPRESULT } p_1, \ldots, p_n)
\end{array}
$$

- PROG *is a set of dynamic* AFSK-D*-terms. Elements of* PROG *are denoted by the metavariable* τ*, possibly indexed. The syntax of dynamic terms is defined as follows:*

$$
\begin{array}{rcl}
\tau & ::= & t \\
& | & \texttt{LET } d = \tau_1 \texttt{ IN } \tau_2 \texttt{ END} \\
& | & p(\tau_1, \ldots, \tau_n) \\
& | & \texttt{IF } \tau_0 \texttt{ THEN } \tau_1 \texttt{ ELSE } \tau_2 \texttt{ FI} \\
& | & (\tau_1 ; \tau_2) \\
& | & \texttt{WHILE } \tau_0 \texttt{ DO } \tau_1 \texttt{ OD}
\end{array}
$$

3.2 Intended Meaning

We will first describe the structures we use as models for a specification. Each structure has a set of objects which is called the *universe* of the structure. Every object has its own *local* state. The local state of an object is a black box to the rest of the world. The only way to get information about the local state of an object is via specified operations on objects. With this view of objects and local states we stay within the object-oriented paradigm.

The universe of every structure is never empty, since we demand that it contains at least the truth values. In AFSK-D there are three truths values: **T**, **F** and $\perp$. The value $\perp$ is the *undefined* value. This object is used to model partial functions.

A *global state* is a (total) function from objects to local states. Every structure has a set of global states. This set of global states restricts both the set of possible local states of a specific object and the possible combinations of objects and local states. For example, suppose we have a structure with the objects u_1, u_2, local states s_1, s_2, s_3, s_4, s_5 and global states $\{u_1 \mapsto s_1, u_2 \mapsto s_4\}$, $\{u_1 \mapsto s_2, u_2 \mapsto s_4\}$ and $\{u_1 \mapsto s_3, u_2 \mapsto s_5\}$. In this structure the set of possible local states for u_1 is $\{s_1, s_2, s_3\}$ and for u_2 $\{s_4, s_5\}$. Furthermore, even though s_3 is a possible local state of u_1 and s_4 of u_2 the combination of u_1 with local state s_3 and u_2 with local state s_4 is impossible.

Operations are functions from a tuple of objects (possibly the empty tuple) and a global state to an object and a global state. The operations on objects can change the global state. This means that some objects' local states are changed by the execution of an operation. Function and procedure names are interpreted with the aid of a so called *procedure name interpretation function.* This is a function which maps every function and procedure name to an operation.

Note that the dynamics of a structure is internal to that structure. By this we mean that state changes do *not* change the algebra. Therefore, we do not use a "states as algebras" view as is done in [FJ92], [GdL93] or [GdL95].

Thus a structure has a (non-empty) set of objects, a set of local states, a set of global states and a procedure name interpretation function. In definition 4.1 this is formalized.

A specification is a set of terms. All the terms in a specification should be read as axioms, with this we mean that all the terms must be true, **T**, in some structure. When there isn't a structure such that the all the terms are true, the specification is inconsistent. The structures in which all the terms are true are called *models* of the specification. To give an interpretation to a specification one must assign an object to every free variable occurring in the specification. This is done by an variable valuation function.

We will now informally describe the meaning of all the AFSK-D-terms. Note that the meaning of term can be divided in two parts. The first part is a function

which yields an object. This part is called the *object-interpretation* of a term. The second part is a function which yields a global state. This part is called the *state-interpretation* or of a term. Both functions take a tuple of objects and a global state as input parameters.

Static terms are terms which don't change the state. This means that their state-interpretation is a function from a tuple of objects and a global state to a global state, such that the input global state is the same as the output global state. Since static terms all have the same state-interpretation, we will say nothing more about it in the rest of this section.

The following terms are called *logical* terms, because their object-interpretation is a truth-value.

TRUE	(EXISTS d t)
FALSE	[d := τ] t
UNDEF	(τ_0 MODSTATE $\tau_1, \ldots, \tau_n$)
!τ	(τ MODRESULT $p_1, \ldots, p_n$)
(τ_1 = τ_2)	(τ_0 DEPSTATE $\tau_1, \ldots, \tau_n$)
NOT (t)	(τ DEPRESULT $p_1, \ldots, p_n$)
(t_1 OR t_2)	

Since the logic of AFSK-D is a three-valued logic, there are three truth-values: **T** (*true*), **F** (*false*) and $\perp$ (*undefined*), which in AFSK-D are denoted by TRUE, FALSE and UNDEF, respectively.

When one incorporates a third truth-value, one has many possibilities to define the logical connectives. The specific choice that was made for AFSK-D is not yet formally documented. We will, however, try to make our choice a bit plausible by the following points:

- The truth-values are ordered: **T** is the greatest and **F** the smallest truth-value. $\perp$ lies between **T** and **F**: $\mathbf{F} < \perp < \mathbf{T}$
- The connective OR yields the maximum of its arguments.
- The connective NOT "mirrors the order", i.e., **T** is mapped to **F** and **F** to **T**.
- The other connectives, see below, are defined in terms of OR and NOT.

Since these points all hold for ordinary two-valued first order predicate logic, we think they can be used safely for the three-values logic of AFSK-D. Again, however, there are more possibilities, some of which could be interesting for AFSK-D. In this report the three-valuedness of AFSK-D will not be further investigated.

The interpretation of the propositional connectives NOT and OR is described by the following truth-tables. In these tables the symbol * can be any (semantic) value except **T** or **F**. This means that in AFSK-D the logical terms treat everything other than **T** and **F** as $\perp$.

term	is an abbreviation for
t_1 AND t_2	NOT(NOT(t_1) OR NOT(t_2))
t_1 ==> t_2	NOT(t_1) OR t_2
t_1 <=> t_2	(t_1==>t_2) AND (t_2==>t_1)
(FORALL d t)	NOT(EXISTS d NOT(t))
[τ] t	[x := τ] t (*)

(*) Where $x \in$ VAR does not occur in τ

Table 3.1: Abbreviations for some AFSK-D terms

NOT	T	F	*
	F	T	$\perp$

OR	T	F	*
T	T	T	T
F	T	F	$\perp$
*	T	$\perp$	$\perp$

The existential quantification (EXISTS d t) is **T** whenever there is a value $x \neq \perp$, such that mapping d onto x will make the interpretation of the term t **T**. Note that the quantified variable d only ranges over defined values.

In the examples given below we will also use the logical connectives AND, ==>, <=> and FORALL. These connectives are defined in table 3.1 as syntactic abbreviations.

The term [d := τ] t resembles the *necessity-term* from dynamic logic, see for example [vBvDKMV94] (Dutch) or [Har84]. [d := τ] t is **T**, whenever t is **T** in the state resulting from the evaluation of τ. Whenever t is **F** in this state the meaning of [d := τ] t is **F**. In all other cases the meaning is $\perp$. Note that the evaluation of τ need not terminate. In this case [d := τ] t is undefined: $\perp$. This is different than in dynamic logic, because there the formula $[\pi]\phi$ is always true when π isn't terminating.

Another aspect of [d := τ] t is that the global state in which τ is executed is remembered, which makes it possible to refer to this state somewhere in the term t. To refer to this *previous* state, one decorates subterms of t with a prime: ' as is explained somewhere at the end of this section. The formalisation of "remembering the previous state" is done by using a stack-mechanism, as can be seen in the next chapter.

The variable d in the term [d := τ] t can be used to refer in t to the result of τ. Before evaluating t, the valuation is changed such that d has the value returned by τ. Although this term contains an assignment, it is a restricted form of an assignment. Its effect can only be noticed in the term t. The reason for doing this is that a general assignment to variables would enable the user to use dummies as programming variables.

The term [d := τ] t can be used to describe procedures. For example, we could write:

```
(Value(i)=k) ==> ([x := Incr(i)] (Value(i) = k+1))
```

Here we specify that the procedure Incr increases the integer programming-variable with one. However, we do not specify that Incr doesn't do anything else. So, an implementation of Incr could be a procedure that increases the given programming-variable, but which also resets all other programming-variables to 0. This implementation doesn't contradict the specification.

The problem is that we've only specified what the procedure is supposed to do, not what is isn't supposed to do. Specifying what a procedure isn't supposed to do is sometimes called *framing* a procedure. In AFSK-D we have two language constructs with which we can frame a procedure:

$(\tau_0$ MODSTATE $\tau_1, \ldots, \tau_n)$
$(\tau$ MODRESULT $p_1, \ldots, p_n)$

The first term says that τ_0 can only change the local states of $\tau_1, \ldots, \tau_n$ and no other local states. We could use this language construct in our example and add the following term to the specification:

```
Incr(i) MODSTATE i
```

Now the procedure Incr can only change the local state of its actual parameter and therefore an implementation of Incr in which all other variables are reset to 0 would violate the specification.

Suppose i isn't just a programming variable, but an object which has more features than just a Value. For example, it could have a name which we would like to refer to with the term Name(i). Now, Value is not the only function which takes i as an argument and therefore a change of the object i could affect other functions which depend on their actual parameter as well as Value. In order to specify which functions are affected by the execution of some procedure, AFSK-D has the language construct MODRESULT. For example, we could add the following term to our specification:

```
Incr(i) MODRESULT Value
```

Now the procedure Incr can only change its actual parameter in such a way that only applications of the function Value can yield different values than before. So, we no can safely say that Incr(i) cannot change the name of i. In fact, every application of any function other than Value will yield the same result as before the execution of Incr(i).

The language constructs MODSTATE and MODRESULT are very useful in framing a procedure. Note, however, that we could combine these construct into one construct, MOD say. To do this we first note that after MODSTATE a finite sequence of terms is expected and after MODRESULT a finite sequence of procedure names (or function names) is expected. So, instead of the MODSTATE and the MODRESULT terms we've added to our example, we could add the (non-AFSK-D) term:

```
Incr(i) MOD i, Value
```

The reason for using two different language constructs is that it is important to notice the difference between their interpretations. The construct MODSTATE is used to specify which local states may be changed by some procedure. It doesn't say anything about the possible effects the execution of the procedure could have. The construct MODRESULT, however, is used to specify the possible effects of executing a procedure. It doesn't say anything about which local states are allowed to be changed by the execution of the procedure.

AFSK-D has two language constructs with which dependencies between objects and functions can be specified. The first is called DEPSTATE. This construct is used in terms of the form: (τ_0 DEPSTATE $\tau_1, \ldots, \tau_n$). With this term we specify that τ_0 only depends on the local states of $\tau_1, \ldots, \tau_n$. This means that the local state of τ_0 can *only* be changed if some of the local states of the objects denoted by $\tau_1, \ldots, \tau_n$ are changed (by the execution of some procedure).

We will illustrate DEPSTATE with the example of integer programming variables given above. Suppose these programming variables and the functions and procedures defined on them are part of a bigger system. Then there must be a way to specify that the Name and the Value of a programming variable only depend on the local state of that programming variable. Otherwise, it would be possible to give an implementation in which as a result of changing the local state of some other object, the name of a programming variable is changed too. The following term will forbid such an implementation:

```
Name(i) DEPSTATE i
```

Since the same holds for the function Value, we also add the term:

```
Value(i) DEPSTATE i
```

to our specification. Now the only way of changing the result of the functions Name and Value is by changing the local state of the object i.

Besides the DEPSTATE, construct AFSK-D has the construct DEPRESULT to specify dependencies. The construct is used as in (τ DEPRESULT $p_1, \ldots, p_n$). This term says that τ is dependent on the procedures $p_1, \ldots, p_n$ in the following way. The object-interpretation of τ can only be changed by the execution of some procedure when at least one of the applications of p_i ($1 \leq i \leq n$) is changed by that procedure as well. For example, we could have a function MaxVar which returns the programming variable with the greatest value. This function depends on the function Value, because only if some application of Value is changed by some procedure, the result of MaxVar *could* be changed. In order to specify this behavior of MaxVar we add the following term to our specification:

```
MaxVar() DEPRESULT Value
```

Note that, according to the AFSK-D syntax, the empty parameter list is necessary. Usually it isn't necessary to add a DEPRESULT-term, because the dependency which is to be specified can be defined directly in terms of other functions. For example, the DEPRESULT-term in our example could be replaced by the term:

```
(FORALL d (Value(d) =< MaxVar())) AND (EXISTS d (Value(d) = MaxVar()))
```

However, when it isn't obvious that the dependency can be defined in such a direct way, the construct DEPRESULT can be useful.

Besides the logical terms discussed above, AFSK-D has the following non-logical terms:

d
$f(t_1, \ldots, t_n)$
$p(\tau_1, \ldots, \tau_n)$
LET d = t_1 IN t_2 END
LET d = τ_1 IN τ_2 END
t'
IF τ_0 THEN τ_1 ELSE τ_2 FI
$(\tau_1 ; \tau_2)$
WHILE τ_0 DO τ_1 OD

Sometimes some of these terms can be viewed as logical terms. For example, the variable d is a logical term if it denotes **T**, **F** or $\perp$. Generally, a term is a logical term whenever its object-interpretation is a truth-value.

The term d doesn't have side effects. This means that it doesn't change the state. The object-interpretation of variables depends on the variable valuation function. As a consequence, this interpretation can not be changed by state-changes. So, whenever a variable denotes a certain object, then no execution of whatever procedure can let it denote another object. The only way to change the object-interpretation of a variable is by one of the following terms:

- (EXISTS d t)
- LET d = τ_1 IN τ_2 END
- [d := τ] t

The first and the last of these terms are discussed already. The other two terms will be discussed below. A point to note here is that the changes of the variable valuation function established by these four terms are local to these terms. So, after the execution of one of these terms, the original variable valuation is reinstalled. This prohibits one to use variables as programming variables.

The term $f(t_1, \ldots, t_n)$ is a special case of $p(\tau_1, \ldots, \tau_n)$, because function names are a subset of procedure names and static terms are special kinds of dynamic terms. The term $p(\tau_1, \ldots, \tau_n)$ is a procedure application. Note that the parameter list can be empty, in that case the term looks like $p()$. The interpretation of

this term depends on the interpretations of the arguments $\tau_1, \ldots, \tau_n$ and on the procedure name interpretation function. The arguments are evaluated from left to right and their side-effects are passed through to each other in the same order. This means that the term τ_{i+1} is evaluated in the state resulting from the evaluation of τ_i. When all the terms are evaluated the procedure name interpretation function determines the result of the application. This functions has as arguments a procedure name, a tuple of objects and a global state. The tuple of objects consists of the object-interpretations of the terms $\tau_1, \ldots, \tau_n$ and the state-interpretation of τ_n is the global state.

The term LET d = t_1 IN t_2 END is a special case of LET d = τ_1 IN τ_2 END. The object-interpretation of this term is the object-interpretation of τ_2 executed in the state-interpretation of τ_1 and with the variable valuation modified in d. The variable d is modified to denote the object-interpretation of τ_1.

The state-interpretation of the complete term is the state-interpretation of τ_2 evaluated in the state-interpretation of τ_1.

Static terms can be decorated with a prime: t'. The object-interpretation of t' is the object-interpretation of t evaluated in the *previous* state. The previous state is the state just before the execution of a particular operation. The only term in which the previous state is remembered, is $[d := \tau]\ t$. Therefore, decorating a term with a prime only makes sense in the subterm t of a term $[d := \tau]\ t$. For example, we could specify the procedure Incr as follows:

```
[Incr(i)] (Value(i) = (Value(i))' + 1)
```

instead of using an extra variable as was done on page 13. Note that decorating a variable with a prime has no effect. As said before, variables don't depend on the state, only on the variable valuation function. So, if d is a variable denoting some object, d', d'' and d''''' denote the same object.

The if-then-else term IF τ_0 THEN τ_1 ELSE τ_2 FI behaves as one would expect. Note however, that the value of τ_0 must be **T** or **F**, otherwise the result will be $\perp$. Since $\tau_0 \in$ PROG it can have side-effects. This means that the state in which τ_1 or τ_2 is evaluated is the state resulting from the evaluation of τ_0.

The sequential composition of terms, $(\tau_1 ; \tau_2)$, does have, besides a possible state transition, a value. This is the value of τ_2 evaluated in the state resulting from the evaluation of τ_1. The result state of $(\tau_1 ; \tau_2)$ is the state resulting from the evaluation of τ_2 in the result state of τ_1.

The WHILE τ_0 DO τ_1 OD is only evaluated for its side effects. Its value is always the same: $\perp$. The term WHILE τ_0 DO τ_1 OD is a loop-construct: whenever the term τ_0 is **T**, the term τ_1 will be evaluated and then the term τ_0 will be evaluated again.

```
% The number zero
IsNat(Zero()) = True

% The successor function
(IsNat(n) AND IsNat(m)) ==> ((IsNat(Succ(n))) AND
                             ((Succ(n) = Succ(m)) ==> (n = m)))

% The number one is defined for convenience:
One() = Succ(Zero())

% Addition of natural numbers
(IsNat(n) AND IsNat(m)) ==> ((Plus(n,Zero()) = n) AND
                             (Plus(n,Succ(m)) = Succ(Plus(n,m))))

% Multiplication of natural numbers
(IsNat(n) AND IsNat(m)) ==> ((Mult(n,Zero()) = Zero()) AND
                             (Mult(n,Succ(m)) = Plus(n,Mult(n,m))))
```

Figure 3.1: Natural Numbers

3.3 Example: Natural Numbers

In this section a specification of the natural numbers is given. Since we would like to demonstrate dynamic AFSK-D features as wel as static features, we also add the integer programming variables and some procedures to increase or decrease the value of an variable. The programming variables were used as an example in the previous section.

In figure 3.1 the static part of the specification is given. The function IsNat is the characteristic function for natural numbers. We have to use such a characteristic function, because AFSK-D doesn't have a typing-mechanism. The function Zero is the number 0. Other naturals are defined in terms of Zero and the function Succ, as usual.

We use some conventions in our examples. Function names and Procedure names begin with a capital. Variables (= dummies) start with a lower case letter. The outermost parentheses are left out. However, in an application of a procedure or a function without parameters, the empty parameter list, (), is not left out. This helps distinguishing variables and functions or procedures. Besides these conventions we use indentation to get a clear layout. Furthermore, the terms of a specification are seperated by at least one blank line. Lines starting with the % sign are comments. These lines are not part of the specification, but contain extra information for the user.

The programming variables and the procedures are given in figure 3.2

```
% Programming variables have a value (a natural number):
IsVar(i) ==> IsNat(Value(i))

% Programming variables can be assigned to:
IsVar(i) AND IsNat(n) ==> ([AssignVar(i,n)] Value(i) = n)

% Increasing/decreasing a variable by 1:
IsVar(i) ==> (([Incr(i)] (Value(i) = Plus(Value(i)',One()))) AND
              (NOT(Value(i)=Zero()) ==>
               ([Decr(i)] (Plus(Value(i)',One())=Value(i)))))

% Framing the procedure AssignVar:
(AssignVar(i,n) MODSTATE i) AND (AssignVar(i,n) MODRESULT Value)

% Framing Incr and Decr:
(Incr(i) MODSTATE i) AND (Incr(i) MODRESULT Value)

(Decr(i) MODSTATE i) AND (Decr(i) MODRESULT Value)

% Dependencies:
(Value(i) DEPSTATE i)
```

Figure 3.2: Integer Programming Variables

4 Semantics of AFSK-D

In this chapter we will give the formal semantics of AFSK-D. In the first section we will describe the mathematical structures which we use to give an interpretation to terms and specifications. Section 4.2 describes the semantics of a term. In section 4.3 the semantics of a specification is given. Finally section 4.4 describes some differences between AFSK-D and other specification languages and formalisms.

4.1 Structures

Before we can give the interpretation of a specification, we first have to define what the interpretation of a term is. Terms are interpreted in a *structure*, which we have described informally in the previous chapter.

As mentioned before, AFSK-D has a three-valued logic. The set of the three truth-values $\mathbf{T}, \perp$ and $\mathbf{F}$ is called $\mathbb{B}$ (booleans).

Def 4.1 *A Structure is a tuple* $(\mathrm{U}, \mathrm{LS}, \mathrm{GS}, \mathrm{P})$, *such that*

- U *and* LS *are sets,*
- $\mathbb{B} \subseteq \mathrm{U}$,
- $\mathrm{GS} \subseteq (\mathrm{U} \rightarrow \mathrm{LS})$,
- $\mathrm{P} \in (\mathrm{PROC} \rightarrow (\mathrm{U}^* \times \mathrm{GS} \dot{\rightarrow} \mathrm{U} \times \mathrm{GS}))$,

The set U is the *universe* of the structure. It contains *objects*. The set $\mathbb{B}$ is a subset of the universe of every structure. So, the universe of a structure is never empty. The set LS contains the *local states* of the objects. The set GS is a set of total functions from U to LS, these functions are *called global* states. P is called the *procedure name interpretation function.*

An *operation* in a structure is a function with type: $\mathrm{U}^* \times \mathrm{GS} \dot{\rightarrow} \mathrm{U} \times \mathrm{GS}$ So, the function P assigns to every procedure name an operation. Note that an operation is a partial function, which is denoted by the dot above the arrow. In order to make every operation a total function we introduce a new symbol: †:

Let † be a new symbol not occurring in the structure $(\mathrm{U}, \mathrm{LS}, \mathrm{GS}, \mathrm{P})$.

- $\underline{\mathrm{GS}} = \mathrm{GS} \cup \{\dagger\}$,
- For every operation $o \in \mathrm{U}^* \times \mathrm{GS} \dot{\rightarrow} \mathrm{U} \times \mathrm{GS}$ define

$$\begin{array}{lcl} \underline{o} & \in & \mathrm{U}^* \times \underline{\mathrm{GS}} \to \mathrm{U} \times \underline{\mathrm{GS}} \\ \underline{o}(\overline{u}, g) & = & \begin{cases} o(\overline{u}, g) & \text{if } g \neq \dagger \text{ and } o(\overline{u}, g) \text{ is defined} \\ \dagger & \text{otherwise} \end{cases} \end{array}$$

- $\underline{\mathrm{P}}(p) \stackrel{\text{def}}{=} \underline{\mathrm{P}(p)}$, $\underline{\mathrm{P}}$ is the extension of P.

For functions with results in $\mathrm{U} \times \mathrm{GS}$ or $\mathrm{U} \times \underline{\mathrm{GS}}$ we use the superscripts ^{u} and ^{s} to denote the first and second projections respectively. For example, we have:

$$\underline{\mathrm{P}}(p)(\overline{u}, g) = (\underline{\mathrm{P}}^u(p)(\overline{u}, g), \underline{\mathrm{P}}^s(p)(\overline{u}, g))$$

We define two classes of equality predicates on functions from the set $\underline{\mathrm{GS}}$.

Def 4.2 *Let* $\mathrm{V} \subseteq \mathrm{U}$ *and* $\mathrm{B} \subseteq \mathrm{PROC}$ *then:*

$$g =_{\mathrm{V}} g' \stackrel{\text{def}}{\Leftrightarrow} (\forall u \in \mathrm{V}\ (g(u) = g'(u)) \vee (g = \dagger \wedge g' = \dagger))$$

$$g =_{\mathrm{B}} g' \stackrel{\text{def}}{\Leftrightarrow} \forall p \in \mathrm{B}, \overline{u} \in \mathrm{U}^*\ (\underline{\mathrm{P}}^u(p)(\overline{u}, g) = \underline{\mathrm{P}}^u(p)(\overline{u}, g'))$$

4.2 Semantics of Terms

The variables that can occur in a term are interpreted with the aid of a variable valuation. This is a function that assigns an object to every variable.

Def 4.3 $\mathrm{VAL} = (\mathrm{VAR} \to \underline{\mathrm{U}})$ *is the set of variable valuations.*

Variable valuations are denoted by v. The expression $v[d \mapsto u]$ is a pointwise modification of v, which is defined by:

$$v[d \mapsto u](d') = \begin{cases} u & \text{if } d = d' \\ v(d') & \text{if } d \neq d' \end{cases}$$

Let S be a set. The set of all sequences over S is denoted by S^ℓ, see definition 2.1. So, for all $\overline{x} = x_1, x_2, \ldots$ with $x_i \in S$ we have $\overline{x} \in S^\ell$. The empty sequence is denoted by ε. Note that S^ℓ can contain infinite sequences, therefore S^ℓ should not be confused with the set S^* of all tuples over S.

Terms are interpreted with the aid of a structure $(\mathrm{U}, \mathrm{LS}, \mathrm{GS}, \mathrm{P})$, a new symbol $\dagger$ not occuring in this structure and a variable valuation v. We use the notation $\underline{\mathrm{GS}}$ and $\underline{\mathrm{P}}$ as defined above.

Def 4.4 *The meaning function $\mathcal{M}$ which gives the interpretation of* AFSK-D *terms is defined by:*

$$\begin{array}{lcl}
\mathcal{M} & \in & \textbf{TERM} \times \textbf{VAL} \times \underline{\textbf{GS}}^{\ell} \to \textbf{U} \times \underline{\textbf{GS}} \\
\mathcal{M}(\tau, v, l) & = & (\mathcal{M}^{u}(\tau, v, l), \mathcal{M}^{s}(\tau, v, l)) \\
\mathcal{M}^{u} & \in & \textbf{TERM} \times \textbf{VAL} \times \underline{\textbf{GS}}^{\ell} \to \textbf{U} \\
\mathcal{M}^{s} & \in & \textbf{TERM} \times \textbf{VAL} \times \underline{\textbf{GS}}^{\ell} \to \underline{\textbf{GS}} \\
\mathcal{M}^{u}(t, v, (\dagger{:}l)) & = & \bot \\
\mathcal{M}^{s}(t, v, (\dagger{:}l)) & = & \dagger \\
\mathcal{M}^{u}(t, v, \varepsilon) & = & \bot \\
\mathcal{M}^{s}(t, v, \varepsilon) & = & \dagger
\end{array}$$

We will now give the remaining definitions for $\mathcal{M}^{u}$ and $\mathcal{M}^{s}$.

Def 4.5 *In this definition we assume that $g \neq \dagger$. For all the terms $t \in$ TERM we define: $\mathcal{M}^{s}(t, v, g{:}l) = g$. The function $\mathcal{M}^{u}$ for static terms is defined as follows:*

$$\mathcal{M}^{u}(d, v, g{:}l) = v(d)$$

$$\mathcal{M}^{u}(\texttt{TRUE}, v, g{:}l) = \mathbf{T}$$

$$\mathcal{M}^{u}(\texttt{FALSE}, v, g{:}l) = \mathbf{F}$$

$$\mathcal{M}^{u}(\texttt{UNDEF}, v, g{:}l) = \bot$$

$$\mathcal{M}^{u}(!\tau, v, g{:}l) = \begin{cases} \mathbf{T} & \text{if } \mathcal{M}^{u}(\tau, v, g{:}l) \neq \bot \\ \mathbf{F} & \text{if } \mathcal{M}^{u}(\tau, v, g{:}l) = \bot \end{cases}$$

$$\mathcal{M}^{u}(\texttt{NOT } (t), v, g{:}l) = \begin{cases} \mathbf{T} & \text{if } \mathcal{M}^{u}(t, v, g{:}l) = \mathbf{F} \\ \mathbf{F} & \text{if } \mathcal{M}^{u}(t, v, g{:}l) = \mathbf{T} \\ \bot & \text{otherwise} \end{cases}$$

$$\mathcal{M}^{u}((\tau_1 \texttt{ = } \tau_2), v, g{:}l) = \begin{cases} \mathbf{T} & \text{if } \mathcal{M}(\tau_1, v, g{:}l) = \mathcal{M}(\tau_2, v, g{:}l) \\ \mathbf{F} & \text{otherwise} \end{cases}$$

$$\mathcal{M}^{u}((t_1 \texttt{ OR } t_2), v, g{:}l) = \begin{cases} \mathbf{T} & \text{if } \mathcal{M}^{u}(t_1, v, g{:}l) = \mathbf{T} \vee \mathcal{M}^{u}(t_2, v, g{:}l) = \mathbf{T} \\ \mathbf{F} & \text{if } \mathcal{M}^{u}(t_1, v, g{:}l) = \mathbf{F} \wedge \mathcal{M}^{u}(t_2, v, g{:}l) = \mathbf{F} \\ \bot & \text{otherwise} \end{cases}$$

$$\mathcal{M}^u(\texttt{(EXISTS } d\ t),v,g{:}l) \;=\; \begin{cases} \mathbf{T} & \text{if } \exists u \neq \bot\ \mathcal{M}^u(t,v[d\mapsto u],g{:}l) = \mathbf{T} \\ \mathbf{F} & \text{if } \forall u \neq \bot\ \mathcal{M}^u(t,v[d\mapsto u],g{:}l) = \mathbf{F} \\ \bot & \text{otherwise} \end{cases}$$

$$\mathcal{M}^u(t',v,g{:}l) \;=\; \mathcal{M}^u(t,v,l)$$

$$\mathcal{M}^u(\texttt{[}d\ \texttt{:=}\ \tau\texttt{]}\ t,v,g{:}l)$$
$$= \begin{cases} \mathbf{T} & \text{if } \mathcal{M}^u(t,v[d\mapsto \mathcal{M}^u(\tau,v,g{:}l)],(\mathcal{M}^s(\tau,v,g{:}l)){:}g{:}l) = \mathbf{T} \\ \mathbf{F} & \text{if } \mathcal{M}^u(t,v[d\mapsto \mathcal{M}^u(\tau,v,g{:}l)],(\mathcal{M}^s(\tau,v,g{:}l)){:}g{:}l) = \mathbf{F} \\ \bot & \text{otherwise} \end{cases}$$

$$\mathcal{M}^u(\texttt{(}\tau_0\ \texttt{MODSTATE}\ \tau_1,\ldots,\ \tau_n\texttt{)},v,g{:}l)$$
$$= \begin{cases} \mathbf{T} & \text{if } g =_{(\underline{\mathrm{U}}\setminus \mathrm{T})} \mathcal{M}^s(\tau_0,v,g{:}l) \\ \mathbf{F} & \text{otherwise} \end{cases}$$
$$\text{where } \mathrm{T} = \{\mathcal{M}^u(\tau_1,v,g{:}l),\ldots,\mathcal{M}^u(\tau_n,v,g{:}l)\}$$

$$\mathcal{M}^u(\texttt{(}\tau\ \texttt{MODRESULT}\ p_1,\ldots,p_n\texttt{)},v,g{:}l)$$
$$= \begin{cases} \mathbf{T} & \text{if } g =_{\mathrm{PROC}\setminus \mathrm{B}} \mathcal{M}^s(\tau,v,g{:}l) \\ \mathbf{F} & \text{otherwise} \end{cases}$$
$$\text{where } \mathrm{B} = \{p_1,\ldots,p_n\}$$

$$\mathcal{M}^u(\texttt{(}\tau_0\ \texttt{DEPSTATE}\ \tau_1,\ldots,\ \tau_n\texttt{)},v,g{:}l)$$
$$= \begin{cases} \mathbf{T} & \text{if } g_1 =_{\mathrm{T}} g_2 \Rightarrow \mathcal{M}^u(\tau_0,v,g_1{:}l) = \mathcal{M}^u(\tau_0,v,g_2{:}l) \\ \mathbf{F} & \text{otherwise} \end{cases}$$
$$\text{where } \mathrm{T} = \{\mathcal{M}^u(\tau_1,v,g{:}l),\ldots,\mathcal{M}^u(\tau_n,v,g{:}l)\}$$

$$\mathcal{M}^u(\texttt{(}\tau\ \texttt{DEPRESULT}\ p_1,\ldots,p_n\texttt{)},v,g{:}l)$$
$$= \begin{cases} \mathbf{T} & \text{if } g_1 =_{\mathrm{B}} g_2 \Rightarrow \mathcal{M}^u(\tau_0,v,g_1{:}l) = \mathcal{M}^u(\tau_0,v,g_2{:}l) \\ \mathbf{F} & \text{otherwise} \end{cases}$$
$$\text{where } \mathrm{B} = \{p_1,\ldots,p_n\}$$

The last step is the definition of the dynamic terms $\tau \in \mathrm{PROG}$. Note that in the previous definition nothing was said about the terms LET d = t_1 IN t_2 END and $f(t_1,\ldots,t_n)$. The reason for this is that these terms are special cases of LET d = τ_1 IN τ_2 END and $p(\tau_1,\ldots,\tau_n)$ respectively. The latter terms will be treated in the next definition.

Def 4.6 *In this definition we assume that* $g \neq \dagger$.

$$\mathcal{M}(\texttt{LET } d\ \texttt{=}\ \tau_1\ \texttt{IN}\ \tau_2\ \texttt{END},v,g{:}l) \;=\; \mathcal{M}(t_2,v[d\mapsto \mathcal{M}^u(\tau_1,v,g{:}l)],g'{:}l)$$
$$\text{where } g' = M^s(\tau_1,v,g{:}l)$$

$$\mathcal{M}(p(\tau_1,\ldots,\tau_n),v,g{:}l) \;=\; \mathrm{P}(p)((u_1,\ldots,u_n),g_{n+1})$$
$$\text{where}\begin{cases} g_1 = g \\ g_{i+1} = \mathcal{M}^s(\tau_i,v,g_i{:}l) \\ u_i = \mathcal{M}^u(\tau_i,v,g_i{:}l)\end{cases}$$

$$\mathcal{M}(\texttt{IF } \tau_0 \texttt{ THEN } \tau_1 \texttt{ ELSE } \tau_2 \texttt{ FI},v,g{:}l)$$
$$= \begin{cases} \mathcal{M}(\tau_1,v,g'{:}l) & \text{if } \mathcal{M}^u(\tau_0,v,g{:}l) = \mathbf{T} \\ \mathcal{M}(\tau_2,v,g'{:}l) & \text{if } \mathcal{M}^u(\tau_0,v,g{:}l) = \mathbf{F} \\ (\bot,g') & \text{otherwise}\end{cases}$$
$$\text{where } g' = \mathcal{M}^s(\tau_0,v,g{:}l)$$

$$\mathcal{M}((\tau_1;\tau_2),v,g{:}l) \;=\; \mathcal{M}(\tau_2,v,(\mathcal{M}^s(\tau_1,v,g{:}l)){:}l))$$

$$\mathcal{M}(\texttt{WHILE } \tau_0 \texttt{ DO } \tau_1 \texttt{ OD},v,g{:}l) = (\bot,\sqcup\overline{\phi}(g))$$
$$\text{where}\begin{cases} \phi_i : \mathrm{GS} \to \mathrm{GS} \\ \phi_0(h) = \dagger \\ \phi_{i+1}(h) = \phi_i(\mathcal{M}^s(\tau_1,v,(\mathcal{M}^s(\tau_0,v,h{:}l){:}l))) & \text{if } \mathcal{M}^u(\tau_0,v,h{:}l) = \mathbf{T} \\ \phi_{i+1}(h) = \mathcal{M}^s(\tau_0,v,h{:}l) & \text{otherwise}\end{cases}$$

The definition of $\mathcal{M}^s$ for the while-term needs a justification. We have to prove that $\sqcup\overline{\phi}$ exists. First we define a partial order on GS:

$$g_1 \sqsubseteq_1 g_2 \;\Leftrightarrow\; (g_1 = g_2) \vee (g_1 = \dagger)$$

It can easily be verified that with $\sqsubseteq_1$ GS is a cpo. Next, we define a partial order on GS $\to$ GS as usual:

$$\phi_1 \sqsubseteq_2 \phi_2 \;\Leftrightarrow\; \forall g\,(\phi_1(g) \sqsubseteq_1 \phi_2(g))$$

It follows from lemma 2.7 that $(\mathrm{GS} \to \mathrm{GS}, \sqsubseteq_2)$ is a cpo. Remains to show that $\overline{\phi}$ is a chain in this cpo, i.e., $\forall k\,(\phi_k \sqsubseteq_2 \phi_{k+1})$. This will be done with induction on k.

For $k = 0$ the proof is trivial, so suppose $k > 0$. The induction hypothesis **(IH)** is defined by: $\phi_{k-1} \sqsubseteq_2 \phi_k$

We have to prove that $\phi_k \sqsubseteq_2 \phi_{k+1}$ holds. This means we have to show: $\forall g \in \mathrm{GS}\,(\phi_k(g) \sqsubseteq_1 \phi_{k+1}(g))$. So, let $g \in \mathrm{GS}$ be a global state. Since $k > 0$ we have to consider two cases for $\phi_k(g)$:

case 1: $\mathcal{M}^u(\tau_0,v,g{:}l) = \mathbf{T}$. Now, let $g' = \mathcal{M}^s(\tau_1,v,\mathcal{M}^s(\tau_0,v,g{:}l){:}l)$, then we have

$\phi_k(g)$
$=$ {definition of g' and ϕ_k and $k > 0$}
$\phi_{k-1}(g')$
$\sqsubseteq_1$ {**IH** and definition of $\sqsubseteq_2$}
$\phi_k(g')$
$=$ {definition of g' and ϕ_k and $(k > 0) \to (k+1 > 0)$}
$\phi_{k+1}(g)$

Therefore we have $\phi_k \sqsubseteq_2 \phi_{k+1}$.

case 2: $\mathcal{M}^u(\tau_0, v, g{:}l) = \mathbf{F}$. This is simple:

$$\begin{aligned} & \phi_k(g) \\ = \; & \{\text{definition of } \phi_k \text{ and } k > 0\} \\ & \mathcal{M}^s(t_1, v, g) \\ = \; & \{\text{definition of } \phi_k \text{ and } (k > 0) \rightarrow (k+1 > 0)\} \\ & \phi_{k+1}(g) \end{aligned}$$

So, $\phi_k \sqsubseteq_2 \phi_{k+1}$.

In order to illustrate that the WHILE construct behaves as one expects, we will prove that the term WHILE τ_0 DO τ_1 OD is equivalent to the term IF τ_0 THEN (τ_1 ; WHILE τ_0 DO τ_1 OD) ELSE UNDEF FI.

Lemma 4.7

Let $A =$ WHILE τ_0 DO τ_1 OD

$B =$ IF τ_0 THEN $(\tau_1;$ WHILE τ_0 DO τ_1 OD) ELSE UNDEF FI

Then $\mathcal{M}(A, v, g{:}l) = \mathcal{M}(B, v, g{:}l)$

Proof

We will first show that $\mathcal{M}^u(A, v, g{:}l)$ equals $\mathcal{M}^u(B, v, g{:}l)$. For term A we find, according to the previous definition:

$$\begin{aligned} & \mathcal{M}^u(A, v, g{:}l) \\ = \; & \{\text{definition of } A\} \\ & \mathcal{M}^u(\texttt{WHILE } \tau_0 \texttt{ DO } \tau_1 \texttt{ OD}, v, g{:}l) \\ = \; & \{\text{definition 4.6 for the WHILE ..}\} \\ & \bot \end{aligned}$$

For term B there are three cases:

case 1: $\mathcal{M}^u(\tau_0, v, g{:}l) = \mathbf{T}$ Using the clauses for IF.. and WHILE.. from definition 4.6 we get:

$$\begin{aligned} & \mathcal{M}^u(B, v, g{:}l) \\ = \; & \{\text{definition 4.6 with } \mathcal{M}^u(\tau_0, v, g{:}l) = \mathbf{T} \text{ and let } \mathcal{M}^s(\tau_0, v, g{:}l) = g'\} \\ & \mathcal{M}^u((\tau_1; \texttt{WHILE } \tau_0 \texttt{ DO } \tau_1 \texttt{ OD}), v, g'{:}l) \\ = \; & \{\text{definition 4.6 and let } \mathcal{M}^s(\tau_1, v, g'{:}l) = g''\} \\ & \mathcal{M}^u(\texttt{WHILE } \tau_0 \texttt{ DO } \tau_1 \texttt{ OD}, v, g''{:}l) \\ = \; & \{\text{definition 4.6 for the WHILE ..}\} \\ & \bot \end{aligned}$$

case 2: $\mathcal{M}^u(\tau_0, v, g{:}l) = \mathbf{F}$ In this case we get:

$\mathcal{M}^u(B, v, g{:}l)$
$=$ {definition of B}
$\mathcal{M}^u(\texttt{IF } \tau_0 \texttt{ THEN } (\tau_1; \texttt{WHILE } \tau_0 \texttt{ DO } \tau_1 \texttt{ OD}) \texttt{ ELSE UNDEF FI}, v, g{:}l)$
$=$ {definition 4.6 with $\mathcal{M}^u(\tau_0, v, g{:}l) = \mathbf{F}$ and $\mathcal{M}^s(\tau_0, v, g{:}l) = g'$}
$\mathcal{M}^u(\texttt{UNDEF}, v, g'{:}l)$
$=$ {definition 4.5}
$\perp$

case 3: otherwise Now, according to the clause for IF..THEN..ELSE..FI in definition 4.6, the result is $\perp$.

The state-interpretation is a bit more difficult. For term A we find $\mathcal{M}^s(A, v, g{:}l) = \sqcup\overline{\phi}(g)$. If $\mathcal{M}^u(\tau_0, v, g{:}l) \neq \mathbf{T}$ then, using the definition for ϕ_i, we find: $\sqcup\overline{\phi}(g) = \mathcal{M}^s(\tau_0, v, g{:}l)$.

For term B we have, again, three cases:

case 1: $\mathcal{M}^u(\tau_0, v, g{:}l) = \mathbf{T}$ Now we get:
$\mathcal{M}^s(B, v, g{:}l)$
$=$ {see similar step in **case 1** of $\mathcal{M}^u$ }
$\mathcal{M}^s((\tau_1; \texttt{WHILE } \tau_0 \texttt{ DO } \tau_1 \texttt{ OD}), v, g'{:}l)$
$=$ { see similar step in **case 1** of $\mathcal{M}^u$ }
$\mathcal{M}^s(\texttt{WHILE } \tau_0 \texttt{ DO } \tau_1 \texttt{ OD}, v, g''{:}l)$
$=$ { see similar step in **case 1** of $\mathcal{M}^u$ }
$\sqcup\overline{\phi}(g'')$

case 2: $\mathcal{M}^u(\tau_0, v, g{:}l) = \mathbf{F}$ In this case we get:
$\mathcal{M}^s(B, v, g{:}l)$
$=$ {definition of B}
$\mathcal{M}^s(\texttt{IF } \tau_0 \texttt{ THEN } (\tau_1; \texttt{WHILE } \tau_0 \texttt{ DO } \tau_1 \texttt{ OD}) \texttt{ ELSE UNDEF FI}, v, g{:}l)$
$=$ {definition 4.6 with $\mathcal{M}^u(\tau_0, v, g{:}l) = \mathbf{F}$}
$\mathcal{M}^s(\texttt{UNDEF}, v, \mathcal{M}^s(\tau_0, v, g{:}l){:}l)$
$=$ {definition 4.5}
$\mathcal{M}^s(\tau_0, v, g{:}l)$

case 3: otherwise Now, according to the clause for IF .. THEN .. ELSE .. FI in definition 4.6, the result is $\mathcal{M}^s(\tau_0, v, g{:}l)$.

It is clear that for case 2 and 3 we have proven the equality of the state-interpretations of A and B. Remains to show that:

$$\sqcup\overline{\phi}(g) = \sqcup\overline{\phi}(g'')$$

where $g'' = \mathcal{M}^s(\tau_1, v, \mathcal{M}^s(\tau_0, v, g{:}l){:}l)$.

$$
\begin{array}{ll}
 & \sqcup\overline{\phi}(g) \\
= & \{ \text{lemma 2.8} \} \\
 & \sqcup\overline{\phi_i(g)} \\
= & \{ \text{definition of } \phi_i \text{ for } \mathcal{M}^u(\tau_0, v, g{:}l) = \mathbf{T} \} \\
 & \sqcup\overline{\phi_{i-1}(\mathcal{M}^s(\tau_1, v, \mathcal{M}^s(\tau_0, v, g{:}l){:}l))} \\
= & \{ \text{definition of } g'' \} \\
 & \sqcup\overline{\phi_{i-1}(g'')} \\
= & \{ \text{lemma 2.5} \} \\
 & \sqcup\overline{\phi_i(g'')} \\
= & \{ \text{lemma 2.8} \} \\
 & \sqcup\overline{\phi}(g'')
\end{array}
$$

4.3 Semantics of a Specification

In the previous section we defined formally the meaning of AFSK-D terms. The next step is to define the meaning of a specification. We will use an S to denote a specification.

As mentioned before, a specification is a set of terms: $S \subseteq$ TERM. These terms are interpreted with respect to a structure, a special symbol $\dagger$ and a valuation. Given a global state and a valuation the function $\mathcal{M}^u$ gives the object-interpretation of a term. The object-interpretation is an object of the universe of the structure.

Def 4.8 *Let I be a structure, v a valuation and $g \neq \dagger$, then*

$$I, v, g \models t \text{ iff } \mathcal{M}^u(t, v, g{:}\varepsilon) = \mathbf{T}$$

Whenever $I, v, g \models t$ we say that t is *valid* in state g of I with valuation v. When $I, v, g \models t$ holds for every valuation v the v can be omitted: $I, g \models t$ and when $I, g \models t$ holds for every global state g we write: $I \models t$. If we say that a term t is valid in a structure I, we mean: $I \models t$, i.e., it is valid for every valuation and ever global state.

Def 4.9 *Let S be a specification and I a structure. I is a model for S, $I \models S$, if and only if $\forall t \in S\ I \models t$.*

It is possible that a specification has more than one model. We define the class of all models to be the meaning of a specification. This kind of semantics is

called *loose semantics*. For the language COLD [FJ92] also a loose semantics is given. The function $[\![\cdot]\!]$ gives the meaning of a specification:

Def 4.10 $[\![S]\!] = \{I \mid I \models S\}$

4.4 AFSK-D and Related Formalisms

In this section we will compare several other specification languages and formalisms with AFSK-D. QDL stands for Quantified Dynamic Dogic, see [Har84]. MLCM stands for Modal Logic of Creation and Modification, which is a variant of QDL described in [GdL93]. The paradigm *states as algebras* which is at the core of MLCM is also at the core of FLEA (formal language for evolving algebra) [GdL95] and of COLD (common object-oriented language for design) [FJ92] and [FJM94]. Since this paradigm is used in at least three of these formalisms, we want to find out if the same can be said for AFSK-D. In QDL the states as algebras paradigm cannot be found, however, but, since QDL has influenced all other formalisms we mentioned above, including AFSK-D, it deserves some attention.

States as algebras can be described as follows. In all of the five languages mentioned, the user can define names for functions, predicates, etc. These names are syntactic entities. To give an interpretation to these user-defined names every language uses its own mathematical structures. With the aid of these structures, all user-defined names are associated with a semantical entity, usually a function. In FLEA, MLCM and COLD this association of syntatic names with semantic functions is called an algebra. In this section we will call these associations in QDL and AFSK-D algebras, too.

In COLD, MLCM, and FLEA the algebra is variable. This means that the association of syntactic names with semantic functions can be changed by *dynamic constructs* of the particular language. A change of the algebra corresponds to a state change. Wether or not a state change corresponds to a change of the algebra, depends on the particular language, as we will see in a moment.

In QDL the algebra plays a minor role, since it is constant. By constant we mean that the dynamic constructs of QDL, which are called *programs*, cannot change the algebra. Programs in QDL can only change the value of a variable and, as a consequence, in QDL there is no distinction between logical and programming variables.

In AFSK-D the algebra is constant, too, but this does not mean that QDL and AFSK-D are more related than, for example, COLD and AFSK-D. We wil soon show that it is more plausible to find COLD and AFSK-D more related to each other than QDL and AFSK-D.

```
CLASS

 PRED Value :   VAR

 PROC Change : -> MOD Value

 AXIOM INIT => Value
 AXIOM <Change> TRUE

 AXIOM INIT => (([Change] NOT(Value)) AND
                ([Change;Change] Value) AND
                ([Change;Change;Change] Value))
END
```

Figure 4.1: COLD specification

In FLEA and in MLCM states are, by definition, algebras. This means that the same algebras represent the same states. So, for FLEA and MLCM we could rephrase "states as algebras" as "states are algebras". In COLD this identification is not made. There the states as algebras view means that associated with every state is an algebra. So, there can be different states with the same algebra. From now on we will use the term "states as algebras" for the COLD approach. In QDL and AFSK-D we cannot associate with every state an algebra. In QDL states are described as variable valuations and in AFSK-D (global) states are mappings from objects to local states, as we have seen in definition 4.1.

To illustrate a difference between "states are algebras" and "states as algebras" we will give an example of a COLD specification. In this specification we define a system with one predicate Value and one procedure Change. The procedure Change changes the state in the way as described in figure 4.1.

This specification says that initially Value holds. When the procedure Change is executed once, NOT(Value) holds, when it is executed twice Value holds, etc. So, the values of Value in succesive states reached by the procedure Change are: TRUE, FALSE, TRUE, TRUE, FALSE, TRUE, TRUE, ... A model of this specification could be a structure with four states, s_0,s_1, s_2 and s_3. s_0 is the initial state. The algebras *associated* with states s_0, s_2 and s_3 are the same: Value holds in this algebra and there are no other function names. The algebra associated with s_1 is different, since there NOT(Value) holds. The interpretation of the procedure Change in this model could be the relation $\{(s_0, s_1), (s_1, s_2), (s_2, s_3), (s_3, s_1)\}$. This means that consecutive executions of Change starting in s_0 will yield the sequence $s_0, s_1, s_2, s_3, s_1, s_2, s_3, s_1 \ldots$

Note that is important not to identify the states s_2 and s_3, since then the specification would be inconsistent. Therefore, if we want to describe a predicate

```
Value() MODSTATE

Value() OR NOT(Value())

Change() MODRESULT Value

Value() => (([Change()] NOT(Value())) AND
            ([Change();Change()] Value()) AND
            ([Change();Change();Change()] Value()))
```

Figure 4.2: AFSK-D specification

Value and a procedure Change in FLEA or in MLCM, we would have to add something to the algebra in order to distinguish the algebras s_2 and s_3, otherwise the algebras would be the same and therefore the states would be the same, too. This could be a reason why in [FJ92] it is emphasized that te for every object. states and algebras should not be identified. In COLD a state can hide information from the algebra associated with that state. This gives the possibility to write specifications like the one given above.

In AFSK-D the algebra is constant. Given a specification and a structure, the algebra is determined by the procedure name interpretation function. However, the specification in figure 4.2 is not inconsistent, since one easily finds a model for it. In AFSK-D we can hide information in the local states of the objects. So, both in COLD and in AFSK-D we can describe procedures which depend on information which is hidden in a state or local state. The difference is that COLD uses only one state whereas AFSK-D uses a state for every object.

A difference between COLD and AFSK-D is the way in which states are defined. In AFSK-D we tried to stay within the object-oriented paradigm. Therefore we have structures in which every object has a local state. The association of objects with their local states is called a global state. In COLD states are used to describe models for so called state-based specifications. There is no direct relation between objects and states, even though "object-oriented" is part of the name of the language. Therefore we think our approach is more object-oriented than the COLD approach.

5 Examples

In the previous chapters we have described AFSK-D and its semantics. In this chapter we will give some specification written in AFSK-D. These examples are not meant to be realistic. Their purpose is to illustrate how AFSK-D could be used. However, since AFSK-D is a kernel language, the specifications are sometimes not very nice to see. For example, since AFSK-D does not have a typing-mechanism, we sometimes had to use characteristic functions to denote special sets of objects. This is a bit of a nuisance, but we think the specifications still are clear enough.

5.1 Programming Variables

In this section we will give an example of an AFSK-D specification. The goal of this section is twofold. First, it is a demonstration of how AFSK-D can be used. Second, this section gives a more thorough explanation of the difference between logical variables and programming variables.

Since there are no sorts in AFSK-D, we use the characteristic function `IsVar` to distinguish between programming variables and other objects. The operations `NewVar` and `DisposeVar` are used to create and destroy programming variables, respectively. One can get the value of a programming variable by the function `Value`. Finally, there is an operation with which we can assign a new value to a programming variable: `AssignVar`.

In the specification we use two logical variables, `c` and `d`. Thus, the signature of the specification consists of the function names `IsVar` and `Value` and the procedure names `NewVar`, `DisposeVar` and `AssignVar`. The specification is given in figure 5.1. Lines beginning with a % are comments. With the term `IsVar(d) MODSTATE` we say that `IsVar` cannot change the state. This means that `IsVar` is a function: an operation without a side-effect. The same holds for `Value`. The term `NewVar() MODSTATE NewVar()` says that the only object which state can be changed by `NewVar` is the object-interpretation of `NewVar` itself.

Programming vs logical variables. Logical variables have a static nature; they always refer to the same object. By this we mean that the value of a logical variable is determined by a variable valuation and therefore it can only be changed by changing the variable valuation. In general, the only way to change the variable valuation is by quantification. In AFSK-D there are two other possibilities to change the variable valuation: $[d := \tau]\ t$ or `LET` d `=` τ_1 `IN` τ_2 `END`. These terms change the variable valuation such that when determining the meaning of t

```
% The characteristic function IsVar
IsVar(d) DEPSTATE d
IsVar(d) MODSTATE
IsVar(d)   OR NOT(IsVar(d))

% Variables have a value
IsVar(d) ==> !Value(d)
Value(d) DEPSTATE d
Value(d) MODSTATE

% Creating a new variable
NewVar() MODSTATE NewVar()
NewVar() MODRESULT IsVar, Value
[v := NewVar()] (IsVar(v) AND NOT(IsVar(v)'))

% Destroying variables
DisposeVar(d) MODSTATE d
DisposeVar(d) MODRESULT IsVar
[DisposeVar(d)] NOT(IsVar(d))

% Assigning values to variables
AssignVar(d,c) MODSTATE  d
AssignVar(d,c) MODRESULT Value
IsVar(d) ==> ([AssignVar(d,c)] Value(d) = c)
```

Figure 5.1: Programming Variables

or τ_2 respectively, the variable d has the value of τ and τ_1 respectively. Note that, just like the EXIST quantifier, the variable valuation is only changed locally. For example, if we say:

```
[d := proc1(a,b)] term1
term2
```

In `term1` the variable `d` is bound to the value of `prog(a,b)`, but it is free in `term2`.

So, logical variables are not really static, but when their value is changed, the change is local to the term in which they are changed. As mentioned before in AFSK-D there are three terms which can change logical variables:

(EXISTS d t)

[d := τ] t

LET d = τ_1 IN τ_2 END

Besides these terms we also use the abbreviations:

`(FORALL` d t`)`

`[`τ`]` t

see table 3.1 on page 12.

Programming variables have a dynamic nature; their values can vary in time. In [FJM94] programming variables are viewed as variable functions (see page 41). In figure 5.1 we have specified variables as objects with certain features: they have a value which can be changed by the procedure `AssignVar`. Just like any other objects, programming variables have their own local state. This state can be changed by the procedure `AssignVar`, in which case the function `Value` has a different result. Note that there is no direct relation between the variable valuation and the value of a programming variable.

In the dynamic logic of Harel, see [Har84], there is no distinction between logical variables and programming variables. There the variables which you use in quantifications like $\forall x\ E$ are the same as the variables which you use as the left-hand side of assignments $x{:=}E$. This gives the user the possibility to use logical variables as if they were programming variables, which can easily result in a lot of confusion.

There are variants of dynamic logic in which one cannot use logical variables as programming variables. For example, MLCM as described in [GdL93]. Here logical variables are treated similar to the way they are treated in COLD, see [FJ92, FJM94]. In the latter logical variables are called *object names* (see page 114). These names can be used in expressions to refer to certain objects. In fact, we use the same approach for logical variables: the value of a variable is determined by a variable valuation. The variable valuation can only be changed temporarily by a quantifier or a term like `[`d `:=` τ`]` t or `LET` d `=` τ_1 `IN` τ_2 `END`.

We treat programming variables differently, because we model them as objects with a local state, whereas in MLCM and in COLD they are modeled as variable functions.

5.2 A Storage Allocator

A storage allocator is a program which manages blocks of storage which users may want to access. Almost every operating system has a storage allocator which takes care of the distribution of blocks of memory to users. Sometimes the name *memory manager* is used for the storage allocator (see [Tan87]). In this chapter we give an example of a specification of a storage allocator. Since it is not our purpose to describe a real live storage allocator, the storage allocator is simplified.

In section 7.4 in [WL88] a Z-specification of a storage allocator is given. In this chapter we will stay close to that example. This gives us the possibility to compare

both specifications. However, we do not claim that the AFSK-D-specification given here describes the same storage allocator as the Z-specification in [WL88].

5.2.1 Users and Blocks

We assume there is a set of users and a set of memory blocks. The users can access the blocks of storage. Since there are no sorts in AFSK-D we use the characteristic function `Is_User` to identify users. Blocks are identified by the characteristic function `Is_Block`. The storage allocator keeps track of which user has allocated which blocks. To do this safely, the following requirements must be met.

1. No block is allocated by more than one user
2. Every block is either allocated by a user or it is free

The boolean-valued function `Free` is used to distinguish between free and non-free blocks. The function `Dir` maps a block to the user who has allocated that block. This is a partial function, because some blocks may be free. `Free` and `Dir` only depend on their actual parameters and since they are functions they cannot change the state. We can specify this with the `DEPSTATE` and `MODSTATE` constructs of AFSK-D as is done in the terms {-Free1-} and {-Dir1-}.

```
{-Free1-} (Free(b) DEPSTATE b) AND (Free(b) MODSTATE  )
{-Dir1-}  (Dir(b)  DEPSTATE b) AND (Dir(b) MODSTATE   )
```

In AFSK-D partial functions like `Dir` are specified with the use of `Undef`. The applications of a partial function on arguments for which the function is not defined, will yield `Undef`. In this way, all functions are in fact total functions. The function `Dir` is not defined for blocks which are free:

```
{-Dir2-} Free(b) ==> (Dir(b) == Undef)
```

Since we do not want the storage allocator to "lose" blocks, we formulate the following invariant:

```
{-INV-} FORALL bl (Is_Block(bl) ==> (Free(bl) OR Is_User(Dir(bl))))
```

{-INV-} says that if `bl` is a block, then it is free or it has a user. So, there can be no blocks dangling around in the operating system without the storage allocator knowing about it. {-INV-} is *not* part of the specification, but we claim that whenever the storage allocator is in a state in which {-INV-} holds, it will hold forever.

Since we want {-INV-} to be an invariant of the storage allocator, we have to show that no operation of the storage allocator violates {-INV-}. Every specification of an operation below thus yields a proof obligation. We cannot formally proof the invariance of {-INV-}, because we do not have a formal proof system for AFSK-D. However, in the next subsection we will say something about the proof rules we would like to have and give a sketch of a proof.

5.2.2 Requesting and Releasing Blocks

A block request is done by the operation Request_Block. This operation has one parameter: a user doing the request. The result of request_block is block of memory. The operation only succeeds if there is a free block. Therefore, the precondition of Request_Block(u) is: EXISTS b (Free(b)). Request_Block(u) only modifies the local state of its result. In {-Req1-} we have defined this. After a successful request the functions Free and Dir are changed. So, Request_Block modifies the results of these two functions, see {-Req2-}.

```
{-Req1-} Request_Block(u) MODSTATE Request_Block(u)
{-Req2-} Request_Block(u) MODRESULT Free, Dir
{-Req3-} (Is_User(u) AND (EXISTS b (Free(b)))) ==>
         [b := Request_Block(u)]( Free(b)' AND NOT(Free(b))
                                  AND (Dir(b) = u))
```

We now have the following proof-obligation: If {-INV-} holds before the execution of b := Request_Block(u) and (IsBlock(b) AND Free(b)) and Is_User(u) hold too, then {-INV-} must hold afterwards. There are two cases: bl = b and bl ≠ b.

case 1. bl = b Now we can reformulate {-INV-} as:

```
Is_Block(b) ==> (Free(b) OR Is_User(Dir(b)))
```

The term {-Req3-} says that after the execution of b := Request_Block(u) we have NOT(Free(b)) AND (Dir(b)= u). Since the only functions which can be changed by Request_Block are Free and Dir, the result of Is_User(u) can not be changed. Therefore, after the execution of b := Request_Block(u) we know that Is_User(Dir(b)) holds, since Dir(b) = u. So, {-INV-} must hold afterwards. To proof this formally, we would like to have, among others, the following proofrule:

Suppose $f \not\equiv g$ are two different function names and τ is an arbitrary dynamic term, then

$(\tau$ MODRESULT $g) \Rightarrow$ FORALL $\overline{x}\, y\, (f(\overline{x}) = y$ ==> $[\tau](f(\overline{x}) = y))$

case 2. bl ≠ b Now we need not to reformulate {-INV-}. We know by {-Req1-} that the only object which state can be changed by Request_Block(u) is the block returned by Request_Block(u). Furthermore, the functions Free and Dir only depend on their actual parameter. So, it is clear that:

```
{-INV-} ==> ([b := Request_Block(u)] {-INV-})
```

holds, since bl ≠ b. The proofrule we would like to have, could be stated as:

Let p be a procedure name, $d_1 \not\equiv d_2$ are two different variables and τ is an arbitrary dynamic term, then

$((p(d_1)$ DEPSTATE $d_1)$ AND $(\tau$ MODSTATE $\tau)) \Rightarrow$
$[d_2 := \tau]$ (NOT(d_1=d_2) ==> ($p(d_1)$ = $p(d_1)$'))

We will not go any further into the subject of proof oblications and formal proofrules. This would go beyond the scope of this report. However, this is an interesting research area, which needs to be investigated if AFSK-D and its user language AFSL-D are to be used in practice.

When a user does not need a block anymore he can release it with the operation Release_Block. This operation has two parameters: the user who releases a block and a block which is being released. Release_Block(u,b) succeeds if the block b is allocated by the user u. In this case, the block is free after the execution of Release_Block(u,b). The operation Release_Block can change the functions Free and Dir. However, just like the Request_Block operation, it can do this only by changing the local state of the object b.

```
{-Rel1-} Release_Block(u,b) MODSTATE b
{-Rel2-} Release_Block(u,b) MODRESULT Free, Dir
{-Rel3-} (Dir(b) = u) ==>
         [Release_Block(u,b)] Free(b) AND (NOT(Dir(b) = u))
```

In figure 5.2 the complete specification of the storage allocator is given. The signature of this specification consists of the procedure names: IsBlock, Free, Dir, IsUser, Request_Block and Release_Block.

5.2.3 Error Situations

The operations Request_Block and Release_Block are only described for cases in which their preconditions hold. In this section we will describe what happens when the operations are executed and the preconditions do not hold. These are so called *error situations.*

When a user does a request for a block of memory and there is no free block left, we want the storage allocator to notify the user that he/she cannot get a block. The actual distribution of blocks over users should not change in this situation. To notify the user that there are no more blocks, we assume there is a set of *error-codes* one of which is denoted by No_More_Blocks. When the precondition of Request_Block(u) does not hold, the result of Request_Block(u) will be this error code.

```
{-ReqErr-} NOT(EXISTS b (IsBlock(b) AND Free(b))) ==>
           [err := Request_Block(u)] err == No_More_Blocks
```

The operation Release_Block has two parameters: a user u and a block b. The precondition of Release_Block(u,b) is Dir(b) = u. So, when this does not hold, the storage allocator will be in an error situation. Note that Dir is a partial

```
-- A Storage Allocator
--
{-Free1-} Free(b) DEPSTATE b
{-Dir1-}  Dir(b)  DEPSTATE b

-- Dir is a partial function
{-Dir2-} Free(b) ==> (Dir(b) == Undef)

-- Requesting blocks
{-Req1-} Request_Block(u) MODSTATE Request_Block(u)
{-Req2-} Request_Block(u) MODRESULT Free, Dir
{-Req3-} Is_User(u) ==> (EXISTS b (IsBlock(b) AND (Free(b)))) ==>
         [b := Request_Block(u)]( Free(b)' AND NOT(Free(b))
                                  AND (Dir(b) = u))

-- Releasing blocks
{-Rel1-} Release_Block(u,b) MODSTATE b
{-Rel2-} Release_Block(u,b) MODRESULT Free, Dir
{-Rel3-} (Dir(b) = u) ==>
         [Release_Block(u,b)] Free(b) AND (NOT(Dir(b) = u))
```

Figure 5.2: A Storage Allocator

function on blocks. Only when b is not free, the term Dir(b) is specified. So, we have two possible error situations for Release_Block(u,b):

1. Free(b): The block is already free
2. NOT(Dir(b) = u): The block is not owned by the user

We assume there are error codes Block_Already_Free and Permission_Denied which we will use in these two cases respectively.

```
{-RelErr1-} Free(b) ==>
      [err := Release_Block(u,b)] (err == Block_Already_Free)
                                          AND Free(b)
{-RelErr2-} NOT(Dir(b)==u) ==>
      [err := Release_Block(u,b)] (err == Permission_Denied)
                                         AND (Dir(b) == Dir(b)')
```

5.2.4 Comparing AFSK-D with Z

The storage allocator example originates from [WL88]. There a Z-specification of a storage allocator was given. In this section we will compare the Z-specification with our AFSK-D-specification.

In Z there is no way to specify modification rights of operations other than by schemes. In a schema every component should be specified. For example, in the schema of $Request_0$ in [WL88] page 149, it is specified that the function *free'* is the same as the function *free*, except for the argument *b*!. For the function *Dir* a simmilar treatment can be found. In our AFSK-D specification we did not have to say anything about applications of `Free` and `Dir` other than for `Free(b)` and `Dir(b)`, because in `{-Free1-}` and in `{-Dir1-}` we have stated that these functions only depend on their actual parameters.

Another difference between Z and AFSK-D is the way in which functions and operations are handled. In Z operations are other semantical entities than functions. Functions have a domain and a range and they are modeled as special sets. Operations are described by schemas. In a schema there is a declaration part in which some variables are introduced, and a predicate which constraints these variables. A schema can have three different sorst of variables: variables, input variables and output variables. Besides that it is possible to have parameterized schemas. The input variables of a schema in which an operation is specified can be viewed as the parameter of the operation. The output variable is the result of the operation. However, there is no limit in the number of variables per schema, so in Z an operation can have more than one results. For example, the result of $request_0$ is both a block: *b* and a so called *report*: *okay*.

In AFSK-D functions and procedures are the same kind of semantical operations. In chapter 6 we will give a classification of operations in which we will define functions and procedures as different classes. However, the class of functions is a subset of the class of procedures, and therefore the difference between functions and procedure is much smaller than in Z.

There are, of course, a lot more differences betwen Z and AFSK-D, but we cannot give them all. The last difference we want to give is not directly related to the example of the storage allocator, but it is about the semantics of both languages. The semantics of Z, as described in [WL88], is defined using ordinary set theory. Relations and functions are modeled as sets with special features. So, one can say that a Z specification of whatever system is a description of that system in terms of sets. Probably this is not the way in which Z-users think about their specifications, but the point we want to make is that a model for a Z specification is a mathematical structure of suitable sets. For this reason Z is sometimes called a *model-oriented* specification language.

Models for AFSK-D specification are structures as described in definition 4.1 in which the interpretation of the terms of the specification are true. So, these models are structures with certain properties corresponding to the terms of the specification. Therefore we call AFSK-D a *property-oriented* specification language.

5.3 Files

In this section we give a specification of files and some operations on files. The reason for choosing files is that they are very common in the computer-world and there are interesting dynamic operations on files like creating, writing or deleting files. In [Jon91] a filesystem example is used to describe the pre- and postcondition technique used in COLD. We will not describe a pre- and postcondition technique for AFSK-D, because in AFSK-D there is no direct way to describe pre- and postconditions. However, the filesystem example given in [Jon91] has inspired us, as can be noticed in the next example.

5.3.1 The File-Objects

Files are sequences of bytes. We use the function `IsFile` to distinguish between files and other objects. We will not mention bytes in our specification, because that will not be necessary for this example. We will call the bytes in a file the contents of that file.

Every file has a name. The name of a file uniquely identifies a file, so there are not two files with the same name. The name of a file is always defined.

The number of bytes in a file is the size of the file. The size of a file is a natural number and therefore always defined. The bytes of a file are numbered from 0 to $n-1$, where n is the size of the file.

The file-pointer of a file is a natural number less than or equal to the size of the file. If the file-pointer is m, it points to the m^{th} byte of the file. When the size of the file is 0, the file-pointer is undefined.

The byte pointed at by the file-pointer is called the current byte. This byte can be accessed by file-operations which we will describe in the section 5.3.3. When the size of the file is 0, the current byte does not exist, i.e, it is undefined. The current byte depends on the contents and the file-pointer of a file. In figure 5.3 we have specified the file-objects.

5.3.2 Creating and Deleting Files

When a file is created, the size is set to 0. The name of the file is a parameter of the `create` operation. This name may not be the name of an already existing file. The file-pointer is moved on the first byte of the file. Since the size is 0 `Curr` need not be defined.

The result of deleting a file is that the object is not a file anymore. The name of a deleted file is no longer bound to that file and can be used for other files. The create and delete operations are specified in figure 5.4

```
IsFile(f) MODSTATE
IsFile(f) DEPSTATE f
IsFile(f) ==> (!Name(f) AND !Size(f))

(IsFile(f) AND Size(f)) > 0 ==>
   (!Fptr(f) AND !Curr(f) AND (0 <= Fptr(f) <= Size(f)))

FORALL n ((0 <= n  < Size(f)) ==> !Contents(f,n))

Name(f) MODSTATE
Name(f) DEPSTATE f

Size(f) MODSTATE
Size(f) DEPSTATE f

Fptr(f) MODSTATE
Fptr(f) DEPSTATE f

Contents(f,n) MODSTATE
Contents(f,n) DEPSTATE f

Curr(f) = Contents(f,Fptr(f))

Eof(f) = (Size(f)=0) OR (Fptr(f) = Size(f))

(Name(f1) = Name(f2)) ==> (f1 = f2)
```

Figure 5.3: The file-objects

```
Create(nm) MODSTATE Create(nm)
Create(nm) MODRESULT IsFile

(NOT(EXIST f (IsFile(f) AND (Name(f) = nm)))) ==>
 ([f := Create(nm)] ( IsFile(f)      AND (Name(f) = nm) AND
                      (Size(f) = 0) AND (Fptr(f) = 0)  AND
                      NOT(IsFile(f)')))

Delete(f) MODSTATE f
Delete(f) MODRESULT IsFile
[Delete(f)] NOT(IsFile(f))
```

Figure 5.4: Creating and deleting files

5.3.3 Accessing Files

In this section we describe some operations with which we can access files. The file-operations we will describe are: `Read`, `Write`, `MovePointer`, `Rename`, `Reset` and `Rewrite`. The operation `Read` yields the current byte of a file. This is only possible if the end of the file is not reached yet, so the precondition for `Read(f)` is `NOT(Eof(f))`. Besides returning the current byte, `Read` also advances the file-pointer by one position.

The `Write` operation first writes a byte on the position pointed at by the file-pointer and then advances the file-pointer one position. When `Eof(f)` holds prior to `Write(f,b)`, the file `f` will be extended with one byte. Therefore, `Write(f,b)` can change the contents, the size and the file-pointer of a file.

The functions `Curr` and `Eof` are defined in terms of `Fptr` and `Contents`. Therefore, every procedure which can change the results of these functions also has to be able to change the results of `Curr` and `Eof`. That's why we listed both `Curr` and `Eof` in the `MODRESULT`-terms for the procedures `Read` and `Write`.

The file-pointer can be moved to an arbitrary position by the operation `MovePointer`. The only condition for moving the file-pointer is that the new position is less than or equal to the actual size of the file. Thus, the size of the file cannot be changed by `MovePointer`. Since `Fptr` can be changed by `MovePointer` we have to add `Curr` and `Eof` to the `MODRESULT` list of `MovePointer`.

The operation `Rename` changes the name of a file. The new name is only allowed when it is not already a name of some other file. `Rename` cannot change any other function than `Name`.

The operation `Reset` sets the filepointer of a file to the first byte of the file. The operation `Rewrite` cleares the contents of a file by setting the size of a file to 0.

```
Read(f) MODSTATE f
Read(f) MODRESULT Fptr, Curr, Eof

NOT(Eof(f)) ==>([c := Read(f)] (c = Curr(f)') AND
                               (Fptr(f) = Fptr(f)' + 1))

Write(f,b) MODSTATE f
Write(f,b) MODRESULT Contents, Size, Fptr, Curr, Eof
Eof(f) ==> ([Write(f,b)] Size(f) = Size(f)' + 1)
NOT(Eof(f)) ==> ([Write(f,b)] Size(f) = Size(f)')
[Write(f,b)]((FORALL n (((n <> Fptr(f)') AND (0 <= n < Size(f)')) ==>
                        (Contents(f,n) = Contents(f,n)')))
             AND (Contents(f,Fptr(f)') = b)
             AND (Fptr(f) = Fptr(f)' + 1))

MovePointer(f,n) MODSTATE f
MovePointer(f,n) MODRESULT Fptr, Curr, Eof
(n <= Size(f)) ==> [MovePointer(f,n)] Fptr(f) = n

Rename(f,nm) MODSTATE f
Rename(f,nm) MODRESULT Name
FORALL f2 (IsFile(f2) ==> NOT(Name(f2) = nm))
 ==> ([Rename(f,nm)] Name(f) = nm)

Reset(f) MODSTATE f
Reset(f) MODRESULT Fptr, Curr, Eof
Size(f)>0 ==> ([Reset(f)] Fptr(f)=0)

Rewrite(f) MODSTATE f
Rewrite(f) MODRESULT Size, Fptr, Contents, Curr, Eof
[Rewrite(f)] (Size(f)=0)
```

Figure 5.5: Accessing files

6 A Classification of Operations

In chapter 4 we have seen that in AFSK-D there is no distinction between predicates, functions and procedures. In AFSK-D there are only procedures and when there is need for a predicate or a function we have to put restrictions on certain procedures. For example, in figure 5.3 we have the term:

```
Name(f) MODSTATE
```

With this term we say that the procedure Name cannot change the local state of any object. This makes Name(f) a *static* term and therefore we prefer to call Name a *function* instead of a procedure. For the same reason we find that Size and Fptr are functions too.

The difference between functions and procedures described here is based on the former being static and the latter being dynamic. In this chapter we will give a formal clasification of operations in which there is a class for functions, a class for procedures and several other classes. In order to give such a classification, we first have to define a theoretical framework which describes what operations are in general. This framework has already been given in definition 4.1. Definition 6.1 will repeat what is important for this chapter. After that we will give some definitions in which several classes of operations are described.

The classification will be given in section 6.2. After that a section follows in which we first describe the classification in AFSK-D, and second describe the possible influence it could have for the development of the user language AFSL-K.

6.1 Classifying Operations

In AFSK-D operations are denoted by procedure names. Just after the definition of a structure, definition 4.1, we said that an operation is a function from a tuple of objects and a global state to an object (the *object-result*) and a global state (the *state-result*). Using this description of an operation we can give the following definition.

Def 6.1 *Let* (U, LS, GS, P) *be a structure.*

- *A function* $o : U^* \times GS \rightarrow U \times GS$ *is called an operation*
- *The class of all operations is denoted by:* OPS

In this chapter it is not important for an operation o to be a total function[1], so in general it could be a partial function. For an operation o, a tuple of objects $\overline{u}$ and a global state g the object-result of an operation is denoted by $o^u(\overline{u}, g)$ and the state-result is denoted by $o^s(\overline{u}, g)$.

The classification is based on two aspects of operations:

- Dependency on the global state
- Modifiability of the global state

We say that an operation is not dependent on the global state if, given a tuple of objects $\overline{u}$, for every global state g $o(\overline{u}, g)$ yields the same object-result $o^u(\overline{u}, g)$. We call these operations NSD:

Def 6.2 *An operation o is not-state-dependent (*NSD*) iff*

$$\forall g_1, g_2, \overline{u}\ (o^u(\overline{u}, g_1) = o^u(\overline{u}, g_2))$$

An operation can also depend on the local states of its actual parameters. In this case it will be called LSD. In the definition of LSD operations we use definition 4.2 on page 20.

Def 6.3 *An operation o is local-state-dependent (*LSD*) iff*

$$\forall g_1, g_2, \overline{u}\ (g_1 =_{\overline{u}} g_2 \Rightarrow o^u(\overline{u}, g_1) = o^u(\overline{u}, g_2))$$

Note that according to these definitions operations that are NSD are also LSD. Operations that cannot be classified as LSD are said to be *global-state-dependent*. This class is the class of all operations, OPS, which is already defined in definition 6.1

$$\text{NSD} \subset \text{LSD} \subset \text{OPS}$$

The classification given so far is only based on the dependency of an operation on the global state. The next step is to describe how a operation can change the global state. The first class of "modifying" operations are those operations that cannot modify the global state at all. The operations in this class are called NSM.

Def 6.4 *A operation o is not-state-modifying (*NSM*) iff*

$$\forall g, \overline{u}\ (o^s(\overline{u}, g) = g)$$

[1] In this chapter we will use the following variables: $u \in \text{U}$, $\overline{u} \in \text{U}^*$, $g \in \text{GS}$ and $o \in \text{OPS}$. These variables can be subscripted with natural numbers, like u_1.

		Dependency		
		NSD	LSD	OPS
Modifiability	NSM	constant	attribute	function
	LSM		method	
	OPS			procedure

Table 6.1: Classification of operations

As with state-dependency, there is a class of operations can modify only the local states of the set of actual parameters. These operations are called LSM:

Def 6.5 *An operation o is local-state-modifying (LSM) iff*

$$\forall g, \overline{u}\ (o^s(\overline{u}, g) =_{U \setminus \overline{u}} g)$$

Note that NSM is a real subset of LSM and that there are operations which are not LSM. Therefore LSM is a real subset of OPS: So, now we have the following inclusions:

$$\text{NSM} \subset \text{LSM} \subset \text{OPS}$$

We cannot give any inclusion relations between NSD and LSM or between LSD and NSM. In this sense the two aspects of the classification are orthogonal. However, we can combine the two aspects, which would lead to more classes of operations, as can be seen in the next section.

6.2 The Formal Classification

In the two previous sections two classifications were given. In this section we will merge the two classifications into one classification. This merging is a simple operation of combining all the possible classes with eachother, which finaly results in something like table 6.1.

The class NSD-NSM is called constant. The reason for this is that these kind of operations always have the same object-result and state-result and therefore we can identify these operations with their object-result. In AFSK-D these operations are used to describe, for example, Pascal constants or constants like the natural numbers.

The class LSD-NSM is called attribute. In object-oriented languages attributes are used to describe features that certain objects have. For example, a file-object as described in section 5.3 have an attribute `Size(f)`. This attribute cannot

change the global state and it depends only on the local state of the file f. The class attribute as defined here also includes operations with more than one parameter, for example plus(n,m) from figure 3.1. However, in this case we do not want to call plus an attribute.

The class LSD-LSM is called method since the operations in this class correspond with the methods in object-oriented languages. A method is an operation which can change the local state of its actual parameters. Besides that it is dependent on these local states. For example, the operation Write(f,b) changes the local state of f.

The class OPS-NSM is called function. The reason for this is that we think a function is an operation without side-effects. This definition of a functions conflicts with Pascal functions, since Pascal functions can have side-effects. However, functions in COLD [FJ92], in MLCM [GdL93] and in FLEA [GdL95] can all be modeled as operations of this class. In fact, we think that the keyword function in Pascal is a bit misleading, since the only difference between Pascal functions and procedures is that the latter do not have a result whereas the former do have a result. Therefore, Pascal-functions and procedures are almost the same sort of operations, which, as we assume, was for the developers of Modula-2, a successor of Pascal, a good reason to use the keyword procedure for both sorts of operations.

The class OPS-OPS, or just OPS, is called procedure. This class contains all operations and the meaning of a *procedure name* in AFSK-D is an operation. Pascal functions and procedures can be described with operations of this class.

6.3 Describing the Classification in AFSK-D

It is possible to describe the classification in AFSK-D. We will do this by using the language constructs DEPSTATE and MODSTATE. A one-parameter-operation with the procedure name P which belongs to the class NSD can be defined in AFSK-D as follows:

```
P(d) DEPSTATE                    % a NSD-operation
```

i.e., the list of objects on which P(d) depends is empty. In a similar way an LSD operation is given by:

```
P(d) DEPSTATE d                  % a LSD-operation
```

For procedures with more than one parameter, we can give similar descriptions. If we replace DEPSTATE with MODSTATE we get NSM and LSM operations respectively:

```
P(d) MODSTATE                    % a NSM-operation

P(d) MODSTATE d                  % a LSM-operation
```

If an operation P is both NSD and LSM, then we can use the two AFSK-D terms corresponding to these classes. We will call these terms patterns, since they can be used to remind you how to define an operation of a certain class. For example, the pattern for an LSD-NSM operation is:

```
P(d) DEPSTATE d
P(d) MODSTATE
```

If an actual operation of this class with the procedure name operation and with four parameters is to be defined, we use this pattern to write:

```
operation(a,b,c,d) DEPSTATE a,b,c,d
operation(a,b,c,d) MODSTATE
```

And if we see the operation defined in this way, we recognize the "LSD-NSM pattern" in it, and therefore know to what class it belongs.

In the examples in the previous chapter we have already defined some operations using this or other patterns. For example, the operation Release_Block is an LSD-LSM operation, see figure 5.2.

So, in the examples we defined different sorts of operations using patterns of terms. It would be nicer if we could just introduce a procedure name with a special keyword and then leave out the terms which defined to which class the operations belongs. For example, we could use the keyword FUNC to introduce operations of the class OPS-NSM, as in:

```
FUNC MaxVar
```

This defines that the operation associated with the procedure name MaxVar is a function and therefore we could read this term as an abbreviation for:

```
MaxVar() MODSTATE
```

Other keywords which could be usefull are CONST, ATTR, METHOD and PROC, for respectively the classes named as constant, attribute, method and procedure. These keywords not only would reduce the number of terms in a specification, but it also improves the readability of the specification. A procedure name introduced with the keyword ATTR directly suggests that it is an attribute. In contrast to this take a look at the terms:

```
F(p) MODSTATE
F(p) DEPSTATE p
```

These terms also define that F denotes an operation of the class LSD-NSM, but the association with attributes is much harder to make.

However, AFSK-D is a kernel language. This means that we do not want to add all kinds of different keywords to AFSK-D unless they really increase the

expressive power of the language. For the keywords given here, it is clear that they will not increase the expressive power of AFSK-D, since they are used to define operations which can be defined with other AFSK-D terms as wel. Therefore we do not add these keywords to AFSK-D, but we think it would be very nice to have such keywords in the user language AFSL-D that is based on AFSK-D.

In this chapter a formal classification of operations was given. The idea of the classification arose whenwe used AFSK-D for the examples of the previous chapter. In these examples we discovered several syntactic patterns to define operations. We have defined, according to these patterns, different classes of operations based on global state dependency and global state modifiability. It turnèd out that the common sorts of operations, which are considered different in practice, could be classified as different sorts of operations in our formal framework.

7 Conclusions

In this chapter we will draw some conclusions about the material that we have provided in the previous chapters. Also we will mention some areas related to AFSK-D which could be interesting for further research.

In the introduction we have said that this report describes the language AFSK-D. For the development of AFSK-D we used some guidelines, see page 2. The first guideline said that the underlying concepts of AFSK-D should be simple.

We think we have not violated this guideline, since we only used some basic cpo theory to describe the formal semantics of AFSK-D, see chapters 2 and 4. Furthermore, our formalisation of the "objects with a local state"-view, which is usually informally described in the OO-literature, is very direct. We can, however, not say that this is the best way to give a formalisation of object orientation, but it surely is not the most difficult one. Finally, the concept of terms both denoting a value (or an object) and a side-effect is not new. In COLD this approach can be found, too.

The second guideline said that AFSK-D should be object oriented, or, at least, it should be usable in an object-oriented environment. In the introduction we have already mentioned that AFSK-D is not object-oriented according to [Boo94]. However, as said above, we have given a formalisation of a very important concept of the object-oriented paradigm, namely: "objects with local states". This formalisation is at the core of the semantics of AFSK-D. So, the influence on AFSK-D of the object-oriented paradigm can not be denied.

Finally, AFSK-D should contain usual programming constructs. When we take a look at the syntax of AFSK-D in definition 3.1, we see that AFSK-D has IF..THEN..ELSE..FI, a WHILE..DO..OD and a LET..IN..NI terms together with sequential composition. Besides that, a user can define procedures, which then can be used in other terms. AFSK-D does not have an *assignment* construct for (programming) variables. The reason for this is that programming variables can be described as normal objects, see section 5.1. This means that there is no need to extend the language with programming variables and an assignment construct to change the values of these variables. Furthermore, if we would have included programming variables in AFSK-D this could lead to confusion, because then we would have had both programming and logical variables.

A final conclusion we want to draw is about the classification of operations given in chapter 6. AFSK-D has only one sort of operations, therefore a classification based on the semantics of operations is possible. The point we want to make is that, at least for a kernel language, it is not necessary to introduce different sorts of operation when introducing the syntax of a language. Just define one sort of operations, make sure that the semantics of these operations is general

enough, and then define a classification based on the semantics. The advantage of this approach is that it keeps the syntax smaller and, as a consequence, less semantical definitions are needed. An advantage of the classification is that the differences between the classes of operations is formalised in terms of the semantics of operations.

Further research should be done to define the user-laguage AFSL-D. This language should contain an executable sub-language in which executable programs could be written. Since AFSL-D will be an user-language, a lot of attention should be given to the implementation of tools to assist the AFSL-D user. One of the tools we think of is a (semi-)automatic proof-checker with which one could give proofs about the specifications. In order to establish this, there must be an axiomatization of and a proofsystem for the logic of AFSK-D. This will need thorough research in the semantics of AFSK-D.

Bibliography

[Boo94] Grady Booch. *Object-oriented Analysis and Design* with Applications. Benjamin Cummings, second edition, 1994.

[dB80] Jaco de Bakker. *Mathematical Theory of Program Correctness.* series in computer science. Prentice Hall, 1980.

[FJ92] Feijs, L.M.G. and Jonkers, H.B.M. *Formal Specification and Design.* Cambridge Tracts in Theorethical Computer Science. Cambridge University Press, 1992.

[FJM94] Feijs, L.M.G., Jonkers, H.B.M, and Middelburg,C. A. *Notations for Software Design.* FACIT. Springer-Verlag, 1994.

[GdL93] Rix Groenboom and Gerard R. Renardel de Lavalette. Reasoning about dynamic features in specification languages. *Proceedings of Workshop in Semantics of Specification Languages*, pages 340–355, October 1993.

[GdL95] R. Groenboom and G.R. Renardel de Lavalette. A formalisation of evolving algebras. In *Proceedings of Accolade 95*, pages 17–28, Amsterdam, 1995. Dutch Graduate School in Logic.

[Har84] D. Harel. Dynamic logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume II, pages 497–604. D. Reidel Publishing Company, 1984.

[Jon91] H.B.M. Jonkers. Upgrading the pre- and postcondition technique. Technical Report RWR-531-hj-910128-hj, Philips Research, December 1991.

[Tan87] Andrew S. Tanenbaum. *Operating Systems* Design and Implementation. Prentice-Hall, 1987.

[vBvDKMV94] J.F.A.K. van Benthem, H.P. van Ditmarsch, J. Ketting, and W.P.M. Meyer-Viol. *Logica voor Informatici.* Addison-Wesley, tweede edition, 1994.

[WL88] Jim Woodcock and Martin Loomes. *Software Engineering Mathematics.* Pitman, 1988.