Expression of Styles in Formal Specification

Jeanine Souquières and Maritta Heisel

CRIN—INRIA-Lorraine[*] and Technische Universität Berlin[**]

Abstract. This paper presents a framework for supporting the acquisition of formal specifications. It is possible to identify certain *styles* or orientation approaches according to which the specification process is performed. These styles are used locally, i.e. even in one development, one switches between different styles. Therefore, it is not reasonable to identify styles with specification languages, as is usually the case. We propose to *explicitly* represent styles as sets of development operators in a *process oriented* way which is independent of the respective specification language that is used. Our approach is illustrated by a case study where of subset of the Unix file system is specified.

1 Introduction

In recent years, much progress has been made in the design of specification languages. On the other hand, *tools* for specification development usually support semi-formal methods. These often use a graphic notation that is particularly suitable for human readers.

Each formal specification language comes with a semantics that associates a meaning with each construct offered by the language. Of course, different languages offer different constructs. Some constructs may be used very elegantly, others only in a clumsy way. Thus, they implicitly represent certain *specification styles*. However, representing a specification style does not mean supporting it. One of the reasons that formal specification is not widely used in industry is that the concrete application of specification languages is not supported in a satisfactory way[3]. It does clearly not suffice to hand a language description to specifiers and then expect them to be able to produce formal specifications.

Today, software engineers in industry mostly use semi-formal techniques like data-flow diagrams, entity-relationship diagrams, finite state machines or decision tables/trees. These techniques have the advantage that they support the intuitive understanding of specifications by a graphic representation. Moreover, sophisticated tool support is widely available. For a survey of semi-formal specification techniques and available tools, see [McD91]. The drawback of semi-formal

[*] B.P. 239 Bâtiment LORIA, F-54506 Vandœuvre-lès-Nancy Cedex, souquier@loria.fr

[**] FB Informatik – Softwaretechnik, Franklinstr. 28-29, Sekr. FR 5-6, D-10587 Berlin, heisel@cs.tu-berlin.de

[3] Exceptions are Z and VDM [Spi92, Jon90]. This is probably the reason why these languages receive some recognition by industry, see e.g. [CGR93].

methods is that they do not provide an unambiguous semantics and thus can be subject to misinterpretations. This makes them an insecure basis for the development contract.

These observations suggest to combine the usage of formal languages with satisfactory tool support. Machine support should include

- bookkeeping,
- guidance of the specifier through the specification process by giving hints what has to be done in which order,
- a graphical representation of the development state,
- recording the particular way the specification has been developed and justifications for the choices made during this process.

A system offering all the above features exists in a prototype version [BS93].

We strongly advocate that the development of a specification be oriented on the problem, not on the specification language that is used. When developing a formal specification, we should find out if the system that is to be built will have a global state that is changed by the operations of the system or if it is more suitable to abstractly describe the properties of the system and its operations. It should also be checked if it is possible to combine and adjust existing specifications to obtain a specification for the new system. Depending on the answers to these questions, a specifier will follow different paths to develop the specification. These make up different *specification styles*. We briefly mentioned three of them: *state-based*, *algebraic*, and *re-use*. As will be illustrated by an example in the following sections, it is convenient to make use of different styles during development of a larger specification.

A specification style describes a certain "spirit" in which a specification is set up. The aim of the re-use style is to re-use specifications contained in a library whenever possible. Of course, one cannot expect to set up a new specification exclusively using existing specifications of a library. Usually, library items will have to be modified, and certain parts will have to be developed from scratch. Hence, a style is nothing strict.

We vote for making specification styles *explicit* instead of representing them implicitly by specification languages. Specification styles can be described and used independently of the specification language to a large extent. With this approach, we can support specifiers even when they are forced, e.g. by the company policy, to use a specification language that is not suited best for a given problem.

Throughout the paper, we consider the specification of the user's view of the Unix file system [BGM89]. The system to be specified is a *tree* of files and directories, where the root and the inner nodes of the tree are always directories, and the leaves of the tree are either files or empty directories. Each node has a name and can have an arbitrary number of successors. Sibling nodes must have different names. The user can navigate in this tree, add and remove directories and files, and access information stored in the system. This specification models the Unix file system from a different point of view than the one presented in [MS87] where the "machine view" is considered.

Section 2 briefly describes our framework which forms the basis for the explicit representation of specification styles. Section 3 demonstrates how the re-use style can be supported by development operators in a language independent way. Sections 4 and 5 present the algebraic and state-based styles. The relation to previous work is described in Section 6. A summary of our approach concludes the paper.

2 The Development Framework

In the proposed framework [Sou93], [SD95] the specification process is modeled as a problem solving process in which *tasks* have to be solved. Initially, a specifier has the task to develop a specification for some informally described system. This task is decomposed into smaller ones by application of *development operators*. These also establish a *precedence relation* on the generated subtasks. Successive application of development operators results in a graph of tasks, called *workplan*.

Workplan. During the development of the Unix file system specification, a task of the form *Specify Directory* may be reduced by a development operator to the three subtasks *Specify generic part*, *Specify Directory actual parameters* and *Achieve Directory specification*. This is represented graphically as in Figure 1.

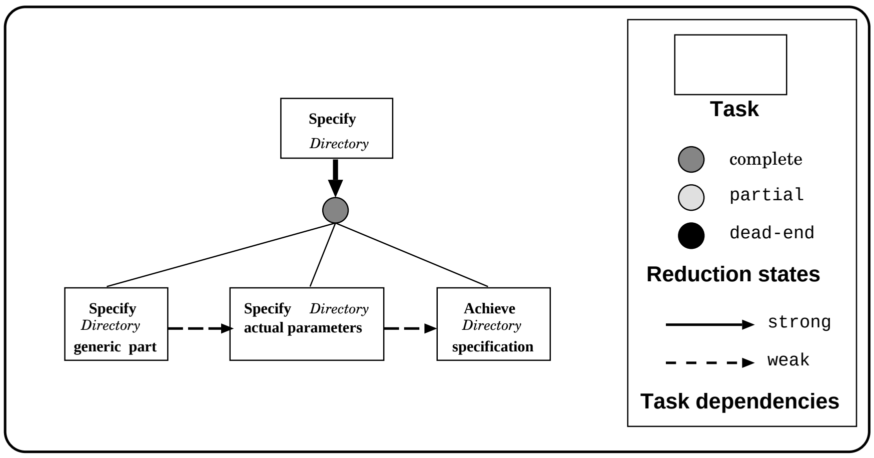
Specify Directory
Specify Directory generic part
Specify Directory actual parameters
Achieve Directory specification
Task
complete
partial
dead-end
Reduction states
strong
weak
Task dependencies

Fig. 1. A workplan example

Solving a task corresponds to reducing it, i.e. decomposing it into subtasks. As in the classical top-down reduction approach to problem solving [Nil71], workplans are modeled by *and/or*-graphs with two kinds of nodes: *task* nodes and *reduction* nodes. The workplan graph is acyclic and has a unique root node. *And* links represent the reduction of tasks into subtasks. *Or* links represent alternative reductions.

To each reduction node, a *state* is assigned: a reduction is *partial* if its list of subtasks can be extended, *complete* if its list of subtasks cannot be extended or a *dead-end* if the decomposition cannot progress towards the objective associated with the parent task. Reduction states are denoted by different kinds of reduction nodes. In Figure 4, five complete and one partial reduction are shown.

Moreover, a developer might wish to express that reducing some task requires some other ones to be reduced first. This point motivates the introduction of *terminated* reductions and *dependency relations* in the framework. Two kinds of dependency relations have been introduced. *Strong precedence* means that one task may not be reduced while another task has a current non-terminated reduction. *Weak precedence* means that reduction of one task may start as soon as the reduction of another task has made *sufficient progress*. This informal condition can only be decided by the specifier.

Product. In parallel with the workplan, a formal specification is set up. It is called *product* and consists of both formal and informal text. The role of the informal text is to aid understanding of the formal text. During the development of a specification, an incomplete part of the product is represented by a typed *placeholder* (denoted by ... in Figure 2[4]). Placeholders are instantiated with a product schema or replaced by formal or informal text via application of development operators.

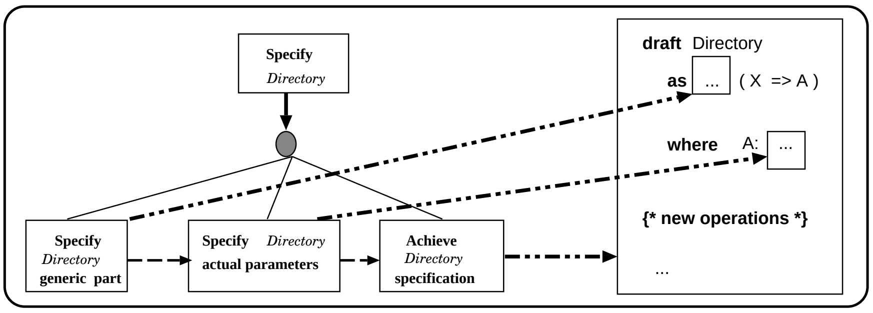
Specify Directory
Specify Directory generic part
Specify Directory actual parameters
Achieve Directory specification
draft Directory
as ... (X => A)
where A: ...
{* new operations *}
...

Fig. 2. Workplan–Product links

Workplan–Product Links. The workplan must be related to the product. We require that each task must be related to at least one placeholder and each reduction must be related to a product version. Conversely, each placeholder is associated with a unique task and a unique goal stated in the workplan. Links are represented in Figure 2 by dotted-dashed arrows between tasks and product components.

[4] In this figure, the **draft** construct of PLUSS is displayed. Drafts are incomplete specifications in that the sets of denoted values need not be fixed. Sorts and operations may be defined without knowing the constructor functions. To turn a draft into a specification, the constructor functions have to be fixed. For details, see [BGM89].

Development operators. They work simultaneously on the workplan and product to reduce tasks and construct or modify the product text. Their parameters are obtained interactively from the specifier and from the current development state. They consist of a language-independent section which describes the action on the workplan and a part concerning the product definition which has to be instantiated with a specification language.

The development process consists in reducing some task, either introducing new subtasks or finishing the work assigned to the task. In the last case, we use a **Terminate** operator whose application results in a complete definition of the placeholders associated to the chosen task without new decomposition.

Specification Styles. In this framework, a style is represented as a collection of development operators. The workplan part of the operators is language independent. For each supported specification language, there is a corresponding product schema implementing its product part.

3 Re-using Existing Specifications

Re-using existing specifications or parts of specifications is one of the big challenges of software development. This concept is supported by specification languages offering genericity and combination of specifications like the *use* construct in PLUSS. However, it has turned out that re-use mechanisms should not only be included in the *languages* but also in the *development process*. Re-using a specification by verbatim inclusion or instantiation of parameters is not realistic. Usually, modifications of re-used specifications are necessary. Two special operators taking into account the necessary modifications when adding a new component [SL93] or an invariant [Lev90] to an existing specification have been defined. In our Unix example, the task is to specify directories. Suppose we have a generic definition of trees called *NAMED-TREE[X]* in our library. A tree has a name, a content of type X and a list of subtrees. To define a directory by means of such a tree a mere instantiation of the parameter X does not suffice: we need the notion of path to make precise the access to nodes via names, and we have to take into account that inner nodes and leaves have to be treated differently. In the following, we perform the steps sketched above by application of development operators associated with the re-use style. The same development is performed with PLUSS as well as Z.

Re-use with PLUSS. As shown in Figure 3, the **Use-generic-spec** development operator decomposes the initial task into three subtasks. This re-use operator is a refinement of the one which has been applyied to obtain the workplan presented in Figure 2 where the first subtask, *Specify <name> generic part*, has been refined by two subtasks, namely *Identify an existing generic specification* and *Extend generic specification*.

The first step, when applying this operator to specify directories, consists in reducing the task *Identify an existing generic specification*. This is simply done by selecting the generic specification *NAMED-TREE[X]* of our library:

$$\textbf{draft}\ \text{DIRECTORY}\ \textbf{as}\ \text{NAMED-TREE}(\ \ldots\ \longrightarrow \text{X})$$

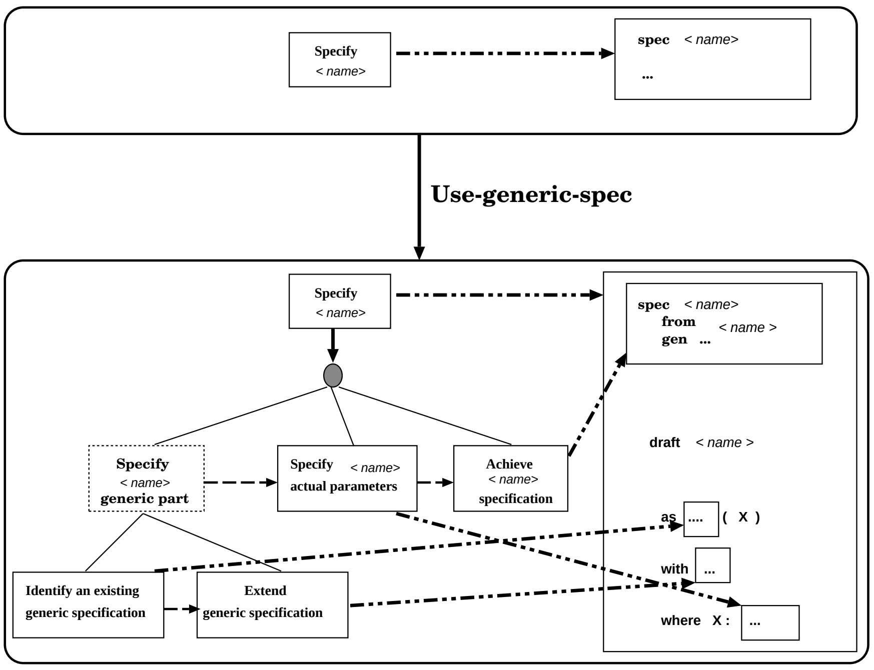

Fig. 3. The Use-generic-spec workplan operator

To continue with the development, we have to reduce the second subtask. To extend the generic specification, we have in mind the idea of combining it with the notion of path. To do this, we apply another operator called **Combine** allowing one to put together different specifications. The different specifications either exist or need to be developed. Here, we combine *NAMED-TREE[X]* with *PATH*. The **Combine** operator generates two new tasks, *Specify intermediates* and *Specify new operations* as shown in Figure 4. We solve the first one by application of an operator called **Directly-reuse**. This operator simply instantiates the parameters of a generic specification. In our case, the predefined type *NONEMPTY-LIST* is instantiated with the data type *NAME*:

$$\textbf{spec}\ \text{PATH}\ \textbf{as}\ \text{NONEMPTY-LIST(NAME)}$$

The next step is to define functions working on the combination of named trees and paths. In Unix, paths are used to navigate in the file system. 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 respectively select an item, prune the tree or add a new subtree under a given path. For each of these, a new task is generated in the workplan. For reasons of space, the development of their specifications cannot be presented here. The corresponding development state is given in Figure 4.

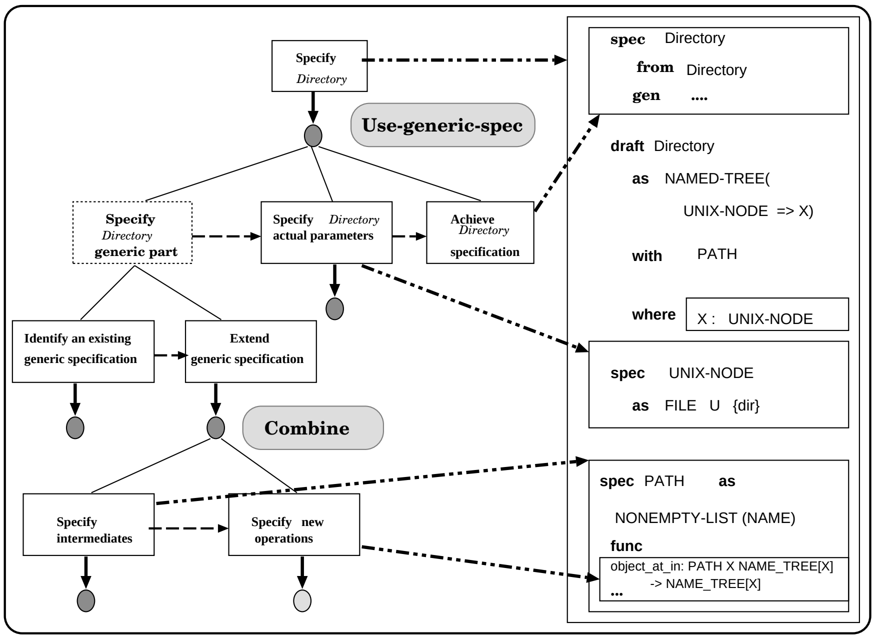

Fig. 4. A development state

The task *Specify Directory actual parameters* can be reduced with **Terminate** and giving the following PLUSS expression:

$$\textbf{spec}\ \text{UNIX-NODE}\ \textbf{as}\ \text{FILE} \cup \{\ \text{dir}\ \}$$

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

Now, in order to *Achieve the specification*, we need to introduce 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 which must then be converted from a **draft** module into a **spec** module.

axioms ∀ d: DIRECTORY, n: NAME, i, j: INTEGER
    Is-file(Content(d)) ⇒ Is-leaf(d)
    (n = Pos(i, Namelist(Subtrees(d))) ∧
        n = Pos(j,Namelist(Subtrees(d)) ⇒ i = j))

Re-use with Z. To specify directories using Z, we can follow exactly the same development process as for PLUSS, but the resulting specification looks somewhat different: here, named trees with parameter type X are defined as finite partial functions from nonempty sequences of natural numbers to the Cartesian product $NAME \times X$. The sequence of natural numbers which is the argument to such a function represents the "address" of a node in the tree. The value of the function is the content of the node. Of course, these functions must satisfy some additional constraints in order to represent a tree. For details, see [Hei95]. The solution of the task *Extend generic specification* is

$$UNIX\_NODE ::= dir \mid file\langle\langle FILE \rangle\rangle$$
$$PATH == seq_1\ NAME$$

where we do not present the additional functions working on trees and paths. The specification is *achieved* by

$$\begin{array}{|l}
DIRECTORY : \mathbb{P}\ UNIX\_NODE \\
\hline
\forall d : DIRECTORY;\ n : NAME;\ i,j : \mathbb{N} \bullet \forall p : dom\ d \bullet \\
\quad (second(d\ p) \in ran\ file \Rightarrow p \in leafs\ d) \wedge \\
\quad (n = namelist(subtrees\ d)(i) \wedge n = namelist(subtrees\ d)(j) \Rightarrow i = j)
\end{array}$$

This finishes the specification of directories. In this example, three different operators associated with the re-use style have been applied: **Directly-reuse** directly re-uses a piece of code by its name, thus completing the task for which it is applied. The other operators expand the workplan by either combining several specifications which have to be defined in turn (**Combine**) or instantiating and modifying existing specifications (**Use-generic-spec**).

Of course, these are not the only conceivable re-use operators. Since styles are just collections of development operators, it is always possible to add new operators to a given style.

4 The Algebraic Style

Specification styles that are usually identified with certain languages are the **algebraic** or **property-oriented** style where the aim of a specification is to describe a system in terms of its desired properties and the **state-based** or **model-oriented** style where the aim is to construct an abstract model of the system. Z and VDM especially support the state-based style, whereas PLUSS or ASL [AW86] are candidates for the algebraic style. In this and the next section, we show that these styles are not so closely tied to specification languages as it might appear.

When specifying algebraically, we define sorts and generators for these sorts. The operations of the system are defined by giving axioms in terms of the chosen generators. Additionally, constraints between generators can be stated. The

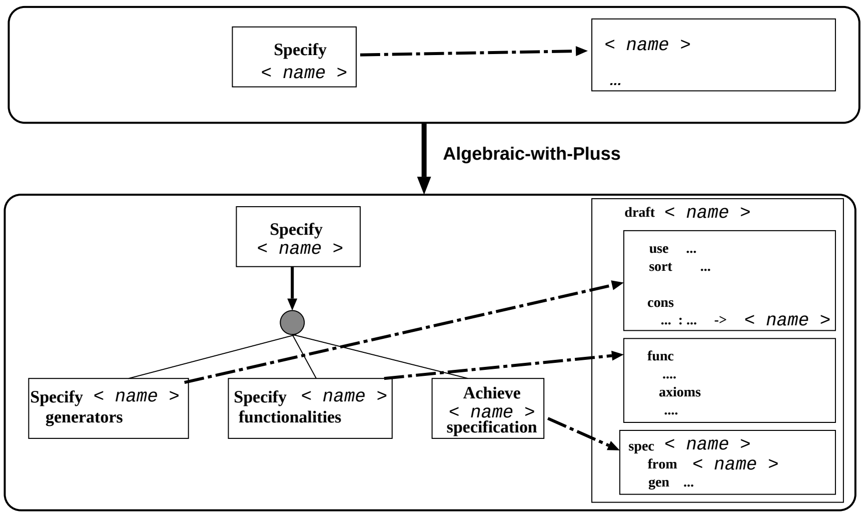

Fig. 5. The Algebraic development operator with the Pluss Language

corresponding development operator is shown in Figure 5 with the PLUSS specification language. For the Unix example, we apply this operator to specify the data type *Displacement* which is needed to define paths relative to a given working or home directory. In doing so, we obtain a new development state with new tasks and their product placeholders as defined in Figure 5. A solution for the first subtask consists in giving the definition of the generators:

draft DISPLACEMENT
    use PATH
    sort Displacement
    cons
        empty_d: $\rightarrow$ Displacement
        d: Path $\rightarrow$ Displacement
        _ / _: Displacement $\times$ Name $\rightarrow$ Displacement
        ../ : Displacement $\rightarrow$ Displacement

Not surprisingly, this kind of procedure works well for algebraic languages like PLUSS. Suppose, however, we want to use a model-based language like Z and nevertheless follow the algebraic style. In many cases, this can be done using the *free type* construct of Z which is very similar to an algebraic data type definition. Difficulties arise when genericity is needed. If only one instance of the generic type is needed then the generic parameters can be defined as basic types. Instantiation is done by giving equations for the basic types. Only if several instances are needed, the free type construct cannot be used and the development operator given in Figure 6 cannot be applied. The other functions defined on the free type are specified with global axiomatic.

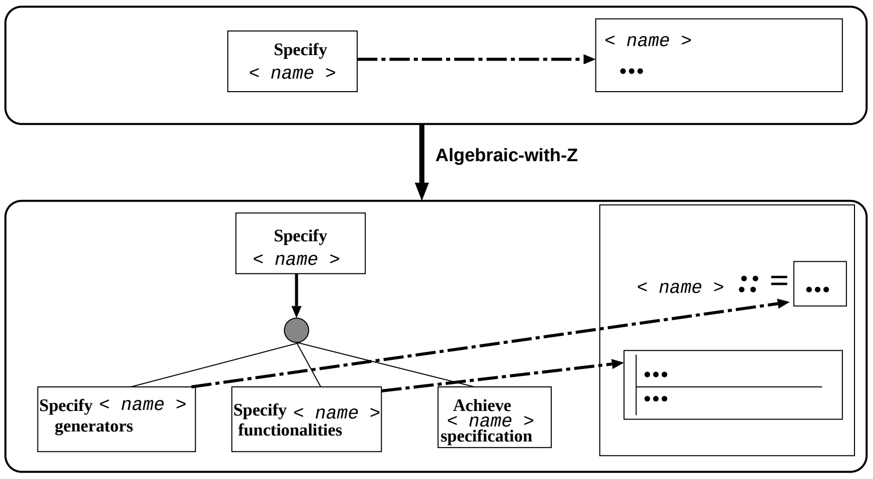
Specify
< name >
< name >
•••
Algebraic-with-Z
Specify
< name >
< name > ::= •••
•••
•••
Specify < name > generators
Specify < name > functionalities
Achieve < name > specification

Fig. 6. The Algebraic development operator with the Z Language

In solving the task *Specify generators* for displacement, we obtain the following Z definition:

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

We can express uniform product parts of the algebraic development operator for PLUSS as well as for Z. This shows that the algebraic approach should not be identified with a class of specification languages [LS94]. Instead, it can be characterized by development operators.

5 The State-Based Style

When specifying a state-based system, we first define the set of admissible states S and an initial state. Then the operations of the system are specified. These operations can take inputs and produce outputs. In addition, they may change the global state. This style of specification is supported particularly well by model-based languages like Z or VDM. In Z, the global state of a system is specified by a *schema*. This schema consists of a declaration part in which the state components are specified, and a predicate part specifying the state invariant, i.e. the conditions a legal system state must satisfy. Operations are defined as schemas which import the global state. The workplan part of the corresponding operator is given in Figure 7.

In our Unix example, it is convenient to consider the tree of files and directories together with the home and working directories as the system state which

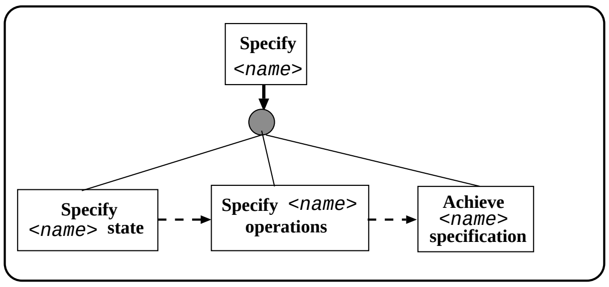

Fig. 7. Workplan associated to the state-based style

may be changed by the application of Unix commands. We give the corresponding schema for Z:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{OneUserView}} \\
\hline
root : DIRECTORY \\
home\_dir : PATH \\
working\_dir : PATH \\
\hline
home\_dir\ is\_existing\_path\_of\ root \\
second(object\_at\_in(home\_dir, root)(\langle\rangle)) = dir \\
working\_dir\ is\_existing\_path\_of\ root \\
second(object\_at\_in(working\_dir, root)(\langle\rangle)) = dir \\
\hline
\end{array}
$$

The state invariant says that both the home and the working directory must be represented by legal paths that lead to directories, not to files.

As an example of a Unix command, we define *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. If an absolute path is supplied, the working directory is set to this path, provided the path exists and leads to a directory. Given a displacement, the new working directory is computed as the absolute path yielded by combining the old working directory with the given displacement. We give the second ("absolute") version:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{cd\_abs}} \\
\hline
\Delta OneUserView \\
p? : PATH \\
\hline
p?\ is\_existing\_path\_of\ root \\
second(object\_at\_in(p?, root)(\langle\rangle)) = dir \\
root' = root \\
home\_dir' = home\_dir \\
working\_dir' = p? \\
\hline
\end{array}
$$

In contrast, algebraic specification languages are not designed to specify systems in this manner. However, there is a uniform way of algebraically specifying state-based systems: we 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 is given as an axiom. 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 the function are the conjunction of the axioms for the state definition of the "before"-state, the "after"-state and the axioms defining the operation. For our Unix example, it is possible to simplify the specification yielded by this uniform procedure: since the value of the function *cd* is directly expressed in terms of the constructor function $< -. - .- >$, it suffices to require the state invariant to hold for all values of the constructor function.

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

In this way, it is possible to apply development operators of the state-based style even when using an algebraic language. A refinement of this operator is proposed in [DS93].

6 Related Work

The aim of our work is to provide satisfactory support of the *specification process*. To this end, we developed the notion of style. Styles encompass *problem* *solving knowledge*. Development operators are a means to represent such knowledge. *Tool support* for our approach makes it applicable in practice. Hence, there are relations to the following areas of research: (i) approaches to support the specification process, (ii) representation of problem solving or design knowledge, and (iii) specification support tools.

Approaches to Support the Specification Process. Johnson and Feather [JF91] take a transformational approach to specification development. Starting out from a simple initial specification, so-called *evolution transformations* are applied. These may change the semantics of the specification and add more detail to it. Compared to these, specification styles and development operators are concepts of a higher level of abstraction and closer to human reasoning.

Hußmann [Huß93] presents a process model for requirements engineering which describes steps to obtain a formal requirements specification from an informal requirements description. A large part of the model deals with finding the appropriate requirements, whereas our approach only comes into play when the requirements are known.

Representation of Problem Solving/Design Knowledge. Wile [Wil83] presents the development language Paddle. Its control structures are called *goal structures*. Paddle programs are a means to express developments, i.e. procedures to transform a specification into a program. A disadvantage of this procedural representation of process knowledge is that it enforces a strict depth-first left-to-right processing of the goal structure.

Potts [Pot89] uses *Issue-based Information Systems* (IBIS) [CB88] to represent design methods. Not only is represented what to do in which order when a design step is performed. *Reasons* for design decisions are also recorded: each design step raises certain *issues*. Different possibilities to resolve an issue are called *positions*. *Arguments* are in favor or against positions. This rich structure causes representations of even small examples to become quite complicated.

Declarative representations like IBIS allow for more flexibility in the problem solving process. The same holds true for the notion of *strategy* presented in [Hei94]. There, problem solving knowledge is represented as *relations* on problems and solutions. Strategies are similar to development operators. The difference is that they are designed to enforce the development of *correct* solutions to problems, where the notion of correctness can be defined freely. As a consequence, the reduction of a problem to subproblems must always be complete.

Specification Support Tools. The requirements apprentice is a part of the programmer's apprentice [RW88]. It uses *requirements clichés* to support its users in setting up a requirements document. Clichés describe prototypical systems, e.g. repositories or information systems. To set up a requirements document, a dialogue in natural language is performed. The resulting document is "conventional", i.e. not formal. Apart from this, it is a question of taste if one chooses to represent knowledge in form of "examples" which can be customized or as general problem solving knowledge.

In [WL93], several systems – mostly for VDM and Z – are presented. Apart from some theorem provers, these can be divided into two classes: either, they perform type-checking or other analyses of a given specification. These systems cannot be used to set up a specification. Or the systems provide editing facilities for the language they support. Editors do not provide a process model and cannot support design decisions.

7 Conclusion

In presenting the Unix case study, we have demonstrated the following:

1. Different approaches for the development of a specification can be identified. We call these *styles* and represent them by sets of development operators. Three such styles have been presented: re-use, algebraic, and state-based. Another style which was not applied here is the object-oriented style.
2. In one specification, several styles may be needed to specify different components. This clearly shows that it is not satisfactory to identify styles with specification languages.
3. Performing the same steps, i.e. applying the same development operators in the same order, it is possible to obtain "equivalent" specifications in different languages. This shows that, to a large extent, the development process can be driven exclusively by the problem.

Specifiers are supplied with machine support, a graphical representation of the specification state and guidance throughout the specification process. Not only the formal specification itself, but also the specification process is represented. Associating one or more specification styles with each development operator helps structuring the library of development operators. Once a user has chosen a certain style, a support system need only offer the operators associated with that style. Thus, the choice of the right operator can be made easier.

In this fashion, the development of formal specifications is supported in a similar way as semi-formal methods, and we can also hope for a similar acceptance of this approach by practitioners. The main activity of the specifier no longer consists in writing down the specification, but in applying predefined operators to evolve the specification.

In summary, our approach permits a flexible modeling of the specification process. General problem solving knowledge is represented in a declarative way. Tool support for this approach is not limited to a special technique or specification language. In this respect, it combines and generalizes ideas presented in the works mentioned in Section 6.

Acknowledgment. This work was partially supported by the ESPRIT Project 2537 (ICARUS) and the French-German PROCOPE program.

References

[AW86] E. Astesiano and M. Wirsing. An introduction to ASL. In *Proc. of the IFIP WG 2.1 Conf. on Program Specifications and Transformations*, 1986.

[BGM89] M. Bidoit, M.-C. Gaudel, and A. Mauboussin. How to make algebraic specifications more understandable: An experiment with the PLUSS specification language. *Science of Computer Programming*, 12:1–38, 1989.

[BS93] G. Bosch and J. Souquières. *Development Editor User Manual.* Esprit2 ICARUS Tool Deliverable No D#48, ICARUS, January 1994.

[CB88] J. Conklin and M. L. Begeman. gIBIS: A hypertext tool for explanatory policy discussions. *ACM Trans. Office Information Systems*, 6:303–331, 1988.

[CGR93] D. Craigan, S. Gerhart, and T. Ralston. An international survey of industrial applications of formal methods. Technical Report NISTGCR 93/626, National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg, MD 20899, 1993.

[DS93] R. Darimont and J. Souquières. A development model: Application to Z specifications. In *Proceedings of the WG 8.1 Conference on Information System Development Process*, 1993.

[Hei94] M. Heisel. A formal notion of strategy for software development. Technical Report 94–28, TU Berlin, 1994.

[Hei95] M. Heisel. Specification of the unix file system: A comparative case study. In V.S. Alagar and Maurice Nivat, editors, *Proc. 4th Int. Conference on Algebraic Methodology and Software Technology*, volume 936 of *LNCS*, pages 475–488. Springer-Verlag, 1995.

[Huß93] H. Hußmann. Zur formalen Beschreibung der funktionalen Anforderungen an ein Informationssystem. Technical Report TUM–I9332, TU München, 1993.

[JF91] W.L. Johnson and M.S. Feather. Using evolution transformations to construct specifications. In Michael R. Lowry and Robert D. McCartney, editors, *Automating Software Design*, pages 65 – 92. AAAI Press, 1991.

[Jon90] C.B. Jones. *Systematic Software Development using VDM.* Prentice Hall, 1990.

[Lev90] N. Lévy. Definition of Add_an_Invariant, a specification construction process operator. Technical Report ForSem-006-R, ICARUS, 1990.

[LS94] N. Lévy and G. Smith. *A Language-Independent Approach to Specification Construction.* Proceedings SIGSOFT'94: Symposium on the Foundations of Software Engineering, New Orleans, USA., December 1994.

[McD91] J.A. McDermid, editor. *Software Engineer's Reference Book.* Butterworth-Heinemann, Oxford, 1991.

[MS87] C. Morgan and B. Sufrin. Specification of the UNIX Filing System. In Ian Hayes, editor, *Specification Case Studies.* Prentice-Hall, 1987.

[Nil71] N.-J. Nilsson. *Problem Solving Methods in Artificial Intelligence.* Mac Graw-Hill, Computer Sciences Series, 1971.

[Pot89] C. Potts. A generic model for representing design models. In *Proc. ICSE 11.* IEEE, 1989.

[RW88] C. Rich and R.C. Waters. The programmer's apprentice: a research overview. *IEEE Computer*, pages 10–25, November 1988.

[SL93] J. Souquières and N. Lévy. Description of the specification development. In IEEE, editor, *Proc. of RE'93*, pages 216–223, San Diego(CA), January 1993.

[Sou93] J. Souquières. *Aide au Développement de Specifications.* Thèse d'état, CRIN, Université de Nancy 1, 1993.

[SD95] Jeanine Souquières et Robert Darimont. La description du développement de spécifications. *Technique et Science Informatiques*, 14(9), novembre 1995. A paraitre.

[Spi92] J. M. Spivey. *The Z Notation – A Reference Manual.* Prentice Hall, 2nd edition, 1992.

[Wil83] D.S. Wile. Program developments : Formal explanation of implementations. *Communication of the ACM*, 26(11):902–911, 1983.

[WL93] J.P.C. Woodcock and P.G. Larsen, editors. *FME '93: Industrial-Strength Formal Methods*, LNCS 670, Springer-Verlag, 1993.

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