Specification of the Unix File System: A Comparative Case Study

Maritta Heisel

Technische Universität Berlin
FB Informatik – FG Softwaretechnik
Franklinstr. 28-29, Sekr. FR 5-6
D-10587 Berlin, Germany
heisel@cs.tu-berlin.de
fax: (+49-30) 314-73488

Abstract. The starting point of this investigation are two different formal specifications of the user's view of the Unix file system, one algebraic and one model-based. The different features exhibited by the specifications give rise to a discussion of desirable and undesirable properties of formal specifications.

1 Why yet another specification of the Unix file system?

The Unix file system is one of the best (or at least most) specified software systems. Several versions have been published: [1, 3, 5][1]. All of these versions are distinguished in important aspects, e.g. in the view that is considered or the executability of the specification.

We will present yet another version, and it is not even new in the above-mentioned aspects. It models the user's view of the Unix file system, just like the specification of Bidoit, Gaudel and Mauboussin [1]. In fact, it was inspired by this specification. When investigating language-independent issues of specifications [7], we thought it a nice exercise to express the given specification, written in the algebraic language PLUSS, in the model-based language Z [8], where the Z specification was to resemble the PLUSS specification as close as possible. Had this enterprise succeeded, no need would have arisen for yet another paper on the Unix file system.

The Unix file system presents itself to the user as a tree where each node has a name and an arbitrary number of successors. A specification of such trees should be present in some library for re-use, where the content of the nodes (as opposed to their names) should be a generic parameter. The first trial to define such trees indeed looked promising:

$$[NAME]$$

$$\begin{aligned} TREE[X] ::= & \; lf\langle\langle NAME \times X \rangle\rangle \\ | & \; node\langle\langle NAME \times \text{seq } TREE[X] \rangle\rangle \end{aligned}$$

[1] Our apologies to all those not mentioned here.

It was discussed with a Z specialist (who will remain unnamed here) and approved. What an unpleasant surprise that the Z type checker rejected this specification! A look at the Z grammar showed that the checker was right (of course). Free types in connection with genericity are indeed not allowed in Z.

What began with thorough disappointment ended up in a lesson on the aesthetics of formal specifications. We were able to make a virtue out of necessity and came up with a specification that looks entirely different than the one we started out from, although it basically models "the same thing". From our point of view, the new Z specification is not inferior to the original one in PLUSS. It turned out that the strengths and weaknesses of the two versions lie in different areas, so that it is virtually impossible to prefer one over the other without reservations.

The aim of this paper is to stimulate a discussion on the various desirable features of formal specifications and how they can be achieved. In the following, we present parts of the specification in two versions[2]. Important differences are contrasted, and it is tried to distill the lessons to be learned from the example and to come to a better assessment of the qualities of formal specifications.

2 Re-Usability

That re-usability is a desirable property of specifications is undisputed. For our example, it is our intention to start out from a generic specification of named trees, and instantiate and adapt this specification to define directories.

2.1 Generic Specification of Named Trees

It seems perfectly clear that trees are defined as recursive data structures, doesn't it? The following PLUSS specification is recursive even if it does not look like it at first sight: it uses lists which are defined recursively.

proc *NAMED_TREE*(*X*)
  use LIST, NAME
  sorts Named_Tree
  cons <_ . _>: (Name × X) × List(Named_Tree) → Named_Tree
  func
    ...
  axiom
    ...
end *NAMED_TREE*(*X*)

As already stated, our trial to define trees in Z recursively was no success. Z does not support recursive definitions very well since it is part of the "Z philosophy" to define types as *sets*. It turns out that adhering to the "Z philosophy"

[2] where we do not stick literally to the specification given in [1].

yields a modeling of trees in which all the necessary functions can be expressed quite elegantly. This modeling is *not* recursive.

In Z, lists are called sequences; they are defined as finite partial functions from the natural numbers into some type X. Similarly, named trees will be finite partial functions from sequences of positive natural numbers into the Cartesian product $NAME \times X$.

$$
\begin{array}{l}
[NAME] \\
NAMED\_TREE[X] == \\
\quad \{f : \operatorname{seq} \mathbb{N}_1 \pfun NAME \times X \mid \\
\qquad \langle\rangle \in \operatorname{dom} f \\
\qquad \land\ (\forall\, path : \operatorname{seq}_1 \mathbb{N}_1 \mid path \in \operatorname{dom} f \bullet \\
\qquad\quad front\ path \in \operatorname{dom} f \\
\qquad\quad \land\ (last\ path \neq 1 \Rightarrow front\ path \frown \langle last\ path - 1\rangle \in \operatorname{dom} f))\}
\end{array}
$$

This definition models trees as functions mapping "addresses" to the content of the node under the respective address. Each node consists of a name and an item of the parameter type X. The empty sequence is the address of the root. The length of an address sequence coincides with the depth of the node in the tree. The number i denotes the i-th subtree. Hence, an address can only be valid if its *front* is also a valid address. And if there is an i-th subtree for $i \geq 1$ then there must also exist an $i-1$-th subtree.

2.2 Discussion

By refraining from using free types we managed to obtain a generic tree definition in Z that may be re-used later. But this was mere luck.

Issue 1 *Is there a convincing reason why genericity in connection with free types is forbidden in Z?*

In comparison with PLUSS and other algebraic languages, Z's support for genericity and re-use is poor. In contrast, PLUSS, even offers a **param** construct that makes it possible to state restrictions on the types used as actual parameters in the instantiation of generic specifications. This feature is not too frequent in specification languages.

There might be members of the Z community who would oppose to this opinion. Wordsworth [10], p.25, for instance, states that genericity can be achieved by introducing the parameters of the specification as basic types. The following would indeed have been legal:

$$
\begin{array}{l}
[NAME, X] \\
\\
TREE ::= lf\langle\!\langle NAME \times X \rangle\!\rangle \\
\qquad\quad \mid\ node\langle\!\langle NAME \times \operatorname{seq} TREE \rangle\!\rangle
\end{array}
$$

An instantiation of this "generic" definition would redefine X:

$$X == \ldots$$

Issue 2 *Is the use of basic types and their later redefinition a satisfactory solution for genericity?*

For example, what happens if more than one instantiation of the "generic" specification is needed in one and the same specification?

3 Descriptive vs. Recursive Specifications

Without further functions allowing one to manipulate and access named trees, the above definitions would not be of much use. We compare the different manners of specifying such functions as they are supported by the different languages.

3.1 Completing the Generic Specifications

We first give a more complete specification of named trees in PLUSS. For those familiar with algebraic languages, it bears no surprise. Lists are assumed to be defined by a constant *nil*, a constructor function "/", and selector functions *the head of_* and *the tail of_*.

proc NAMED_TREE(X)
  use LIST, NAME
  sorts Named_Tree
  cons <_ . _>: (Name × X) × List(Named_Tree) → Named_Tree
  func
    the name of_: Named_Tree → Name
    the content of_: Named_Tree → X
    the children of_: Named_Tree → List(Named_Tree)
    the name list of_: List(Named_Tree) → List(Name)
    the number of children of_: Named_Tree → Integer
    the child of _ named _: Named_Tree × Name → Named_Tree
  pred
    _is leaf: Named_Tree
  precond forall n:Name, t:Named_Tree
    child: the child of t named n **is defined when**
      n belongs to the name list of t
  axiom forall n, n':Name, t:Named_Tree, x:X, l:List(Named_Tree)
    name: the name of <(n,x) . l> = n
    cont: the content of <(n,x) . l> = x
    st: the children of <(n,x) . l>) = l
    nl1: the name list of nil = nil
    nl2: the name list of t / l = the name of t / the name list of l
    nb: the number of children of <(n,x) . l> = length(l)
    isl: <(n,x) . l> is_leaf **iff** l = nil
    child1: n = the name of the head of the children of t
      ⇒ the child of t named n = the head of the children of t

child2: n $\neq$ the name of the head of the children of <(n',x) . l>
$\Rightarrow$ the child of t named n
= the child of <(n',x) . the tail of l> named n
end NAMED_TREE(X)

The corresponding definitions in Z look as follows, where we also define some auxiliary functions on $NAMED\_TREE[X]$ which have no counterpart in the PLUSS specification.

$$
\begin{array}{|l}
\hline
[X] \\
\hline
child : NAMED\_TREE[X] \times \mathbb{N}_1 \pfun NAMED\_TREE[X] \\
number\_of\_children : NAMED\_TREE[X] \fun \mathbb{N} \\
children : NAMED\_TREE[X] \fun \mathbb{P}\, NAMED\_TREE[X] \\
leafs : NAMED\_TREE[X] \fun \mathbb{P}(\mathrm{seq}\, \mathbb{N}_1) \\
\hline
\mathrm{dom}\, child = \{t : NAMED\_TREE[X]; i : \mathbb{N}_1 \mid \langle i \rangle \in \mathrm{dom}\, t\} \\
\forall\, t : NAMED\_TREE[X]; i : \mathbb{N}_1 \bullet \\
\quad \langle i \rangle \in \mathrm{dom}\, t \Rightarrow \\
\quad\quad child(t, i) = \{s : \mathrm{seq}\, \mathbb{N}_1; nx : NAME \times X \mid \\
\quad\quad\quad\quad \langle i \rangle \frown s \in \mathrm{dom}\, t \land nx = t(\langle i \rangle \frown s)\} \land \\
\quad \mathrm{dom}\, t \neq \{\langle\rangle\} \Rightarrow \\
\quad\quad number\_of\_children\, t = max\{k : \mathbb{N}_1 \mid \langle k \rangle \in \mathrm{dom}\, t\} \land \\
\quad \mathrm{dom}\, t = \{\langle\rangle\} \Rightarrow number\_of\_children\, t = 0 \land \\
\quad children\, t = \\
\quad\quad \{k : 1 \,..\, number\_of\_children\, t \bullet child(t, k)\} \land \\
\quad leafs\, t = \{s : \mathrm{seq}\, \mathbb{N}_1 \mid s \in \mathrm{dom}\, t \land (\forall\, s1 : \mathrm{seq}\, \mathbb{N}_1 \bullet s \frown s1 \notin \mathrm{dom}\, t)\} \\
\hline
\end{array}
$$

With the help of these functions we can now specify:

$$
\begin{array}{|l}
\hline
[X] \\
\hline
name\_of\_tree : NAMED\_TREE[X] \fun NAME \\
names : \mathbb{P}\, NAMED\_TREE[X] \pfun \mathbb{P}\, NAME \\
child\_named : NAMED\_TREE[X] \times NAME \pfun NAMED\_TREE[X] \\
\hline
\forall\, n : NAME; t : NAMED\_TREE[X]; ts : \mathbb{P}\, NAMED\_TREE[X] \bullet \\
\quad name\_of\_tree\, t = first(t\langle\rangle) \land \\
\quad names\, ts = \{t : ts \bullet name\_of\_tree\, t\} \land \\
\quad n \in names(children\, t) \Rightarrow (t, n) \in \mathrm{dom}(child\_named) \land \\
\quad child\_named(t, n) \in children\, t \land \\
\quad\quad name\_of\_tree(child\_named(t, n)) = n \\
\hline
\end{array}
$$

These are not all, but the most important functions on named trees.

3.2 Discussion

It is noticeable that in the PLUSS specification, functions usually are defined as recursive equations, where the structure of the recursion follows the list constructors. The same holds true for the specification given in [1], except for the fact that there the tree and forest constructors are used as recursion schemas. This means that they are mostly executable and hence almost an implementation. In the Z specification, the functions are given as closed mathematical expressions.

Unfortunately, one confession must be made here: the above specification of *child_named* is semantically invalid in Z because in the reference manual [8] it is required that "the predicates must define the values of the constants uniquely for each value of the formal parameters." This is not the case here, because *child_named* selects an *arbitrary* child with the given name, whereas the PLUSS specification is deterministic and constructive. "Legal" possibilities would be to either define a relation instead of a function or give an unambiguous definition like in PLUSS. Since the type checker cannot find this "violation", it is hard to prevent (or even detect!) specifications like this.

However, we do not see any difficulties with a definition like the one for *child_named*. On the contrary, it has the advantage to give an implementor the greatest possible freedom: if it is more efficient to search from the back to the front instead of vice versa, it should be possible to do so. The PLUSS specification prohibits this and thus may prevent an efficient implementation. It is even questionable here if one should actually require *child_named* to be a function in the mathematical sense. One could argue that it suffices when the result it yields has the given name.

Issue 3 *Should we strive for very high-level specifications that anticipate as few implementation details as possible?*

To put it in other words: Is it satisfactory to specify functions as recursive equations along some constructors, as suggested by many algebraic specification languages?

It should be noted that this is not a language issue, but an issue of style. It would well be possible to specify functions as closed terms when using an algebraic language with a sufficiently expressive logic (i.e. more expressive than the one of PLUSS). The point is that this would be more complicated because something like the mathematical toolkit of Z had to be predefined. However, once this were done, algebraic specifications could look very much the same as in Z, as far as the use of recursive equations is concerned. It seems that the tradition to define functions with recursive equations stems from the time when algebraic specification languages had an initial semantics. In these times, there was no other possibility indeed.

4 Modularity

The next step in the specification of the user's view of the Unix file system is to specify directories, re-using the specification of named trees. It turns out that it is by far not enough to instantiate the generic parameter.

4.1 Specifying Directories

Before this can be done, the generic specification itself has to be modified in order to make it possible to use names for navigation in the tree. This can be done with *paths*. Paths are nonempty lists of names:

spec PATH **as** NONEMPTY-LIST(NAME)

$PATH == \text{seq}_1\ NAME$

The next step is to define functions working on the combination of named trees and paths. For this purpose, we have to define a predicate *is_existing_path_of* that decides if a path is valid for a given tree, and the functions *object_at_in*, *pruned_at*, and *plus_added_under* which select an item, prune the tree or add a new subtree under a given path. We only present the definition of *is_existing_path_of*.

proc NAMED_TREE_WITH_PATH(X)
  use NAMED_TREE(X), PATH
  pred
    is existing path of: Path × Named_Tree
  axiom forall n, n':Name, p: Path, t:Named_Tree, x:X, l:List(Named_Tree)
    exist1: n / nil is existing path of t **iff** n = the name of t
    exist2: n / n' / nil is existing path of t
      iff n = the name of t **&** n' belongs to the name list of t
    exist3: n' belongs to the name list of t **is false**
      ⇒ n / n' /p is existing path of t **is false**
    exist4: n' belongs to the name list of t
      ⇒ n / n' /p is existing path of t **iff** n = the name of t
        & n' / p is existing path of the child of t named n'
end NAMED_TREE_WITH_PATH(X)

It should be noted that we had the choice between two kinds of clumsiness here. Bidoit, Gaudel and Mauboussin [1] preferred to define paths from scratch. This made it possible to embed names into paths, i.e. to define every name to be a path. However, this specification is 33 lines long. We found that a bit much and preferred the one-line definition of paths as nonempty lists of names. The price for this is that we have to write "n / nil" where Bidoit, Gaudel and Mauboussin only have to write "n". This inconvenience will occur again in Section 5.1 when we define relative paths.

$$
\begin{array}{|l}
\hline
[X] \\
\_is\_existing\_path\_of\_ : PATH \leftrightarrow NAMED\_TREE[X] \\
\hline
\forall\, t : NAMED\_TREE[X]; p : PATH \bullet \\
\quad p\ is\_existing\_path\_of\ t \Leftrightarrow \\
\qquad (head\ p = name\_of\_tree\ t \wedge \\
\qquad (tail\ p \neq \langle\rangle \Rightarrow (\exists\, t_1 : children\ t \bullet tail\ p\ is\_existing\_path\_of\ t_1))) \\
\hline
\end{array}
$$

We note that in PLUSS, a new generic specification is defined (without new constructors), whereas in Z a new global generic definition is added. Now, the actual parameters can be defined:

spec UNIX-NODE **as** FILE ∪ { dir }
spec DIRECTORY **as** NAMED_TREE_WITH_PATH(UNIX-NODE)

where *FILE* defines files as being either text files or binary files.

To finish the specification of directories, we must further specify some constraints related to the nodes: (i) a file may only be a leaf node; (ii) all successors of a node have different names. These constraints on the data type cannot be added to the parameter or to the generic specification but only to the whole instantiated generic specification:

axiom forall d: Directory, n: Name, i,j: Integer
file: the content of d is a file ⇒ d is leaf
inj: n = the name of the i th element of the children of d
& n = the name of the j th element of the children of d
⇒ i = j

For Z, we define:

$$UNIX\_NODE ::= dir \mid file\langle\langle FILE\rangle\rangle$$

$$DIRECTORY == NAMED\_TREE[UNIX\_NODE]$$

The global constraint that has to be added is

$$\forall d : DIRECTORY; n : NAME; i, j : \mathbb{N} \bullet \\ \forall p : \mathrm{dom}\, d \bullet (second(d\, p) \in \mathrm{ran}\, file \Rightarrow p \in leafs\, d) \wedge \\ \#(names(children\ d)) = \#(children\ d)$$

Requirement (ii) can be expressed somewhat more elegantly in Z because the Z specification is based on sets instead of lists. But even if sequences had been used, it would be possible to directly express that the sequence must be injective.

4.2 Discussion

Modularity is a very desirable feature for formal specifications. First, it is important for re-use. Libraries of predefined specifications should contain relatively small and self-contained modules so that they can serve as a kind of construction kit. Second, it is very hard to read, comprehend and maintain large, unstructured formal specification documents. Unfortunately this is exactly what Z forces its users to build. There is no possibility of nesting specification constructs (importing of schemas is just a shorthand for textually copying the content of the

imported schema) or grouping parts of specifications together to form a new entity. Hence, the whole specification is spread out at the top-level.

In PLUSS, for instance, it is possible to import named trees without paths. In Z, you get all or nothing: once you have defined an instance of *NAMED_TREE*, you also have defined the operations dealing with paths, no matter if you want them or not.

Object oriented versions of Z are under development [6, 4] that provide better facilities for modularizing specifications. However, this is not a solution to the problem. Currently, Z is being standardized. The existence of a standard is of some importance to industry. They seem to prefer standardized products over others. If the standard will not contain better facilities for modularizing Z specifications, hundreds of industrial specifiers and programmers will have to live with the poor structuring facilities of Z.

Issue 4 *Can poor language facilities be compensated for by a better specification discipline?*

It it our experience that Z specifications can be well readable. This can be achieved by detailed comments (which are strongly advocated by the Z methodology) and by a skillful layout of the specification. But perhaps the question should be asked the other way around: why do specifiers *have* to compensate for poor language facilities?

5 Freely Generated Data Types and Z

Not only absolute, but also relative paths (relative to a given path) can be used to navigate in the directory tree. These are best specified as an abstract data type.

5.1 Defining Relative Paths

In PLUSS, relative paths, called displacements, can be defined straightforwardly:

spec RELATIVE_PATH
  use PATH
  sort Displacement
  cons
    empty_d: → Displacement
    _ : Path → Displacement
    _ / _: Displacement × Name → Displacement
    ../ : Displacement → Displacement
  func
    _ || _ : Path × Displacement → Path
  axiom forall p: Path, n: Name, dp: Displacement
    cat1: p || empty_d = p
    cat2: p || (n / nil) = p / n

cat3: p || (dp / n) = (p || dp) / n / nil
cat4: (n / nil) || ../dp = (n nil) || dp
cat5: (p / n) || ../dp = p || dp
end RELATIVE_PATH

Paths are embedded into displacements, i.e. each path is also a displacement. In Z, this is impossible. There, we have to define a function d that converts a path into a displacement (see below). The list constructor "/" is overloaded here; overloading is allowed in PLUSS. Again, we have to write "n / nil" instead of "n".

As long as no genericity is involved, we can use free types in Z:

$$\begin{aligned} DISPLACEMENT ::= & \; empty\_d \\ | & \; d\langle\langle PATH\rangle\rangle \\ | & \; (\_/\_)\langle\langle DISPLACEMENT \times NAME\rangle\rangle \\ | & \; ../\langle\langle DISPLACEMENT\rangle\rangle \end{aligned}$$

This definition corresponds to the **cons** part of the PLUSS specification. The function || has to be defined by an axiomatic box.

$$\_ \,||\, \_ : PATH \times DISPLACEMENT \pfun PATH$$

$$\begin{aligned} \forall\, & p : PATH; n : NAME; dp : DISPLACEMENT \bullet \\ & p \,||\, empty\_d = p \land \\ & p \,||\, d\,\langle n\rangle = p \frown \langle n\rangle \land \\ & p \,||\, (dp/n) = (p \,||\, dp) \frown \langle n\rangle \land \\ & \langle n\rangle \,||\, (../dp) = \langle n\rangle \,||\, dp \land \\ & (p \frown \langle n\rangle) \,||\, ../dp = p \,||\, dp \end{aligned}$$

Disregarding the lack of modularity of the Z specification, see Sect. 4, both specifications of relative paths are adequate. Therefore, nothing needs to be discussed and no new issues need to be raised.

6 State-Based Systems and Algebraic Languages

It is convenient to consider the user's view of the Unix file system as a *state* that can be changed by user commands.

6.1 Defining the User's View

We are now ready to define the system state: it consists of a directory, and two paths, one for the home directory and one for the working directory. In Z, such system states are easily defined by a schema:

OneUserView

$root : DIRECTORY$
$home\_dir : PATH$
$working\_dir : PATH$

$home\_dir\ is\_existing\_path\_of\ root$
$second(object\_at\_in(home\_dir, root)(\langle\rangle)) = dir$
$working\_dir\ is\_existing\_path\_of\ root$
$second(object\_at\_in(working\_dir, root)(\langle\rangle)) = dir$

As an example of a Unix command, we consider the command *cd* which changes the working directory. Since *cd* can be called with various parameters, we have to define several schemas for this operation, due to the strong typing of Z. If no argument is supplied to *cd*, the working directory is set to the home directory by default.

cd_def

$\Delta OneUserView$

$root' = root$
$home\_dir' = home\_dir$
$working\_dir' = home\_dir$

If an absolute path is supplied to *cd*, the working directory is set to this path, provided it is a legal one. Legal means that the path exists in the directory and that it leads to a directory, not to a file.

cd_abs

$\Delta OneUserView$
$p? : PATH$

$p?\ is\_existing\_path\_of\ root$
$second(object\_at\_in(p?, root)(\langle\rangle)) = dir$
$root' = root$
$home\_dir' = home\_dir$
$working\_dir' = p?$

If a displacement is supplied to *cd*, the new working directory is computed as the absolute path yielded by combining the old working directory with the given displacement.

cd_rel

$\Delta OneUserView$
$dp? : DISPLACEMENT$

$(working\_dir \mid\mid dp?)\ is\_existing\_path\_of\ root$
$second(object\_at\_in(working\_dir \mid\mid dp?, root)(\langle\rangle)) = dir$
$root' = root$
$home\_dir' = home\_dir$
$working\_dir' = working\_dir \mid\mid dp?$

To define state-based systems in algebraic languages, one possible "schema" (similarly to the definition of freely generated types in Z) is to first define a *data type* S (instead of a schema) modeling the global state. If the state schema consists of more than one variable, S has to be defined as the Cartesian product of the types of the state variables. The state invariant must be given as a global axiom on S. Each operation in a state-based system is specified by a function having the state before execution of the operation as an additional input parameter and the state after execution of the operation as an additional output parameter. Generally, the axioms for such a function are the conjunction of the axioms for the state definition of the "before"-state, the "after"-state and the axioms defining the operation. We are lucky: for our Unix example, it is possible to use a simpler version, although we now have the obligation to show that each *cd* function yields indeed a legal state.

spec ONE_USER_VIEW
  use *DIRECTORY*, *RELATIVE_PATH*
  sort User-view
  cons $< - . - . - >$: Directory × Path × Path → User-view
  func
    cd: User-view → User-view
    cd: User-view × Path → User-view
    cd: User-view × Displacement → User-view
    . . .
  precond forall root: Directory, hd,wd, p: Path, dp: Displacement
    cd1: cd(<root.hd.wd>,p) **is defined when**
      p is an existing path of root
      & the object at p in root is a Directory
    cd2: cd(<root.hd.wd>,dp) **is defined when**
      cd(<root.hd.wd>, wd || dp) **is defined**
  axioms forall root: Directory, hd,wd, p: Path, dp: Displacement,
      uv:User-view
    state: uv = <root.hd.wd>
    ⇒ hd is an existing path of root
      & the object at hd in root is a Directory
      & wd is an existing path of root

& the object at wd in root is a Directory
cd1: cd(<root.hd.wd>) = <root.hd.hd>
cd2: cd(<root.hd.wd>,p) = <root.hd.p>
cd3: cd(<root.hd.wd>,dp) = <root.hd.wd||dp>
end ONE_USER_VIEW

We observe that, in PLUSS, there is no need to invent different names for the different versions of the *cd* command because overloading is permitted.

6.2 Discussion

The general approach to define state-based systems in algebraic languages seems to be a bit clumsy. In the worst case, it could be necessary to repeat the state invariant over and over again. We are not aware of any satisfactory solutions to this problem but would be glad to learn about them if they exist.

Issue 5 *Are there more elegant ways to deal with states in algebraic languages?*

It seems that this issue cannot be neglected by the algebraic specifications community. For implementation purposes, efficiency considerations will probably always play a major role because the impressive MIPS numbers of new computer generations are eaten up by the ever more complicated, comfortable, and large new programs that are written (and demanded!) for them. Research on functional programming languages takes this fact into account. In this area, it is even more evident that it is impractical to copy large data structures for each function call that is executed. There are approaches to avoid this and nevertheless retain referential transparency [9]. Similar ideas for algebraic specification languages would be most welcome.

7 The Moral of the Story

What do we learn of this comparative case study? The good news is that we indeed succeeded in specifying the system we wanted. The languages available today are basically useful. The bad news is that sometimes they seem to create more problems than they solve. This is irrespective of the fact that much research activity has been devoted to the design of specification languages.

There seems to be a large gap between the algebraic and the model-based communities. But in reality, in every system specification there will be parts where clean algebraic properties are wanted and other parts where a state is necessary. This means we cannot (or at least do not want) to do without one or the other of these features. Both means of expression are useful and necessary. Ignoring this fact, algebraic languages pretend there is no need for a state, and model-based languages pretend there is no need for comfortable, abstract and encapsulated data type definitions.

Additionally, we learned that a good specification discipline and style can make up for many deficiencies of today's specification languages. However, this can be no substitute for real language support.

Both stylistic and language issues come into play when we ask ourselves what makes up a "good" specification. Some of the properties discussed in the preceding sections are not disputed, e.g. re-usability or modularity. At most the best ways to achieve them are subject of discussions. Concerning the call for a very high level of abstraction in formal specifications, this is probably different. Some people strongly argue in favor of executable specifications because of their prototyping potential. This is an issue where it is hard to choose between conflicting goals: it is our strong conviction that specifications should not enforce premature implementation decisions ; on the other hand, animating specifications is very important because – as Brooks [2] put it – "For the truth is, the client does not know what he wants". It is a fact of life that system requirements cannot be fixed from the beginning and will continue to change during the project[3]. In the end, it is like everywhere: easy solutions cannot be expected.

Acknowledgment. We would like to thank Thomas Santen for many stimulating discussions and for his comments on this work.

References

1. M. Bidoit, M.-C. Gaudel, and A. Mauboussin. How to make algebraic specifications more understandable? In M. Wirsing and J.A. Bergstra, editors, *Algebraic Methods: Theory, Tools and Applications*, number 394 in LNCS, pages 31 – 67. Springer-Verlag, 1989.
2. Frederick P. Brooks. No silver bullet – essence and accidents of software engineering. *Computer*, pages 10–19, April 1987.
3. O. Declerfayt, B. Demeuse, F. Wautier, P. Y. Schobbens, and E. Milgrom. Precise standards through formal specifications: A case study: the unix file system. In *Proceedings EUUG Autumn Conference, Cascais, Portugal*, 1988.
4. Kevin Lano and Howard Haughton. Specifying a concept-recognition system in Z++. In Kevin Lano and Howard Haughton, editors, *Object-Oriented Specification Case Studies*, chapter 7, pages 137–157. Prentice Hall, 1988.
5. Carroll Morgan and Bernard Sufrin. Specification of the UNIX Filing System. In Ian Hayes, editor, *Specification Case Studies*. Prentic-Hall, 1987.
6. Gordon Rose and Roger Duke. An object-Z specification of a mobile phone system. In Kevin Lano and Howard Haughton, editors, *Object-Oriented Specification Case Studies*, chapter 5, pages 110–129. Prentice Hall, 1993.
7. Jeanine Souquières and Maritta Heisel. How to manage formal specifications? Submitted for publication, 1994.
8. J. M. Spivey. *The Z Notation – A Reference Manual*. Prentice Hall, 2nd edition, 1992.
9. Phil Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi*, volume 118 of *Computer and Systems Sciences*, pages 233–264. Springer-Verlag, 1993.
10. J. B. Wordsworth. *Software Development with Z*. Addison-Wesley, Wokingham, 1992.

[3] In this respect our case study was unrealistic because it specified an already existing system.

This article was processed using the LaTeX macro package with LLNCS style