Case study in coalgebraic specification: Memory management in the Fiasco microkernel

Hendrik Tews
Inst. Theor. Informatik, TU Dresden, D-01062 Dresden, Germany.
tews@tcs.inf.tu-dresden.de

March 20, 2000

Abstract

This case study applies coalgebraic specification and verification techniques to the FIASCO microkernel developed at Dresden University of Technology. A part of the memory management of FIASCO was specified in the Coalgebraic Class Specification Language CCSL [HHJT98]. With the support of a modern theorem prover (namely PVS [ORR+96]) it was possible to check the source code of FIASCO against this specification. It turns out that, on the basis of coalgebras and with the use of state of the art tools like PVS and CCSL, it becomes feasible to verify practically used operation-system software.

Contents

List of Figures and Program Listings **5**

1 Introduction **6**

2 Introducing Algebraic and Coalgebraic Specification **9**
- 2.1 Algebras 9
- 2.2 Coalgebras 10

3 Coalgebraic Specification, formally **15**

4 Memory Management in FIASCO **24**
- 4.1 Virtual Memory and Address Translation 24
- 4.2 Address Spaces in FIASCO 26

5 The Specification **34**
- 5.1 Physical Memory 35
- 5.2 Virtual Memory 38
- 5.3 The Specification of Space_t 41

6 The Verification **46**
- 6.1 A Space_t Coalgebra 47
- 6.2 The Proof 51
  - 6.2.1 The Assertion switchin_context_up 53
  - 6.2.2 Consistency of Address Mappings 53
  - 6.2.3 Ordering of Bitvectors 54

7 Conclusions **62**

8 Bibliography **64**

List of Figures and Program Listings

4.1 Address transformation . . . 26
4.2 Address transformation in C++ . . . 27
4.3 space_t class declaration . . . 28
4.4 The method switchin_context . . . 29
4.5 The method v_insert, Part I . . . 30
4.6 The method v_insert, Part II . . . 31
5.1 Files in the FIASCO case study . . . 36
5.2 Physical Memory in CCSL . . . 37
5.3 Specification of virtual memory . . . 39
5.4 The enumeration type status_t . . . 42
5.5 Specification of space_t . . . 44
6.1 A model for Space_t, Part I . . . 47
6.2 A model for Space_t, Part II . . . 50
6.3 A model for Space_t, Part III . . . 51
6.4 Definition of bv2addr and addr2bv . . . 55
6.5 Prooftree of the proof for Lemma msb_equal . . . 57

1 Introduction

Coalgebraic specification is a relatively newly developed field in theoretical computer science. It was first proposed by Reichel [Rei95] and further developed by a number of authors, including Jacobs [Jac96, Jac97] and Hensel [Hen99]. Coalgebraic specification is based on so called *coalgebras* as underlying mathematical framework and aims especially at the verification of object oriented programs. The joint project *Logic of Object Oriented Programming* (LOOP) with the University Nijmegen, the Netherlands, aims at the further development of coalgebraic techniques and their application in software verification. One part of the project was the design of the *Coalgebraic Class Specification Language* CCSL [HHJT98] and the implementation of a compiler that translates CCSL specifications into logical theories of the theorem prover PVS [ORR+96, ORSvH95] or ISABELLE [Pau94]. The theorem prover can then be used to reason about specifications, especially to establish refinements between specifications and to create models.

Currently commercially successful operating systems like Unix and Windows are monolithic systems of considerable size. Most of the operating-system functionality (like memory management and hard-disc drivers) is hard-wired into the system and cannot easily be changed by the user. So called microkernels present a different approach in operating system design: A preferably very small kernel implements only the most important operating-system features. Other functionality runs as normal user process and can easily be exchanged. The operating-system group at Dresden University uses a Linux kernel running on top of a L4 microkernel [HHL+97] for their work on a real-time operating system DROPS (see [HBB+98] for more information). For academic reasons they reimplemented the L4 microkernel in C++. This new microkernel FIASCO [Hoh00, Hoh98] contains only about $20,000$ lines of code, written in a high level programming language.

The microkernel FIASCO is a nice challenge for the tools and techniques that were developed in the LOOP project. There was the possibility to prove that the coalgebraic specification technique could well be applied to *real* software. For this I selected one class of the FIASCO sources with the help their originator, Michael Hohmuth. The selected class space_t implements page directories for FIASCO and provides the most basic interface for memory management in the microkernel. The case study itself consisted of the development of a specification for the functionality of this class and the verification of its source code.

There is also a more serious background for this case study. Because any application requires an operating system, one can trust an application only as far as one can trust the operating system. This especially applies to security relevant software, like the generation of signatures. There, the security can be broken if the operating system is not safe against attackers like viruses. So to establish confidence in a security relevant application requires first that the operating system fulfills some basic security properties. This case study tries to answer the question, to which extend formal methods can help to establish the security of a system by proving properties of the operating system.

A second motivation is that certain errors in operating-system programming are extremely difficult to debug (for instance triple page faults). Applying formal methods during the development of the operating system could help to exclude such errors right from the start.

The case study was successful in the sense that it was indeed possible to verify a few properties of the FIASCO source code. And only the available time restricted the number of properties that were tackled in the case study. Therefore, it is fair to say that coalgebraic specification can well be applied to operating-system verification. However, for larger projects it is necessary to optimize the available tool set (see the conclusions in Chapter 7 for an explicit discussion). Concerning the correctness of the scrutinized part of FIASCO one has to say that, for certain states and certain parameters, the methods of the class space_t do not fulfill the specification (see Chapter 6 for details). However, the developer of FIASCO confirmed that all uses of the the class space_t check for this condition, before calling the compromised methods of space_t.

After the case study was completed, it was an even greater challenge to explain the specification, the verification technique, and the results of the case study to my colleagues from the operating-systems group. They had only a vague knowledge about notions like signature, specification, or model, which are the basics to understand this case study.

This paper tries to take advantage of this experience. It shows the details of the case study, but is completely self–contained. The paper is especially directed at a joint audience from both the operating-systems community and the formal-methods community. The necessary definitions are simplified as much as possible and basic set-theoretic notions are explained in detail.

The case study was carried out with PVS version 2.2, patch level 1.46. It makes heavy use of the bitvectors library [BSGM97], which is shipped with PVS. The full sources and proofs of the case study are available for download and experimentation at http://wwwtcs.inf.tu-dresden.de/~tews/vfiasco/. This paper also functions partly as a guide through the PVS sources of the case study. It contains many pointers, explaining in which files the relevant sources can be found. FIASCO is available at http://os.inf.tu-dresden.de/fiasco/.

The paper is structured as follows. Chapter 2 contains a gentle introduction into algebras and coalgebras and explains the key notions of the verification business. Chapter 3 gives a more formal introduction into coalgebraic specification. It can be skipped on first reading. The later chapters of this paper are pretty much self-explanatory, so one can follow the development with only a small background in logic. Chapter 4 explains the problem of the case study: virtual memory and its management. It also shows the source code of FIASCO, which was examined in this case study. Chapter 5 describes the specifications for physical memory, for virtual memory, and for the memory management. The relevant features of CCSL are explained on the way. Chapter 6 describes the correctness proof for the FIASCO memory management. The relevant PVS sources are far too large to be presented here. So Chapter 6 concentrates on the important points and shows three selected problems in detail. The relevant PVS details are explained when they are needed.

Concerning Chapter 3 I want to emphasize the following. It is only included to make this paper self–contained. Important topics of coalgebraic specification are spared out completely and some key notions are not defined formally, because the categorical background is not available in this paper. The definitions in Chapter 3 are optimized for this paper and for easy understanding (in contrast to correctness and generality).

Some parts of this case study were performed during my visit to the University of Nijmegen, the Netherlands. I want to thank the group of Frits Vaandrager at this university for their support and continuous effort to integrate myself into the Dutch social life. I would like to thank Bart Jacobs for many discussions on the subject and Michael Hohmuth for explaining FIASCO and memory management on Intel architectures and for debugging my prose.

2 Introducing Algebraic and Coalgebraic Specification

In this chapter I explain coalgebras and coalgebraic specification on an intuitive level. To make this easier I first recall the most important notions from algebraic specification. A comprehensive presentation of algebraic specification is [Wir90].

2.1 Algebras

Algebraic specification centers around the idea of *constructing* elements of a datatype. Such a construction takes always a number of arguments and yields a new element of the datatype. Algebraic specification focuses therefore on *algebras*, which are functions with a *structured domain* of the following form

$$\boxed{\cdots \quad D \quad \cdots} \longrightarrow D \qquad (*)$$

The set D stands here for the *carrier set* (the set of all elements) of the datatype being defined. The constructions that are allowed in the domain are the Cartesian product ($\times$), disjoint union ($\uplus$)[1], and the function space $X \Rightarrow Y \stackrel{\text{def}}{=} \{f : X \longrightarrow Y\}$.[2] Categorically, this is captured by the notion of a polynomial functor. But it is impossible to include an introduction into *Category theory* here as well. For this paper I rely on an intuitive understanding of *structured domain* and a few other terms, which can be given precise definitions in a suitable context.

The functions $(*)$ are sometimes called *constructors*.[3] The standard example of algebras is the datatype of lists. To keep it simple, I discuss lists of natural numbers here (instead of list over an arbitrary set A). This datatype has two constructors

$$\begin{array}{lcccl} \mathsf{nil} & : & \mathbf{1} & \longrightarrow & \mathsf{List} \\ \mathsf{cons} & : & \mathbb{N} \times \mathsf{List} & \longrightarrow & \mathsf{List} \end{array}$$

where $\mathbf{1} \stackrel{\text{def}}{=} \{\emptyset\}$ is the one element set.[4]

[1] The disjoint union of two sets X and Y is formally defined as $X \uplus Y \stackrel{\text{def}}{=} \{(x,0) \mid x \in X\} \cup \{(y,1) \mid y \in Y\}$. There are two canonical injections, $\kappa_1 : X \longrightarrow X \uplus Y : x \longmapsto (x,0)$ and $\kappa_2 : Y \longrightarrow X \uplus Y : y \longmapsto (y,1)$. Note that there is a bijective correspondence between functions $A \uplus B \longrightarrow C$ and pairs $(A \longrightarrow C) \times (B \longrightarrow C)$.

[2] The function space is sometimes denoted by Y^X.

[3] There is an analogy to the term constructor as used in C++. There, constructors are also required to have a codomain of single type.

[4] Note that there is an isomorphism $\mathbf{1} \Rightarrow X \simeq X$. Instead of introducing constants of type X, I prefer to use functions $\mathbf{1} \longrightarrow X$

The ultimate aim of specification and verification is to derive conclusions about semantic entities (a concrete implementation of lists) by purely syntactic reasoning. The start of the syntactic development is a *signature*. The signature of lists declares a new type symbol (List) and two function symbols (nil and cons) with the above types. In terms of software engineering a signature corresponds to an interface description.

Next we have to fix the domain of the semantic entities, the class of *models of the list signature*. In general, this notion of model is derived from the signature. In our example, the semantic domain are all list algebras. A list algebra α consists of a carrier set X (the interpretation of the type List) and two functions $\alpha_{\mathsf{nil}} : \mathbf{1} \longrightarrow X$ and $\alpha_{\mathsf{cons}} : \mathbb{N} \times X \longrightarrow X$, interpreting the two function symbols nil and cons. Note that there exist list algebras, which have nothing in common with our intuitive understanding of lists. For instance, the set $\mathbb{R}$ of real numbers, the constant 0, and addition form a list algebra. This is in analogy with the fact that there exist many implementations (including many incorrect ones) of a given interface description.

Another important notion that is derived from the signature is that of algebra homomorphisms. For the list example assume we have two list algebras, α and β. A list homomorphism $\alpha \longrightarrow \beta$ is a function $f : X \longrightarrow Y$ between the carriers of α and β that preserves the list structure. Formally, this requires that $f(\alpha_{\mathsf{nil}}(\emptyset)) = \beta_{\mathsf{nil}}(\emptyset)$ and $\forall n \in \mathbb{N},\ x \in X \,.\, f(\alpha_{\mathsf{cons}}(n, x)) = \beta_{\mathsf{cons}}(n, f(x))$. Two list algebras are isomorphic if there exists a bijective list homomorphism between them.

A list algebra α is called *initial* if, for any list algebra β, there exists exactly one list homomorphism $\alpha \longrightarrow \beta$. Any two initial list algebras are isomorphic. A key observation is now that the isomorphism class of initial list algebras captures precisely the intuitive notion of lists. It is a nice exercise to prove that your favorite list algebra is indeed initial.

A second important observation is that the induction principle holds in any initial list algebra. In general, the induction principle can also be derived from the signature. Assume for the list example that α with carrier X is an initial list algebra. A predicate $P \subseteq X$ is closed under the list algebra α if $\alpha_{\mathsf{nil}}(\emptyset) \in P$ and $\forall n \in \mathbb{N}, x \in P \,.\, \alpha_{\mathsf{cons}}(n, x) \in P$. The induction principle now says that any predicate $P \subseteq X$ of the carrier that is closed under the algebra contains all of the carrier: $P = X$. To formulate it in a more familiar version: To prove that a predicate P holds for all lists, it is enough to show that P holds for the empty list and that it holds for $\mathsf{cons}(n, l)$ under the assumption that it holds for l.

Now, one can use a syntactic calculus like first order predicate logic, add the induction principle and start to infer theorems for all initial list algebras (= correct list implementations). Initiality or the induction principle can also be used to define functions in the syntactic calculus (see [JR97] for examples).

2.2 Coalgebras

Coalgebras can be understood as dualized algebras. The general development of coalgebraic specification is very similar to that of algebraic specification. This section introduces coalgebraic specification by discussing the example of possibly infinite sequences. The next chapter contains formal definitions for all notions from coalgebraic specification that are important in this case study. I will therefore include a few forward references in this section pointing to the definitions in the next chapter.

The central notion in coalgebraic specification is that of an *observation*. If an instance of a system (or automaton) is available, one can invoke its functionality (press a button) and then the system reacts in a certain way, yielding an observation. Therefore, coalgebras are functions of the form

$$S \longrightarrow \boxed{\cdots \; S \; \cdots}$$

where S is the set of all possible states of the system. The canonical example for coalgebras is that of *possibly infinite sequences*[5] (hereafter simply called sequences). For an elaborate discussion about sequences see [HJ99]. Sequences support the following operation.

$$\mathsf{next} \quad : \quad \mathsf{Seq} \quad \longrightarrow \quad (\mathbb{N} \times \mathsf{Seq}) \uplus \mathbf{1} \tag{$\dagger$}$$

If we have a sequent $s \in \mathsf{Seq}$ then $\mathsf{next}(s)$ equals either $\emptyset$, the only element of $\mathbf{1}$, or (n, s') for some $n \in \mathbb{N}$ and $s' \in \mathsf{Seq}$. The first case tells us that s is empty. In the second case s is nonempty and has n as its head and s' as its tail. We can then further apply next to s' to observe more elements of s.

The function next is sometimes called a destructor, because it destructs an element of Seq yielding an observation and possibly a successor state.[6] If we were to develop a specification for the type of sequences we could consider ($\dagger$) as a *coalgebraic signature* (see Definition 3.2 on page 15) that declares the coalgebra or destructor symbol next.

Similar to the general development in algebraic specification the next interesting notion is that of a model for the signature of sequences (see Definition 3.4 on page 16). Models for the signature of sequences are *sequence coalgebras*. A sequence coalgebra γ is a pair $(S, \gamma_{\mathsf{next}})$ consisting of a set S as interpretation of the type Seq, and a function $\gamma_{\mathsf{next}} : S \longrightarrow (\mathbb{N} \times S) \uplus \mathbf{1}$. The set S is usually called the *state space*. One such coalgebra (which deliberately does not realize all possible behaviors of sequences) is the set $\mathbb{N}$ of natural number as state space together with the function

$$\gamma_{\mathsf{next}}(n) = \begin{cases} \emptyset & \text{if } n = 0 \\ (n, n-1) & \text{otherwise} \end{cases}$$

To continue the syntactic development one needs a logic to reason about coalgebraic signatures and to restrict the class of models. *Coalgebraic specification* refers to the use of such a logic to model and to reason about hard– or software systems. The development of suitable logics for coalgebras is an active research area, see for instance [Rot00, Cîr99, Röß]. I will introduce a logic for coalgebras in the next chapter. Here I continue the discussion about general properties of coalgebras.

[5] Again, for simplicity I restrict myself to possibly infinite sequences of natural numbers.
[6] Destructors in this paper have nothing in common with destructors of C++.

From the form of coalgebras we can draw some immediate conclusions. First, with coalgebras alone it is not possible to construct states of the system out of nothing. Second, partial operations, which are problematic in algebraic specification, are directly supported by coalgebras.[7] Third, parameterized observations (functions of the form $S \times A \longrightarrow B$) fit into the coalgebraic framework, because they can be modeled in their curried versions (as functions $A \longrightarrow B \Rightarrow C$).[8] In practice it is nice if one does not always have to use currying for parameterized observations. Therefore, in this paper I consider also functions $S \times A_1 \times \cdots A_n \longrightarrow B$ as coalgebras, when all the A_i are different from S. Fourth and most importantly, one can obtain arbitrary long observations by applying the coalgebra successively. Coalgebras are therefore especially suited for modeling systems that should never stop.

Two coalgebras on the same state space can be combined into one.[9] For theoretical investigations it is therefore general enough, to work with single coalgebras (combining all the possible observations into one function). In practise, however, it is convenient to separate different possibilities to make observations and to consider finite sets of coalgebras on the same carrier.

The state space S of a coalgebra is seen as a black box. We know nothing about its internal structure, we can only apply coalgebras to elements of S and look at the observations they yield. Consider two sequence coalgebras $\delta = (P, \delta_{\mathsf{next}})$ and $\gamma = (S, \gamma_{\mathsf{next}})$. It is possible that there exists a relation $R \subseteq P \times S$ relating states from both coalgebras, such that

$$s\,R\,p \quad \text{if and only if} \quad \begin{cases} \text{either} & \delta_{\mathsf{next}}(p) = \gamma_{\mathsf{next}}(s) = \emptyset \\ \text{or} & \begin{array}{r} \exists n \in \mathbb{N} \,.\, \delta_{\mathsf{next}}(p) = (n, p') \wedge \\ \gamma_{\mathsf{next}}(s) = (n, s') \;\wedge\; p'\,R\,s' \end{array} \end{cases} \tag{‡}$$

Such a relation R is called a *bisimulation* (see page 17). Two sequences p and s that are related by R either are both empty or share both the same head element. Additionally, if p and s are nonempty then also their successor states are related by R again. Thus, the sequences p and s have equal length and contain equal elements. Although it might well be that $p \neq s$ it is impossible to distinguish both from outside.

Two states are *bisimilar* (see page 17) if they are related by a bisimulation. Equality between states is an uninteresting notion in coalgebraic specification. Instead proof obligations and theorems often state that two states of the system are bisimilar.

A *coalgebra homomorphism* (see page 17) is a function between the state spaces of two coalgebras that preserves the observations. For the sequences example assume again

[7] The set of partial functions from A to B is isomorphic to the set $A \Rightarrow B \uplus \mathbf{1}$. So partial functions are coalgebras.

[8] There is an isomorphism $(A \times B) \Rightarrow C \;\simeq\; A \Rightarrow (B \Rightarrow C)$. That is, any function taking two arguments in A and B is equivalent to a function taking one argument in A and delivering as result a function which takes arguments in B. Assume we are given a function $f : A \times B \longrightarrow C$ and an element of $a \in A$, we get a function $B \longrightarrow C$ by substituting a for the first formal argument of f. For the other direction we have to destruct the pair of two arguments into its constituents and apply them in order. This isomorphism is usually called after the logician Curry.

[9] This is another familiar isomorphism: $(A \Rightarrow B) \times (A \Rightarrow C) \;\simeq\; A \Rightarrow (B \times C)$. That is, any function into the Cartesian product $B \times C$ is given by a pair of functions into the components B and C.

two sequence coalgebras $\delta = (P, \delta_{\mathsf{next}})$ and $\gamma = (S, \gamma_{\mathsf{next}})$. A function $f : P \longrightarrow S$ is sequence homomorphism $\delta \longrightarrow \gamma$, if for all $p \in P$ the following holds: If p is empty (i.e., $\delta_{\mathsf{next}}(p) = \emptyset$) then also $f(p)$ is empty. Otherwise, if $\delta_{\mathsf{next}}(p) = (n, p')$ then $\gamma_{\mathsf{next}}(f(p)) = (n, f(p'))$.

With a weak restriction on the possible observations it is possible to prove that there always exists a *final* coalgebra [Rut96]. A coalgebra z is final if, for any other coalgebra of the same type, there exists exactly one coalgebra homomorphism into z. The final coalgebra for the signature of sequences has as state space the set $\mathbb{N}^\infty$ with

$$\mathbb{N}^\infty \quad = \quad \{ f : \mathbb{N} \longrightarrow \mathbb{N} \uplus \mathbf{1} \mid \forall n \in \mathbb{N} \,.\, f(n) = \emptyset \;\text{ implies }\; \forall m > n \,.\, f(m) = \emptyset \}$$

If $f \in \mathbb{N}^\infty$ then $\mathsf{next}_\infty : \mathbb{N}^\infty \longrightarrow (\mathbb{N} \times \mathbb{N}^\infty) \uplus \mathbf{1}$ is defined as

$$\mathsf{next}_\infty(f) \quad = \quad \begin{cases} \emptyset & \text{if } f(0) = \emptyset \\ \big(f(0), (n \mapsto f(n+1))\big) & \text{otherwise} \end{cases}$$

Here I use $(n \mapsto f(n+1))$ to denote the function $\mathbb{N} \longrightarrow \mathbb{N} \uplus \mathbf{1}$ that maps every $n \in \mathbb{N}$ to $f(n+1)$. The pair $(\mathbb{N}^\infty, \mathsf{next}_\infty)$ is a sequence coalgebra. It is quite obvious that every sequence coalgebra can be uniquely embedded into $\mathbb{N}^\infty$. Further, the final model $(\mathbb{N}^\infty, \mathsf{next}_\infty)$ captures precisely the set of all possibly infinite sequences. Therefore one usually takes the final coalgebra (and their state spaces) as standard semantics for coalgebraic specification.

There also exists a definition principle and a proof principle for the final coalgebra. In analogy to algebraic specification both are called *coinduction*. Coinductive function definitions allow the construction of elements in the final coalgebra. The proof principle allows to prove that two elements of the final coalgebra are equal.

Assume, that we want to define the function $\mathsf{append}_\infty : \mathbb{N}^\infty \times \mathbb{N}^\infty \longrightarrow \mathbb{N}^\infty$ that concatenates two sequences. If s is infinite (i.e., all successors states of s are nonempty) then $\mathsf{append}_\infty(s, s')$ should equal s. The coinduction definition principle tells us that, to define append_∞, it is enough to define a coalgebra with state space $\mathbb{N}^\infty \times \mathbb{N}^\infty$. This can be done by

$$\begin{aligned} \mathsf{next}_{\mathsf{append}} \quad &: \quad \mathbb{N}^\infty \times \mathbb{N}^\infty \longrightarrow \big(\mathbb{N} \times (\mathbb{N}^\infty \times \mathbb{N}^\infty)\big) \uplus \mathbf{1} \\ \mathsf{next}_{\mathsf{append}}(s_1, s_2) \quad &= \quad \begin{cases} \emptyset & \text{if } \mathsf{next}_\infty(s_1) = \mathsf{next}_\infty(s_2) = \emptyset \\ (n, (s_1, s_2')) & \text{if } \mathsf{next}_\infty(s_1) = \emptyset \;\text{ and }\; \mathsf{next}_\infty(s_2) = (n, s_2') \\ (n, (s_1', s_2)) & \text{if } \mathsf{next}_\infty(s_1) = (n, s_1') \end{cases} \end{aligned}$$

Because $(\mathbb{N}^\infty, \mathsf{next}_\infty)$ is the final coalgebra, we can define append_∞ to be the only coalgebra homomorphism $(\mathbb{N}^\infty \times \mathbb{N}^\infty, \mathsf{next}_{\mathsf{append}}) \longrightarrow (\mathbb{N}^\infty, \mathsf{next}_\infty)$. Note that this definition of append_∞ is independent from the internal structure of $\mathbb{N}^\infty$.

The coinduction proof principle can be used now to proof that

$$\mathsf{append}_\infty(s_1,\; \mathsf{append}_\infty(s_2, s_3)) \quad = \quad \mathsf{append}_\infty(\mathsf{append}_\infty(s_1, s_2),\; s_3) \qquad (\sharp)$$

To proof this equation, it is enough to construct a bisimulation that relates both sides. The relation $R \subseteq \mathbb{N}^\infty \times \mathbb{N}^\infty$ with

$$s \, R \, p \qquad \text{if and only if} \qquad \begin{aligned} &\exists s_1, s_2, s_3 \in \mathbb{N}^\infty \,.\, s = \mathsf{append}_\infty(s_1, \mathsf{append}_\infty(s_2, s_3)) \\ &\qquad \text{and } \; p = \mathsf{append}_\infty(\mathsf{append}_\infty(s_1, s_2), s_3) \end{aligned}$$

suffices. The use of the coinduction principle reduces the infinite proof obligation of (♯) to the finite check if (‡) holds for R.

The next chapter discusses the notions of bisimulation, homomorphism, and coinduction in greater detail and illustrates them with further examples.

The shape of coalgebras and the analogy with black-box behavior make coalgebras ideal candidates for a semantic foundation of object-oriented programming, see [Rei95, Jac96, Jac97, HHJT98] for details. Methods, attributes, and public instance variables are modeled with coalgebras. A class corresponds to a set of coalgebras and an object to an element of the state space. Throughout this paper one can think of coalgebras as (functional) methods that are applied to objects of some class.

A nice introduction into coalgebras and coinduction is [JR97]. In [Rut96] Rutten proves many abstract properties of coalgebras. Both references require some basic category theory. The standard reference for category theory is [ML71], but for computer scientists I would recommend [BW84]. An alternative is to have a look at [Gum99]. There, Gumm introduces coalgebras together with the necessary amount of category theory.

As I said before, there exists a specification language CCSL for algebraic and coalgebraic specification. CCSL was developed at the University Nijmegen, the Netherlands, and Dresden University of Technology, Germany, in a joint project called LOOP[10]. The work in Dresden was done in the scope of the Graduiertenkolleg *Tools for the Effective Use of Parallel and Distributed Computing Systems.*[11] Currently the only sources about CCSL are [HHJT98], Chapter 5 of [Hen99] and the preliminary language report [Tew00a]. This case study was carried out with version 0.6.4 of the CCSL compiler. At the moment version 2 is under development. The CCSL compiler is not publicly available (this is mostly due to the lack of manpower to develop a stable version of the compiler).

The CCSL compiler acts as a frontend to an existing theorem prover. Currently it supports PVS [ORR+96, ORSvH95] and ISABELLE [Pau94]. It accepts class specifications in CCSL, Java, and the Java annotation language JML [LBR99]. The CCSL compiler translates all its input class specifications into equivalent specifications in the logic of the theorem prover. These can then be loaded into ISABELLE or PVS, together with statements the user wants to prove. Other material that is loaded into the theorem prover comprises extensions of the logic with user defined types, functions, and theorems. These can then be considered as belonging to the ground signature and can be used in the class specification.

The syntax of CCSL is close to PVS, but follows the general picture of coalgebraic specifications as presented in the next chapter. As far as necessary, CCSL is introduced together with the specifications in Chapter 5.

[10] Information about the LOOP project is contained at http://www.cs.kun.nl/~bart/LOOP/.

[11] See the URL http://www.inf.tu-dresden.de/TU/Informatik/IRS/Graduierte_RS.html for more information about the Graduiertenkolleg (in German only).

3 Coalgebraic Specification, formally

This chapter gives formal definitions for the key notions of coalgebraic specification. It is included to make this paper self–contained. The following chapters, which explain the specifications and the proofs, can be appreciated with only an intuitive understanding of coalgebras and logic. So it is possible to skip this chapter on first reading and come back if more formal definitions are required.

The definitions of bisimilarity and homomorphism require a non–trivial amount of category theory. For the FIASCO case study, these notions are not essential. Therefore, bisimilarity and coalgebra homomorphisms are only illustrated with the running example of this chapter.

In the following, assume a fixed ground signature, containing a set of predefined types, typed function symbols, and typed constants. This ground signature provides us with basic types like the natural numbers and **1**. When defining the semantics, the interpretation of this ground signature in principle is completely free. But the specifications developed in this paper only make sense with a standard interpretation of the ground signature where, for instance, the set of natural numbers is used as an interpretation of the type $\mathbb{N}$.

Definition 3.1 (Admissible types) The set of admissible types is the least set containing

- the special type Self,
- any type τ from the ground signature,
- for any two admissible types τ and σ the product $\tau \times \sigma$, the disjoint union $\tau \uplus \sigma$, and, under the condition that τ does not contain Self, the function space $\tau \Rightarrow \sigma$.

The type Self will denote the state space of coalgebras. The condition that Self must not occur on the left side of an arrow $\Rightarrow$ corresponds to the restriction of using only coalgebras for polynomial functors. This excludes observations that act on two states simultaneously (they are called binary methods in object oriented programming).[1]

Definition 3.2 (Coalgebraic signature) A coalgebraic signature consists of a finite set of coalgebra declarations

$$c : \ \mathsf{Self} \times \sigma_1 \times \cdots \times \sigma_n \ \longrightarrow \ \tau$$

for admissible types $\tau, \sigma_1, \ldots, \sigma_n$, where the types $\sigma_1, \ldots, \sigma_n$ do not contain Self.

[1] There are several workarounds. But this is advanced material, not necessary for this paper. At the end of this chapter I give some pointers for further reading.

Pure coalgebra symbols $c : \mathsf{Self} \longrightarrow \tau$ can occur in a signature by taking $n = 0$. Note that a signature does not contain any functions. It only consists of syntactic information: a set of names together with type declarations. I will use capital Greek letters like Σ to denote signatures without concrete names.

Example 3.3 (Memory) A memory is characterized by two operations. The first is read. It is parameterized by a type Addr of addresses and yields results in a type Data. The second operation is write which can be used to modify the contents of the memory. As a result it yields the updated memory.

$$
\begin{array}{l}
\textbf{Signature}\ \mathsf{Memory} \\
\quad \begin{array}{lcccl}
\mathsf{read} & : & \mathsf{Self} \times \mathsf{Addr} & \longrightarrow & \mathsf{Data} \\
\mathsf{write} & : & \mathsf{Self} \times \mathsf{Addr} \times \mathsf{Data} & \longrightarrow & \mathsf{Self}
\end{array}
\end{array}
$$

This signature says nothing about the ability of the memory to remember values at certain addresses. I will fix this in the following. To get a memory which can also store addresses, we either have to enrich this signature or assume conversion functions between Addr and Data. ∎

There exist different approaches to coalgebraic specification. Cîrstea uses in [Cîr99] a restricted set of signatures and obtains a sound and complete deduction calculus. Kurz and Hennicker extend in [HK99] coalgebraic signatures with algebraic extensions.

With coalgebras alone it is impossible to create initial states. To overcome this problem one can enrich coalgebraic signatures with constructor symbols $\mathsf{new} : \mathsf{Self}$ or even parameterized constructor-function symbols $\mathsf{new} : \sigma \longrightarrow \mathsf{Self}$. In the latter case the type σ must not contain Self.[2]

The logic I use later to state properties about coalgebras is parameterized by a signature. The expressions from which formulae will be built can contain coalgebra symbols from that signature. To give this logic a semantics (i.e., to define which formulae are true and which are false), it is necessary to have an interpretation for the ground signature as well as for the type Self and for the coalgebra symbols.

Let us focus on the interpretation of the ground signature first: An interpretation I fixes actual sets $[\![\tau]\!]$ for all types τ in the ground signature. If we also have an interpretation for Self, then I induces an interpretation for all admissible types, by substituting the set-theoretic constructions for Cartesian product, disjoint union, and function space for the symbols $\times, \uplus$, and $\Rightarrow$, respectively. The interpretation of the ground signature also provides actual functions $[\![f]\!] : [\![\tau]\!] \longrightarrow [\![\sigma]\!]$ for function symbols $f : \tau \longrightarrow \sigma$ and elements $[\![k]\!] \in [\![\tau]\!]$ for constants $k : \tau$ of the ground signature.

Definition 3.4 (Model of Signature) Assume a fixed interpretation I of the ground signature. For a signature Σ a model M of Σ consists of a set S (the interpretation of Self) and functions $[\![c]\!]_M : S \times [\![\sigma_1]\!] \times \cdots \times [\![\sigma_n]\!] \longrightarrow [\![\tau]\!]$ for each coalgebra symbol $c : \mathsf{Self} \times \sigma_1 \times \cdots \times \sigma_n \longrightarrow \tau$ of Σ. The sets $[\![\tau]\!], [\![\sigma_1]\!], \ldots, [\![\sigma_n]\!]$ are the interpretation of $\tau, \sigma_1, \ldots, \sigma_n$ induced by I. If M is clear from the context, I omit the subscript.

[2] The copy constructors of C++ are ordinary methods from the point of view of coalgebraic specification.

Example 3.5 Let me continue Example 3.3 and develop a model for the signature Memory. I fix an interpretation I of the ground signature, such that $[\![\mathsf{Data}]\!]$ is the set of all bitvectors of length 32, and $[\![\mathsf{Addr}]\!]$ is the set of natural numbers less than 2^{30}. As state space I take the set S of all functions $[\![\mathsf{Addr}]\!] \longrightarrow [\![\mathsf{Data}]\!]$; $S \stackrel{\text{def}}{=} [\![\mathsf{Addr}]\!] \Rightarrow [\![\mathsf{Data}]\!]$. Let $f \in S$ be such a function, then

$$\begin{array}{rcl} [\![\mathsf{read}]\!](f, a) & = & f(a) \\ [\![\mathsf{write}]\!](f, a, d) & = & a' \longmapsto \begin{cases} d & \text{if } a = a' \\ f(a') & \text{otherwise} \end{cases} \end{array}$$

Here $a' \longmapsto \cdots$ denotes the function that maps a' to the value described at the right hand side. ∎

As next I discuss the notions of bisimilarity and coalgebra homomorphism. Let us assume we have a model M of the memory signature and two elements m_1 and m_2 from its state space. If a user wants to compare both states, she is surely not interested whether the internal representation is the same. Instead she would be interested in the question whether read operations performed at the same addresses yield the same values, and whether this property holds again after performing the same write operations. This weaker notion of equality is usually called behavioral equality or bisimilarity.

To describe bisimilarity for the memory example, I first have to define the notion of *bisimulation*. A bisimulation is a relation on the state space that is closed under the coalgebra. Assume a model M of the memory signature with state space S. A binary relation $R \subseteq S \times S$ is called a Memory–bisimulation, if for all $m_1, m_2 \in M$ with $m_1 \, R \, m_2$ and for all $a \in [\![\mathsf{Addr}]\!]$ and $d \in [\![\mathsf{Data}]\!]$ it is the case that

$$\begin{array}{rcl} [\![\mathsf{read}]\!](m_1, a) & = & [\![\mathsf{read}]\!](m_2, a) \\ [\![\mathsf{write}]\!](m_1, a, d) & R & [\![\mathsf{write}]\!](m_2, a, d) \end{array}$$

The definition of bisimulation can be derived directly from the signature. It follows from general properties of coalgebras that bisimulations are closed under union, intersection, and composition. *Bisimilarity* is then defined as the largest bisimulation. For all possible signatures, the corresponding notion of bisimilarity is an equivalence relation.

A *coalgebra homomorphism* between two models of a signature is a function f between the state spaces, which is compatible with the coalgebras. In general this notion is also derived from the signature. For the memory example assume a model M with state space S and a model O of the memory signature with state space Q. A function $f : S \longrightarrow Q$ is a Memory–homomorphism, if

$$\begin{array}{rcl} [\![\mathsf{read}]\!]_M(s, a) & = & [\![\mathsf{read}]\!]_O(f(s), a) \\ f([\![\mathsf{write}]\!]_M(s, a, d)) & = & [\![\mathsf{write}]\!]_O(f(s), a, d) \end{array}$$

It follows from general properties of coalgebraic signatures of Definition 3.2 that in the class of all models of a given coalgebraic signature Σ, there always exists a final one. A model Z of Σ is *final*, if for any model M, there exists exactly one coalgebra homomorphism $M \longrightarrow Z$. This definition does not uniquely determine the final model, but any two final models are isomorphic. The unique homomorphism from M to Z is denoted by $!_M$. A general construction how to obtain the final model is contained in [Rut96]. For a smaller set of admissible types, an explicit description is also contained in [Jac96].

The special property of the final model is that any other model can be uniquely embedded (via a coalgebra homomorphism) into the final one. Because coalgebra homomorphisms are compatible with the behavior of the model, it follows that the final model can realize all possible behaviors. Intuitively, one can think of the final model as the union of a sufficiently large set of models, quotient by the largest bisimulation on that union.

Example 3.6 Let me develop the final model of the memory signature as an illustration. Its state space Z is the set of all functions that map lists of $\mathsf{Addr} \times \mathsf{Data}$ to the function space $\mathsf{Addr} \Rightarrow \mathsf{Data}$.

$$Z \quad \stackrel{\text{def}}{=} \quad \mathsf{List}[\mathsf{Addr} \times \mathsf{Data}] \;\; \Rightarrow \;\; (\mathsf{Addr} \Rightarrow \mathsf{Data})$$

The coalgebras that act on Z are a bit complicated. Let nil denote the empty list and let $\mathsf{cons}((a,d),\omega)$ be the operation that appends the pair (a,d) at the head of list ω. The coalgebras for the final memory are then

$$\begin{array}{rcl} [\![\mathsf{read}]\!]_Z(z,a) & = & z(\mathsf{nil})(a) \\ [\![\mathsf{write}]\!]_Z(z,a,d) & = & \omega \longmapsto z\big(\mathsf{cons}((a,d),\omega)\big) \end{array}$$

For the construction of !, assume another model M on state space S of the memory signature. First we need the utility function write^*, which takes a state $s \in S$ and a list of $\mathsf{Addr} \times \mathsf{Data}$ pairs and performs successive write operations with the data of the list.

$$[\![\mathsf{write}^*]\!]_M(s,\omega) \;=\; \begin{cases} s & \text{if } \omega = \mathsf{nil} \\ [\![\mathsf{write}^*]\!]_M([\![\mathsf{write}]\!]_M(s,a,d),\omega') & \text{if } \omega = \mathsf{cons}((a,d),\omega') \end{cases}$$

Now it is possible to define $!_M : S \longrightarrow Z$ as

$$!_M(s) \quad = \quad \omega \longmapsto \big(a \longmapsto [\![\mathsf{read}]\!]_M([\![\mathsf{write}^*]\!]_M(s,\omega),\, a)\big) \qquad \blacksquare$$

In the following I develop a simple logic for coalgebraic signatures. Logical formulae will express certain properties of the coalgebras. Then we can focus on those models that fulfill a given set of properties, thereby restricting our attention to *well-behaved* coalgebras.

To keep things simple, I only consider a special first-order logic in this paper.

Definition 3.7 (Terms and Formulae) Let G be a ground signature and Σ a coalgebraic signature. The set of Σ–terms is the least set containing

- (typed) Variables $x : \tau$ for any admissible type τ, constants $a : \sigma$ and function symbols $f : \tau \longrightarrow \sigma$ from the ground signature G

- the projections $\pi_1(t) : \tau$ and $\pi_2(t) : \sigma$ for a term $t : \tau \times \sigma$ of product type
- the pair $(s, t) : \sigma \times \tau$ for terms $s : \sigma$ and $t : \tau$
- for any admissible type $\tau \uplus \sigma$ the injections $\kappa_1(t) : \tau \uplus \sigma$ and $\kappa_2(s) : \tau \uplus \sigma$ for terms $t : \tau$ and $s : \sigma$
- the pattern matching $\mathsf{CASES}\ s\ \mathsf{OF}\ \kappa_1 x : t_1,\ \kappa_2 y : t_2\ \mathsf{ENDCASES} : \rho$ for a term $s : \sigma \uplus \tau$ and two terms t_1, t_2 of the same type ρ
- the application $f(t) : \sigma$ for term $f : \tau \longrightarrow \sigma$ and $t : \tau$
- the coalgebra invocation $c(s, t_1, \ldots t_n) : \tau$, for all terms $s : \mathsf{Self}, t_1 : \sigma_1, \ldots t_n : \sigma_n$ and coalgebra declarations $c : \mathsf{Self} \times \sigma_1 \times \cdots \times \sigma_n \longrightarrow \tau$

The set of Σ–formulae is the least set containing

- equality $t = s$ for terms $t, s : \tau$ of an admissible type τ not containing Self
- bisimilarity $t \underline{\leftrightarrow} s$ for terms t, s of type Self
- the boolean connectives $\neg F$, $F \wedge G$, $F \vee G$, $F \Longrightarrow G$ for formulae F and G
- the quantifications $\forall x : \tau \,.\, F$ and $\exists x : \tau \,.\, F$ for a formula F and an admissible type τ with Self not occuring in τ

Note that this definition of terms and formulae only depends on the notion of coalgebraic signature and does not need any models for a particular signature. So a logic is pure syntax.

The term for pattern matching probably needs some explanation. Remember that all elements of $\sigma \uplus \tau$ are either from σ or from τ. If we have a term $f(x) : \rho$ containing the free variable $x : \sigma$, a term $g(y) : \rho$ containing the free variable $y : \tau$ and a term s of type $\sigma \uplus \tau$ then we can form the term $\mathsf{CASES}\ s\ \mathsf{OF}\ \kappa_1 x : f(x),\ \kappa_2 y : g(y)\ \mathsf{ENDCASES}$. This term analyses s, and in case its value comes from σ, then it performs $f(x)$, otherwise $g(y)$. The pattern matching binds the variables x and y; therefore, its free variables are the free variables from $f(x)$ without x, plus the free variables from $g(y)$ without y, plus the free variables from s. The two cases $f(x)$ and $g(y)$ must always have the same type.

A crucial restriction of the logic is that for terms of type Self, one has to use bisimilarity instead of equality. Intuitively, this corresponds to the black box paradigm: It is impossible to check two black boxes for equality—all one can do is to compare the observations that they yield. Technically, this restriction accounts for the fact that coalgebra morphisms respect logical formulae, and therefore for the existence of final models; see below.

A logic also contains a derivation system that allows to prove entailments by pure syntactic manipulations. The most commonly known derivation rule is probably the modus ponens

$$\frac{F \qquad F \Longrightarrow G}{G}$$

I will not present a derivation system for the logic here, because in this paper, coalgebraic logic is only used for specification purposes and not for the derivation of properties. Instead, proofs will be carried out in the semantic universe—for this paper in PVS.

Example 3.8 A property of a decent memory is that after writing value d at address a, a read operation at the same address a gives the result d. This is expressed by the following property.

$$\forall a : \mathsf{Addr} \,.\, \forall d : \mathsf{Data} \,.\, \mathsf{read}(\mathsf{write}(x, a, d), a) \;=\; d$$

Another nice property of memories is that writes do not affect reads at different addresses.

$$\begin{aligned}\forall a_1 : \mathsf{Addr} \,.\, \forall a_2 : \mathsf{Addr} \,.\, \forall d : \mathsf{Data} \,.\, (\neg a_1 = a_2) \implies& \\ \mathsf{read}(\mathsf{write}(x, a_1, d), a_2) \;=\;& \mathsf{read}(x, a_2)\end{aligned}$$ ■

Definition 3.9 (Coalgebraic Specification) A coalgebraic specification consists of a coalgebraic signature Σ and a set $\mathcal{F}$ of formulae (the *assertions*) which contain at most one free variable $x : \mathsf{Self}$.

Again I chose here a definition that is close to practical work. Theoretically it is enough to allow for just one assertion. Several assertions can be combined with conjunction.

Assume we have a coalgebraic specification $(\Sigma, \mathcal{F})$ and a model M of the signature Σ together with an interpretation of the ground signature. Then it is possible to give semantics to all terms and formulae of Definition 3.7. The semantics of a term or a formula is denoted by brackets $[\![-]\!]$. If the formula (or the term) contains free variables $x_1 : \tau_1, \ldots, x_n : \tau_n$, then to get its denotation, it is necessary to choose values $v_1 \in [\![\tau_1]\!], \ldots, v_n \in [\![\tau_n]\!]$ from the sets $[\![\tau_i]\!]$ as interpretations of the free variables. The semantics of terms and formulae is then obtained by replacing the syntactical structure by their semantical counterparts. I use $[\![t]\!](v_1, \ldots, v_n)$ for the denotation of a term t where the free variables $x_1, \ldots, x_n$ of t are interpreted by $v_1, \ldots, v_n$.[3]

Definition 3.10 (Semantics) Fix an interpretation I of the ground signature and a model M of a coalgebraic signature Σ. Assume t is a Σ–term with free variables $x_1, \ldots, x_n$. Then its denotation $[\![t]\!]_M(v_1, \ldots, v_n)$ is defined as follows.

- If $t = x_i$ for a free variable x_i, then $[\![t]\!]_M(v_1, \ldots, v_n) = v_i$.
- If $t = k$ for a constant or a function symbol from the ground signature, then $[\![t]\!]_M(v_1, \ldots, v_n) = [\![k]\!]$.
- If $t = \pi_i(t')$ is a projection, then $[\![t]\!]_M(v_1, \ldots, v_n) = \pi_i([\![t']\!]_M(v_1, \ldots, v_n))$.

[3]The right tool to use here are *contexts*, see [Rot00]. But I prefer to leave technicalities aside and to concentrate on the essentials.

- If $t = (r, s)$ is a pair, then $[\![t]\!]_M(v_1, \dots, v_n) = ([\![r]\!]_M(v_1, \dots, v_n), [\![s]\!]_M(v_1, \dots, v_n))$.
- If $t = \kappa_i(t')$ is an injection, then $[\![t]\!]_M(v_1, \dots, v_n) = \kappa_i([\![t']\!]_M(v_1, \dots, v_n))$.
- If $t = \mathsf{CASES}\ s\ \mathsf{OF}\ \kappa_1 x : q,\ \ \kappa_2 y : r\ \mathsf{ENDCASES}$ is a pattern matching, then $[\![s]\!]_M(v_1, \dots, v_n)$ equals either $\kappa_1(u)$ for some $u \in [\![\sigma]\!]$ or $\kappa_2(v)$ for some $v \in [\![\tau]\!]$. In the first case $[\![t]\!]_M(v_1, \dots, v_n) = [\![q]\!]_M(u, v_1, \dots, v_n)$ and in the second case $[\![t]\!]_M(v_1, \dots, v_n) = [\![r]\!]_M(v, v_1, \dots, v_n)$.
- If $t = f(t')$ is a function application, then $[\![t]\!]_M(v_1, \dots, v_n) = [\![f]\!]([\![t']\!]_M(v_1, \dots, v_n))$. Here, $[\![f]\!]$ is the interpretation of f in I.
- If $t = c(s, t_1, \dots, t_n)$ is a coalgebra invocation, then $[\![t]\!]_M(v_1, \dots, v_n) = [\![c]\!]_M([\![s]\!]_M(v_1, \dots, v_n), [\![t_1]\!]_M(v_1, \dots, v_n), \dots, [\![t_n]\!]_M(v_1, \dots, v_n))$.

Assume a Σ–Formula F has free variables $x_1, \dots, x_n$. Then its denotation $[\![F]\!]_M(v_1, \dots, v_n)$ is either true of false.

- If F is an equality $t = s$, then $[\![F]\!]_M(v_1, \dots v_n)$ is true if $[\![t]\!]_M(v_1, \dots, v_n) = [\![s]\!]_M(v_1, \dots, v_n)$.
- If F is a bisimilarity $t \underline{\leftrightarrow} s$, then $[\![F]\!]_M(v_1, \dots v_n)$ is true if $([\![t]\!]_M(v_1, \dots, v_n))\ R\ ([\![s]\!]_M(v_1, \dots, v_n))$, for the bisimilarity relation R on M.
- If F is a boolean combination, say $F_1 \wedge F_2$, then $[\![F]\!]_M(v_1, \dots, v_n)$ is true if both $[\![F_1]\!]_M(v_1, \dots, v_n)$ and $[\![F_2]\!]_M(v_1, \dots, v_n)$ are true. The semantics for the other boolean operators is defined in the same way according to their truth tables.
- If F is a all–quantification, $F = \forall x : \tau_1 \,.\, F_1$, then F_1 contains one more free variable than F. The denotation of $[\![F]\!]_M(v_1, \dots, v_n)$ is true if $[\![F_1]\!]_M(v, v_1, \dots, v_n)$ is true for all $v \in [\![\tau]\!]$. Existentially quantified formulae are handled in a similar way.

It is now possible to define what constitutes a model of a coalgebraic specification.

Definition 3.11 (Model) Let $(\Sigma, \mathcal{F})$ be a coalgebraic specification with $\mathcal{F}$ the set of assertions. Assume a model M of the signature Σ with state space S. Then M is a model for the specification $(\Sigma, \mathcal{F})$ if for all formulae $F \in \mathcal{F}$ and all $s \in S$ the interpretation $[\![F]\!](s)$ is true.

For the memory example, it is easy to see that all models of its specification are isomorphic.[4] For an actual implementation of a memory (in hardware or software) one can check its correctness by proving that it is a model of the specification. Of course, the question remains whether the memory specification itself is correct, that is whether it specifies exactly the right properties. It is not possible to answer this question completely with formal methods.

[4] Strictly speaking, this holds only under the assumption, that Addr is interpreted by a finite set.

The final model of a specification is the final element among all models that fulfill the specification. Again it follows from general properties of the signatures and the logic of this paper that for any consistent specification[5] there exists a final model. One can obtain the final model of a specification $(\Sigma, \mathcal{F})$ by first constructing the final model Z for the signature Σ. One obtains the set Z' of those states of Z that fulfill all the formulae of $\mathcal{F}$. This Z' does not necessarily allow for a Σ–coalgebra structure. The final model of $(\Sigma, \mathcal{F})$ is therefore the largest subcoalgebra of Z with a state space smaller than Z'; see [Jac96] for details.

The final model of the memory specification is given by Example 3.5.

One can now go further and stipulate that every coalgebraic specification defines a new type whose denotation is the final model of that specification. The types obtained this way are sometimes called behavioral data types (in contrast to abstract data types). In complete analogy to algebraic specification (where initiality accounts for inductive function definitions and the induction principle) one can now add *coinductive function definitions* and the *coinduction principle* to the logic. Coinductive function definitions read this way: To define a function f with the newly created type Memory as its codomain, it is enough to give two equations

$$\begin{aligned} \mathsf{read}(f(\cdots), a) &= \cdots \\ \mathsf{write}(f(\cdots), a, d) &= f(\cdots) \end{aligned}$$

which specify the possible observations on the result of the function application. This can be used in a programming language with lazy evaluation to define functions on infinite behavior. An example is Charity [CF92]. Coinductive function definitions exploit the existence part of finality.

The coinduction principle can prove that two elements of a behavioral type are equal. Note that this is in general an infinite proof obligation. The coinduction principle states that bisimilarity (and thereby each bisimulation) on the final model is contained in equality. To prove that two elements of the final model are equal, it is therefore enough to construct a bisimulation that relates both elements. This way, the coinduction principle reduces the original infinite proof obligation into a finite one. The proof principle rests on the uniqueness part of finality.

This was necessarily a very short introduction into coalgebraic specification. I have simplified the matter as much as possible. The signatures and the logic of this chapter form a subset of the specification language CCSL. Currently, the CCSL compiler accepts coalgebraic signatures with parameterized constructors. Coalgebra declarations do not underlay the type restrictions from Definition 3.1; see [Tew00b] for a first discussion on how to model binary methods with coalgebras. Assertions must be formulated in higher-order logic, enriched with powerful modal operators; see [Rot00]. The behavior of the constructors can be restricted with a second set of formulae, the creation conditions.

[5] A specification is consistent, if there exists at least one model, which fulfills the assertions. This is the case, if there are no logical contradictions between the assertions.

In addition to coalgebraic specifications, a CCSL specification can also contain algebraic datatypes. But all these features do not play a role for the FIASCO case study.

4 Memory Management in FIASCO

This chapter introduces the subject of the case study: Virtual memory and its management in FIASCO. Virtual memory is a mechanism that enables the construction of multitasking and multiuser systems. From a program's point of view the memory is a continuous space of at least 4 gigabyte. From the operating system's point of view there are several, possibly simultaneously running programs, which compete for a scare amount of about 128 megabytes of main memory. Additionally, users expect that a crash of one program does not affect other programs or even the operating system.

The solution is to separate virtual addresses from physical addresses. Inside a program virtual addresses refer to instructions and data. Physical addresses select certain bytes from the main memory on the system bus.[1] A special hardware device, the *memory management unit*, maps virtual addresses into physical ones. The memory management unit is controlled by the operating system.

This chapter explains how virtual memory works on Intel architecture 32–bit architectures. The second section shows some source code from the FIASCO microkernel that manipulates the data for the memory management unit. These pieces of source code are the subject of this case study. They will be carefully examined in the next chapters.

4.1 Virtual Memory and Address Translation

Before I explain the data structure that drives the memory management unit, it is necessary to fix some terms for which there often exists only vague understanding outside the operating systems community. A fixed mapping from virtual addresses to physical addresses is called an *address space*. Typically the same virtual address denotes different physical addresses in different address spaces. But, if the address spaces share memory, it is also possible that different virtual addresses denote the same physical address. At processor level an address space is represented by a page directory and a set of page tables.

A *thread* is the smallest unit of activity that can be handled by the operating system. It consists of a stack and a register set, which can be loaded into the CPU. Every thread is assigned to an address space. The operating system loads the right address space into the memory management unit before it passes control to a user thread. A *process* consists of a set of threads that share one address space. From a users point of view, a process corressponds to a running program.

In a usual address space only a small portion of the virtual addresses are mapped to physical memory. If a thread tries to access some of the other virtual addresses, the

[1] For this case study I disregard memory mapped input/output.

memory management unit generates an exception called *page fault* and passes control to the operating system. The operating system decides then wether the page fault was actually a *segmentation violation*. A segmentation violation happens if the program tries to access a virtual address outside its current program and data space. In this case the operating system typically terminates the thread that caused the segmentation violation.

One of the other possible reasons for a page fault is that the operating system swapped parts of the processes data to disc to make space for other processes. In this case, the operating system loads some of the swapped out date back into the main memory and adjusts the address space. Then, the thread resumes execution at the instruction that caused the page fault in the first place.

If a process needs more memory, it has to request it from the operating system. Typically the operating system returns an interval of virtual addresses on such a request. Afterwards accesses to virtual addresses in this interval are valid and do not produce segmentation violations.

It is clear now that the management of address spaces is a major task of the operating system. It has not only to provide every process with some physical memory. It must also protect different processes from each other.

Now let us look at virtual memory on an Intel 32–bit architecture (IA32). There, virtual memory is divided into intervals of 2^{12} bytes, called *pages*. The same applies to the physical memory, but there the intervals are called *frames*. Both frames and pages are *size-aligned*, which means that every frame or page starts at a multiple of 2^{12}. The memory management unit manages address spaces in terms of full pages and frames. So it is not possible to map a virtual address to an arbitrary physical address, rather whole pages are mapped to whole frames. For large memory areas it is also possible to map superpages. A superpage is an interval of four megabytes, that is, it consists of 1024 pages. Superpages must also be size-aligned, they have to start at a multiple of 2^{22}.

On Intel systems addresses have a width of 32 bits. The most significant twenty bits of a virtual address designate a page and the least significant 12 bits select one byte in that page. The memory management unit takes the most significant 20 bits and computes from the current address space another 20 bits that select a frame. These are pasted together with the least significant 12 bits form the virtual address to yield the physical address. Figure 4.1 shows how this works.

A special register in the CPU, the *page-directory base register* (PDBR) selects the currently-enabled address space by pointing to a frame, called the *page directory*. A page directory is a hierarchical structure out of pointers to frames (so the all pointers are physical addresses, not virtual ones). Because frames are always size-aligned, the least significant twelve bits of such a pointer are always zero. The memory management unit uses these bits for status information. Valid pointers for instance must be marked with a one in the least significant bit. When the memory management unit translates a virtual address into a physical one it does the following steps.[2] The most significant ten bits of the virtual address are used as an offset to the page directory base register. This

[2] In practice the CPU uses a special cache to minimize the memory accesses that are caused by address translation.

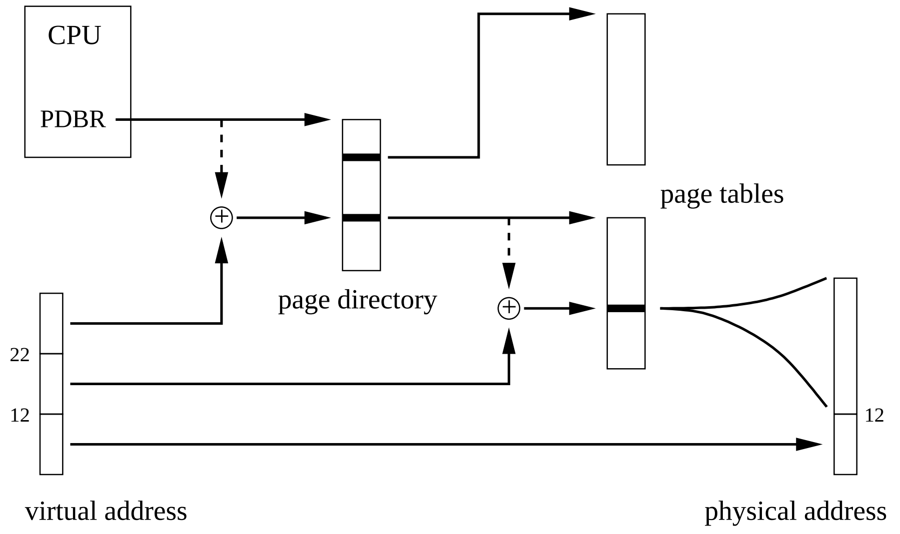


Figure 4.1: Address transformation

selects the *page directory entry*, a 4 bytes wide pointer to a frame. If this pointer is not valid (i.e., if bit number 0 is zero) then the memory management unit generates a page fault. If bit number seven (the superpage bit) of the page directory entry is zero, then it is a pointer to a *page table.* The next ten bits of the virtual address are used to select a *page table entry.* If this entry is valid as well, then its most significant twenty bits are pasted together with the least significant twelve bits from the virtual address.

If the superpage bit of the page directory entry is one, then the second indirection level is left out and the physical address results from pasting together the most significant ten bits of the page directory entry and the least significant 22 bits from the virtual address. Be careful with the terms *page directory entry* and *page table entry.* They are very similar, but it is important to keep them apart.

Occasionally FIASCO must emulate the functionality of the memory management unit to compute the physical address out of a virtual address in a given address space. As an illustration to the whole address translation process Listing 4.2 contains the corresponding C++ code.

4.2 Address Spaces in FIASCO

FIASCO must manipulate page directories and page tables to accommodate for the memory needs of the user processes that run under FIASCO. Therefore FIASCO contains a data type for representing address spaces. According to C++ design patterns this data type is implemented as a class. Its name is `space_t`. The source code of the class declaration is shown in Listing 4.3. For presentation purposes, I simplified a few preprocessor

```
#define PAGE_SIZE (1 << 12)
#define PAGE_MASK (PAGE_SIZE-1)
#define PDESHIFT 22      /* page descriptor shift */
#define PDEMASK  0x3ff   /* mask for page descriptor index */
#define PTESHIFT 12      /* page table shift */
#define PTEMASK  0x3ff   /* mask for page table index */
#define INTEL_PTE_VALID          0x00000001
#define INTEL_PTE_WRITE          0x00000002
#define INTEL_PTE_USER           0x00000004
#define INTEL_PTE_REF            0x00000020
#define INTEL_PTE_MOD            0x00000040
#define INTEL_PDE_VALID          0x00000001
#define INTEL_PDE_WRITE          0x00000002
#define INTEL_PDE_USER           0x00000004
#define INTEL_PDE_REF            0x00000020
#define INTEL_PDE_SUPERPAGE      0x00000080
#define INTEL_PDE_MOD            0x00000040     /* only for superpages */
#define SUPERPAGE_SHIFT          PDESHIFT
#define SUPERPAGE_SIZE           (1 << SUPERPAGE_SHIFT)
#define SUPERPAGE_MASK           (SUPERPAGE_SIZE - 1)
#define CPUF_4MB_PAGES           0x00000008     /* 4MB page support */

typedef unsigned int    size_t
typedef unsigned int    vm_offset_t;
typedef unsigned int    pt_entry_t;
typedef unsigned int    pd_entry_t;

inline vm_offset_t
space_t::virt_to_phys(vm_offset_t a) const
{
  pd_entry_t p = dir[(a >> PDESHIFT) & PDEMASK];
  if (! (p & INTEL_PDE_VALID)) return 0xffffffff;
  if (p & INTEL_PDE_SUPERPAGE) return (p & ~SUPERPAGE_MASK)
                                | (a & SUPERPAGE_MASK);
  pt_entry_t t = reinterpret_cast<pt_entry_t *>
    ((p & ~PAGE_MASK) + kmem::mem_phys)[(a >> PTESHIFT) & PTEMASK];

  return (t & INTEL_PTE_VALID) ? (t & ~PAGE_MASK) | (a & PAGE_MASK)
    : 0xffffffff;
}
```

Listing 4.2: Address transformation in C++ (mainly from file space.h)

directives. Apart from that, the code is taken directly from the FIASCO sources, version 0.6. The non–standard constants and types which are used in the class declaration, are contained in Listing 4.2.

The class space_t contains 13 methods and about 400 lines of code. With the methods it is possible to compute the physical address for an virtual address in the address space that the current object represents (method virt_to_phys). Further, one can test wether a certain interval of the virtual memory is mapped (method v_lookup), insert a new mapping (method v_insert), delete one (method v_delete), or initialize the memory management unit with this address space (method switchin_context). There are also methods for manipulating the input/output address space (in this presentation I disregard input/output address spaces completely), constructors, and destructors.

```
class space_t
{
public:
  enum status_t {
    Insert_ok = 0,
    Insert_warn_exists,       // mapping already existed
    Insert_warn_rw_upgrade, // mapping already existed and was r/w-upgraded
    Insert_err_nomem,         // couldn't alloc new page table
    Insert_err_exists          // a mapping already exists at the target addr
  };

  explicit space_t(unsigned new_number); // con: create new empty address space
  ~space_t();                                         // destructor

  void switchin_context();
  vm_offset_t virt_to_phys(vm_offset_t virtual_address);            // pgtable lookup
  status_t v_insert(vm_offset_t phys, vm_offset_t virt, vm_offset_t size, bool read_only = false);
  bool v_delete(vm_offset_t virt, vm_offset_t size, bool read_only = false);
  bool v_lookup(vm_offset_t virt, vm_offset_t *phys = 0, vm_offset_t *size = 0, bool *read_only = 0);
  bool io_insert(vm_offset_t phys, vm_offset_t size);
  bool io_delete(vm_offset_t phys, vm_offset_t size);

  void *operator new(size_t); // when creating task, makes sure pgdir is page-aligned
  void operator delete(void *block);

private:
  space_t();                     // default constructors are undefined
  space_t(const space_t&);

  // data layout (be careful here!): just a page directory
  pd_entry_t dir[1024];
};
```

Listing 4.3: space_t class declaration (file space.h)

The data layout of this class is very carefully designed such that it meets exactly the requirements of page directories for the Intel 32–bit architecture. This includes a class specific memory allocation routine, which ensures that newly created objects of the class space_t are size-aligned. Therefore it is possible to transform a pointer to an space_t object into its corresponding physical address and load it directly into the PDBR register of the CPU. This is in fact done in FIASCO every time the address space is changed (in the method switchin_context).

Unfortunately, there was only a small amount of time that I could spent on this case study. Therefore I had to restrict my attention to only a part of the functionality of the class space_t. I considered the method v_insert most interesting. Because of time restrictions I examined only the insertion of superpage mappings. Together with the method switchin_context it is possible to tackle the following challenge.

Challenge 4.1

1. *The methods v_insert and switchin_context always terminate correctly without producing a page fault.*

```
inline void space_t::switchin_context()
{
  register unsigned long my_pdir
    = reinterpret_cast<unsigned long>(this) – kmem::mem_phys;

  if (get_pdbr() != my_pdir)
    set_pdbr(my_pdir);
}
```

Listing 4.4: The method switchin_context (file space.h)

2. *Assume that the result of a method call of v_insert for an object o of class space_t indicates that the insertion of a new superpage mapping from a virtual address v to a physical address p was completed successfully. If the address space o is made current afterwards with method switchin_context, then the memory management unit indeed maps the virtual address v to the physical address p.*

Item (1) of the challenge is the essence from two important properties of the microkernel. First, the kernel should not contain any programming errors that yield invalid memory accesses. Second, the data of the microkernel itself is never swapped to disc. Item (2) requires a property of the whole system after successively calling the methods v_insert and switchin_context. Therefore, Item (2) relies partly on Item (1): control would never reach the code of the method switchin_context, if a page fault occurred during the method v_insert.

The rest of this paper centers around the formalization and the proof of this challenge. Therefore I prefer to give it a name and call it the *superpage challenge* hereafter. In Chapter 5 I develop a coalgebraic specification for the class space_t. It will contain a formalized version of the superpage challenge together with some properties that are necessary to state the challenge. In the remainder of this section I will explain the source code of the two methods v_insert and switchin_context.

Let us start with the shorter method switchin_context. Its source code is shown in Listing 4.4. It does not take any arguments (beside the implicit object of class space_t) and does not deliver any results. For its functionality some properties of the memory layout of FIASCO are important. First, the virtual addresses that FIASCO uses itself are mapped in every user address space (of course such that the user process cannot modify the kernel). Therefore it is possible to switch between user address spaces and continue with a subroutine in the kernel, without producing a page fault. Second, the virtual and physical addresses of the FIASCO kernel are fixed at compile time, therefore it is possible to transform a virtual address of the kernel into a physical one by simply adding a constant.

The method switchin_context uses this simpler form of address transformation and calls then routine set_pdbr to (re–) initialize the memory management unit. Because this initialization is very expensive, the method checks first, whether it is really needed. The routines set_pdbr and get_pdbr are implemented with inline assembly.

The method v_insert is more complicated (see Listings 4.5 and 4.6). As arguments

```
space_t::status_t
space_t::v_insert(vm_offset_t phys, vm_offset_t virt, vm_offset_t size,
                  bool read_only = false)
{
  // insert page into page table
  pd_entry_t *p = dir + ((virt >> PDESHIFT) & PDEMASK);
  pt_entry_t *e;

  if (! (*p & INTEL_PDE_VALID)) // we don't have a page table for this area
    {
      // check if we can insert a 4MB mapping
      if (size == SUPERPAGE_SIZE
          && (cpu.feature_flags & CPUF_4MB_PAGES)
          && superpage_aligned(virt) && superpage_aligned(phys))
        {                                                        // Section A
          *p = phys | INTEL_PDE_SUPERPAGE
            | INTEL_PDE_VALID | INTEL_PDE_REF
            | INTEL_PDE_MOD | INTEL_PDE_USER
            | (read_only ? 0 : INTEL_PDE_WRITE);

          return Insert_ok;
        }

      // can't map a superpage - alloc new page table              Section B
      pt_entry_t *new_pt = static_cast<pt_entry_t *>(kmem::page_alloc());
      if (! new_pt)
        return Insert_err_nomem;

      memset(new_pt, 0, PAGE_SIZE);

      *p = kmem::virt_to_phys(new_pt)
        | INTEL_PDE_VALID | INTEL_PDE_WRITE
        | INTEL_PDE_REF | INTEL_PDE_USER;
      e = new_pt + ((virt >> PTESHIFT) & PTEMASK);
    }
  else if (*p & INTEL_PDE_SUPERPAGE)
    {                                                            // Section C
      // already have mapped a superpage

      if (trunc_superpage(*p) + (virt & SUPERPAGE_MASK) != phys)
        return Insert_err_exists;

      // same mapping

      if ((*p & INTEL_PDE_WRITE) || read_only) // no upgrade?
        return Insert_warn_exists;

      // upgrade from read-only to read-write

      if (size == SUPERPAGE_SIZE)
        {
          // upgrade the whole superpage to read-write at once
          *p |= INTEL_PDE_WRITE;

          return Insert_warn_rw_upgrade;
        }
```

Listing 4.5: The method v_insert (file space.cc), Part I

```
      // we need to split the superpage mapping because
      // it's only partly upgraded to read-write
                                                        // Section D
      pt_entry_t *new_pt = split_pgtable(p);
      if (! new_pt)
        return Insert_err_nomem;

      *p = kmem::virt_to_phys(new_pt)
        | INTEL_PDE_VALID | INTEL_PDE_WRITE
        | INTEL_PDE_REF | INTEL_PDE_USER;

      e = new_pt + ((virt >> PTESHIFT) & PTEMASK);
    }
  else
    {                                                   // Section E
      e = static_cast<pt_entry_t *>(kmem::phys_to_virt(*p & ~PAGE_MASK))
        + ((virt >> PTESHIFT) & PTEMASK);
    }

  assert(size == PAGE_SIZE);

  if (*e & INTEL_PTE_VALID) // anything mapped already?
    {                                                   // Section F
      if (trunc_page(*e) != phys)
        return Insert_err_exists;

      if ((*e & INTEL_PTE_WRITE) || read_only) // no upgrade?
        return Insert_warn_exists;

      // upgrade from read-only to read-write
      *e |= INTEL_PTE_WRITE;
      return Insert_warn_rw_upgrade;
    }
  else                  // we don't have mapped anything
    {                                                   // Section G
      *e = phys | INTEL_PTE_MOD | INTEL_PTE_VALID | INTEL_PTE_REF
        | INTEL_PTE_USER
        | (read_only ? 0 : INTEL_PTE_WRITE);
    }

  return Insert_ok;
}
```

Listing 4.6: The method v_insert (file space.cc), Part II it implicitly takes an object o of class space_t, further a physical address phys, a virtual address virt, an interval length size and a flag. I disregard protection against unauthorized writing in this case study, so the flag read_only will not appear in the specification. The intent of this method is that (if possible) it changes the address space o such that virtual addresses in the interval $[\mathsf{virt}, \mathsf{virt} + \mathsf{size} - 1]$ are mapped to the interval $[\mathsf{phys}, \mathsf{phys} + \mathsf{size} - 1]$. There are only two possible values for size. It must be either 2^{12} for usual page mappings or 2^{22} for superpage mappings. The method v_insert delivers an element of the enumeration type status_t. Its value Insert_ok is used for successful termination, the other values signal certain error conditions.

Let us now go through the source line by line. The only value field of an object of class space_t is dir. It contains exactly one page directory, that is, 1024 pointers of 4 byte length. First, the address of the page-directory entry for the argument virt is stored in p.[3] Under the conditions, that first $*$p is an invalid page-directory entry, second the caller requested a superpage mapping, and third both addresses virt and phys are size-aligned for superpages, the address phys together with the necessary flags is stored at $*$p and the method returns. So until Section A the code test if it has to insert a superpage mapping into the address space that is represented by the current object. If the test succeeds, then the address space is adjusted in Section A.

If the page directory entry is invalid, but one of the other conditions is not met, then the code assumes that the caller wants to insert an ordinary page mapping. For this, first a valid page table is necessary. The code in Section B allocates a page from the available memory of FIASCO, initializes it appropriately, and updates the page directory. Finally the variable e is set to point to the page table entry corresponding to the virtual address virt.

The code in the Sections C and D deals with write permissions and is disregarded in this case study.

Section E is entered if p points to a valid page directory entry that is *not* an entry for a superpage. Then simply the variable e is loaded with the pointer to the right page table entry.

Now the first if statement of the method v_insert is completed. If the routine has not returned yet, then the variable e contains a pointer to a page table entry. Section F handles the case where this page table entry is valid. In Section G the current (invalid) page table entry is overwritten with a pointer to the frame of phys.

Note that the source code of v_insert depends on the methods page_alloc, virt_to_phys, phys_to_virt of the class kmem and on the function memset. When I verify the code of v_insert I will have to axiomatize these functionality.

Let me draw your attention to a flaw in the code. Assume, we call v_insert with $\mathsf{size} = 2^{22}$ and a virtual address with the following properties. There is a valid page directory entry for it, but this page directory entry does not describe a superpage mapping. Then, the routine jumps directly to Section E and the assert statement on line 20, Listing 4.6,

[3] The $*$ is the dereferencing operator of C++. Here p is a local variable, it contains pointers to elements of pd_entry_t (i.e., four byte long pointers). The variable p is initialized with its declaration (line 6). In the following source, $*$p denotes the element, to which p points. Assigning something to $*$p will change the memory location to which p points (and not p itself).

fails.

A similar situation happens if there is no valid page directory entry, but one of the addresses virt and phys is not size-aligned for superpages. Then we jump to Section B and after that directly to the assert statement.

This means, that the method v_insert is not robust against certain misuses. The assert statement will fail if the caller uses wrong arguments. This error was found during the verification in the proof of Lemma insert_up_ok, see Section 6.2 on page 51ff. The Lemma insert_up_ok states that the method v_insert, if called in a consistent system state, returns always normally and does not produce any page faults.

5 The Specification

In this chapter I develop the specification for the class `space_t`. First, there are many design questions to answer, for instance, how can one formalize the superpage challenge 4.1? What is an appropriate formal model? Then, similar to programming, decisions must be taken to cut the whole system into smaller pieces. Often these decisions are to some extend arbitrary, and what is best depends on the personal experience and the preferences of the person who carries out the verification. So in this paper I do not present how the memory management of FIASCO should be formalized. Rather I tell, how I formalized it.

If we look at the source code in the previous chapter, we see that apart from the bit manipulations, almost everything can be expressed in terms of read and write operations to a virtual memory. The superpage challenge is a property about the address translation of a given address space. Therefore I decided, to built first a formalization of virtual memory. With this, I can express the properties of the methods of `space_t`.

For the bit manipulations, I use the PVS bitvectors library [BSGM97]. Similar to a software library it contains the data type of bitvectors and useful functions working over bitvectors. In addition there are many theorems together with proofs for general properties of bitvectors. In terms of the formal definitions of coalgebraic specification (Section 3, pages 15ff) the bitvectors library is part of the ground signature.

There are another two important design decisions for the memory specification. First, for a real memory, it is possible to write and read with different data length, for instance bytes (8 bits) and words (32 bits). This means one can read and modify the same data under different addresses. Introducing such a feature into the specification will make it *much more* difficult to work with it. Therefore, I decided to model only a memory of words. Addresses range from 0 to $2^{30} - 1$ and the data that can be read in the memory corresponds to 4 bytes.

The second problem is the view of addresses and data as natural numbers *and* bitvectors. On one hand, the addresses should have an ordering and addition and subtraction as basic operations. On the other hand, one would like to view addresses and the values which are stored in the memory as bitvectors and cut and paste them together. The FIASCO sources solve this problem by relying on the rather unnatural view of C++ that natural numbers *are* bitvectors.

In a theorem prover with a much more rigorous type system, both types are different. The bitvectors library contains bijective conversion functions that map bitvectors of a certain length to an appropriate subset of the natural numbers.

One possibility is to design a memory following Example 3.3 (page 16) and take the natural numbers $\mathbb{N}$ as addresses and data. But if a bitvector manipulation is necessary, one has to use the conversion function from the natural numbers to bitvectors two times, first to transform the number into a bitvector and then back to a number again.

After a few experiments, I tried to minimize the use of the conversion functions and decided to use natural numbers less than 2^{30} as addresses and bitvectors of length 32 as values. The price to pay here is that if a value that was read somewhere in the memory is used directly as an address, then it first has to be converted to the right number (by cutting off the least significant two bits). But this does never happen in the whole case study, there is always some bitvector manipulation involved.

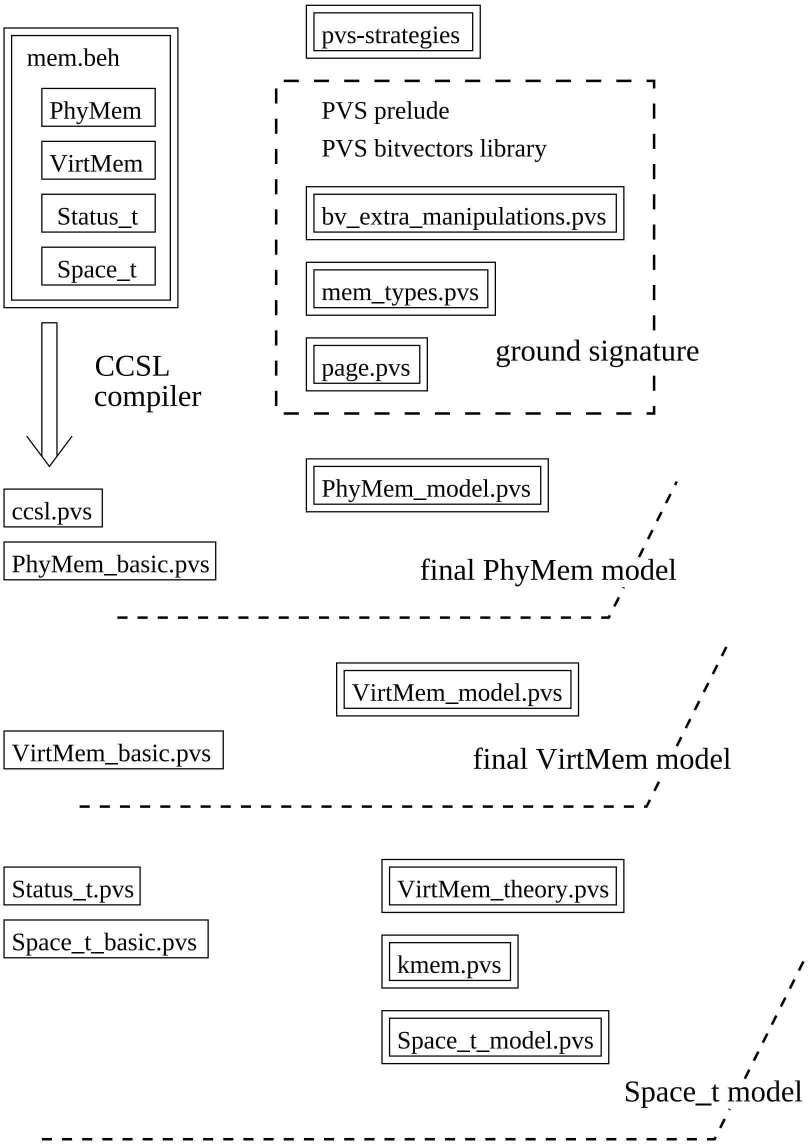
Looking at a virtual memory, one can separate its functionality into an underlying physical memory and the address translation. Therefore I first formalize physical memory and add the features for virtual addresses in a second step. Then I use the virtual memory in the specification for the class space_t. The source code presented in this chapter is the original CCSL specification from the file mem.beh. For the presentation in this paper I only changed some comments and the indentation.

Figure 5.1 shows the files that belong to the case study. The file mem.beh contains the specification of physical memory (see Section 5.1), virtual memory (Section 5.2), and of the class space_t (Section 5.3). The files PhyMem_model.pvs and VirtMem_model.pvs are not discussed in detail; their role is explained together with the specification of physical memory and virtual memory, respectively. There is no space in this paper to explain the contents of the automatically generated files (the ones in single boxes in Figure 5.1), see [HHJT98] and [Tew00a] for details. The file pvs-strategies defines new proof commands for PVS. It is read into PVS at the begin of each session. The PVS prelude contains all the definitions and lemmas that are always available in PVS. The PVS bitvectors library contains almost 40 files. They define the type of bitvectors for PVS and contain many general definitions and lemmas. All the other files in Figure 5.1 contribute directly to the proof of the superpage challenge. I explain their role in Chapter 6.

5.1 Physical Memory

In formalizing some real device, it is a rather bad idea to describe its properties as accurate as possible. Rather one should abstract as most as possible from irrelevant details. What is irrelevant depends on the purpose. For this case study, read and write latencies are certainly irrelevant. I also decide to disregard caches. But in contrast to the example developed in Section 3 the physical memory of this chapter is *finite*. This applies to the number of memory cells as well as to the involved types. Before presenting the CCSL specification, it is necessary to extend the ground signature with the types for addresses and data, as discussed above. This is done in theory MemTypes (file mem_types.pvs). It contains the type declarations bvec32 (for bitvectors of length 32) and Addr (for natural numbers less than 2^{30}). The CCSL specification is in Listing 5.2, it is almost identical to the specification developed in Example 3.8 on page 20.

Let us go through the specification in Listing 5.2 line by line. The first four lines in Listing 5.2 declare type symbols, which are defined in PVS for the CCSL compiler. Comments start in CCSL with a percent sign % and reach till the end of the line. The keywords



This figure gives an overview over the files that belong to the case study. Files within double frames are written by myself. Files in single frames are automatically generated. I discuss the single files and their contents in the corresponding chapters/sections. (See page 35 for more information.)

Figure 5.1: Files in the FIASCO case study

```
1  TYPE bool : BASETYPE    % true and false
2  TYPE nat  : BASETYPE    % the natural numbers
3  TYPE Addr : BASETYPE    % <= 2 wwww 30
4  TYPE bvec32 : BASETYPE  % bitvectors of length 32
5
6  % physical memory with max words capacity
7
8  BEGIN PhyMem[ max : Addr ] : CLASSSPEC
9    IMPORTING "MemTypes"
10   METHOD
11     read : [Self, Addr] -> bvec32;
12     write: [Self,Addr,bvec32] -> Self;
13
14   ASSERTION
15         SELFVAR mem : Self
16         Var a1, a2 : Addr; d : bvec32
17     phymem1 : PVS   a1 < max AND a2 < max AND
18         a1 = a2 IMPLIES read(write(mem,a1,d),a2) = d
19     ENDPVS
20
21     phymem2 : PVS   a2 < max AND
22         NOT a1 = a2 IMPLIES read(write(mem,a1,d),a2) = read(mem,a2)
23     ENDPVS
24
25   ASSERTION
26         SELFVAR mem : Self
27         VAR a : Addr
28     phymem3 : PVS
29         a >= max IMPLIES read(mem,a) = bvec1
30     ENDPVS
31 END PhyMem
```

Listing 5.2: Physical Memory in CCSL (file mem.beh)

BEGIN ... **CLASSSPEC** start a new class specification. Its name is PhyMem and it is parameterized by the amount max of available main memory. Read and write operations at addresses greater than max will go into nowhere. In practice max is always a multiple of the page size, but this condition is not necessary in this case study. From a logical point of view max is all–quantified on the meta level. So all results, which I obtain in this case study, apply to all possible values of max.

As explained at the end of Chapter 2 the CCSL compiler translates the specification into a set of logical theories for PVS. The **IMPORTING** statement tells the CCSL compiler to make the generated theories descendants of theory MemTypes, such that the declarations in MemTypes can be used in the specification of PhyMem.

A CCSL specification consists of a number of sections for coalgebra declarations, assertions, constructors, and creation conditions. Their order is not fixed and not every section must be present in a specification. Methods of classes will be modeled with coalgebras in CCSL, therefore the section for coalgebra declarations starts with the keyword **METHODS** on line 10. The keyword **Self** denotes the state space of the class, the arrow −> the function space and angle brackets [$\cdots$] are used for the Cartesian product. So the method write is declared to have type $\mathsf{Self} \times \mathsf{Addr} \times \mathsf{bvec32} \longrightarrow \mathsf{Self}$.

The specification contains three assertions, the first two with names phymem1 and phymem2 are almost identical to the formulae in Example 3.8. The only difference is that they require the involved addresses to be less than the boundary max. The keyword **SELFVAR** declares the free variable of type Self for the assertion. For convenience, the CCSL compiler allows also the declaration of other variables, they are implicitly all–quantified. The assertion phymem3 specifies the behavior for addresses for which no hardware exists. The constant bvec1 is the bitvector with all bits one. This last assertion is not necessary. The complete case study could be done without it. But it helps us, to convince ourselves that the specification itself is correct.

The whole case study relies on this specification of physical memory. An error in this specification, would make all subsequent results meaningless, even if their proofs are formally correct. If the specification PhyMem is inconsistent, then its is possible, to obtain false theorems from it. So it is a good idea to convince ourselves, that this specification of physical memory is correct.

Note that there is no formal definition of the term *correct memory specification*. This is impossible because there is no absolute specification of physical memory that we could use to check the specification PhyMem. I can only try to convince other people about the correctness of my specification and draw conclusions from my success.

There are two possible problems. Either the specification is too restrictive. This means that the memory I have in mind is not a model of the specification. Or it is too loose, allowing also structures that are not memories, as models. I can check for the first problem by proving that my favorite memory is a model of the specification. If I succeed then I also know that the specification is consistent, that is, that there are no contradictions between the assertions.

For the second problem, it is a good idea to construct the final model of the specification. Because the final model realizes all possible behaviors (see discussion on page 13 and 22), we can examine the states of the final model, and check for unexpected behavior.

For the PhyMem specification both checks fall together. I will not show the PVS sources and the proofs here, but the file PhyMem_model.pvs contains a model very similar to Example 3.5 on page 17. It proves that the set of functions that map natural numbers less than max to bitvectors of length 32 is the final model for PhyMem. So I can conclude that the specification PhyMem is consistent and that it does not have any strange models. The third assertion phymem3 helps here to keep the final model simple. Without it, the final model would be similar to Example 3.6.

5.2 Virtual Memory

In this section I extend the specification PhyMem such that it models the virtual memory of the Intel 32–bit architecture. CCSL follows the object oriented paradigm, so it is possible to construct a new specification by inheriting from existing ones. Inheritance here means that both the method declarations and the assertions are inherited. The resulting specification of virtual memory is in Listing 5.3. It contains a number of new types and functions, I will explain them in the following.

```
BEGIN VirtMem[ max : Addr ] : CLASSSPEC
  IMPORTING "Page"
  INHERIT FROM PhyMem[max] RENAMING
                read AS phy_read AND write AS phy_write
  ATTRIBUTE
    pdbr : Self -> bvec32;

  METHOD
    % set_pdbr : [Self,bvec32] -> Self;
    virt_to_phy:[Self,Addr] -> Lift[Addr];
    read : [Self, Addr] -> Lift[bvec32];
    write: [Self,Addr,bvec32] -> Lift[Self];

  ASSERTION   SELFVAR mem : Self
  v_p : PVS FORALL( v : Addr) : virt_to_phy(mem,v) =
    LET pde = phy_read(mem, pde_addr(pdbr(mem),v))
    in  IF NOT VALID_PDE(pde) THEN bot
        ELSIF SUPER_PAGE(pde) THEN
                up(super_addr(pde,v))
        ELSE LET pte = phy_read(mem, pte_addr(pde,v))
             IN  IF VALID_PTE(pte) THEN
                      up(norm_addr(pte,v))
                 ELSE bot
                 ENDIF
        ENDIF
  ENDPVS

  virt_read : PVS FORALL(v:Addr) :
          up?(virt_to_phy(mem,v)) IMPLIES
                read(mem,v) = up(phy_read(mem,down(virt_to_phy(mem,v))))
  ENDPVS

  virt_read_fail : PVS FORALL(v:Addr) :
          bot?(virt_to_phy(mem,v)) IMPLIES
                read(mem,v) = bot
  ENDPVS

  virt_write : PVS FORALL(v : Addr, d : bvec32) :
          up?(virt_to_phy(mem,v)) IMPLIES
             write(mem,v,d) = up(phy_write(mem,down(virt_to_phy(mem,v)),d))
  ENDPVS

  virt_write_fail : PVS FORALL(v : Addr, d : bvec32) :
          bot?(virt_to_phy(mem,v)) IMPLIES
                write(mem,v,d) = bot                ENDPVS

  pdbr_phy_write : PVS FORALL(v:Addr, d : bvec32) :
        pdbr(phy_write(mem,v,d)) = pdbr(mem)           ENDPVS

  phy_read_set_pdbr : PVS FORALL(v:Addr, p : bvec32) :
        phy_read(set_pdbr(mem,p),v) = phy_read(mem,v)   ENDPVS

%   set_pdbr_change : PVS Forall(a:Addr) :
%              pdbr(set_pdbr(mem,a)) = a                 ENDPVS
%
END VirtMem
```

Listing 5.3: Specification of virtual memory (file mem.beh)

The specification of virtual memory is also parameterized on the amount max of main memory that is available in the system. Because the specification VirtMem will also declare the methods read and write, the inheritance clause on line 3 renames the original methods from PhyMem into phy_read and phy_write. The CCSL compiler makes sure that the inherited assertions from PhyMem apply now to the original, renamed methods.

Classes often contain a number of instance variables. They keep the state of the object and can be read and set in the methods of the class. The CCSL compiler supports this practice with an attribute section. An instance variable of type τ can be declared in the attribute section as a coalgebra $\mathsf{Self} \longrightarrow \tau$. Then the CCSL compiler generates an update method of type $\mathsf{Self} \times \tau \longrightarrow \mathsf{Self}$. It also generates axioms that restrict the behavior of the attribute and the update method. These axioms require first, that reading the attribute after changing it returns the new value. Second, in case there are several attributes, reading attribute a_1 after changing attribute a_2 has to return the unchanged value of a_1.

For the VirtMem specification I use the attribute section of CCSL to declare the page directory base register (PDBR) as an attribute in line 6. From this attribute, an additional method set_pdbr and an assertion set_pdbr_change are generated. I inserted them in CCSL syntax as comments into Listing 5.3. In principle, the page directory base register should contain an address. But for the optimization reasons I described above, I use the corresponding bitvector instead.

Further, the specification contains three methods, virt_to_phy, read, and write. The latter two are used for reading and writing *at virtual addresses*. The method virt_to_phy translates a virtual address into a physical one. As explained in Chapter 4 this is a partial operation[1], which might fail because of a page fault. In the specification this is modeled by the result type Lift[Addr].

The typeconstructor Lift is defined in the CCSL prelude[2], so it is available in the ground signature. It adds a new element to its argument, so $\mathsf{Lift[Addr]} = \mathsf{Addr} \uplus \mathbf{1}$. The function $\mathsf{up} : \mathsf{Addr} \longrightarrow \mathsf{Lift[Addr]}$ and the constant $\mathsf{bot} : \mathsf{Lift[Addr]}$ accompany the type Lift[Addr]. Every element of Lift[Addr] belongs either to the left component of $\mathsf{Addr} \uplus \mathbf{1}$ and equals $\mathsf{up}(a)$ for some $a \in \mathsf{Addr}$ and thus denotes successful termination with result a. Or it belongs to the right component, equals bot, and denotes failure. The function down can be applied to elements of Lift[Addr], which are of form $\mathsf{up}(a)$[3] to extract the a. There are also two predicates up? and bot?. They can be applied to elements of Lift[Addr] and are true exactly if the element is of the form $\mathsf{up}(a)$ or bot, respectively.

[1]Recall, that a partial function is a function that is not defined for some values of its domain. Thus is does not deliver results for those values (similar to a program that aborts on certain inputs). A partial function from A to B can be considered as a total function $A \longrightarrow B \uplus \mathbf{1}$, where all values for which the original partial function is not defined are mapped to the only element $\emptyset$ of $\mathbf{1}$.

[2]The CCSL prelude is a piece of CCSL code that is automatically prepended to every CCSL source file. It contains definitions for standard datatypes like Lift and list.

[3]PVS uses a special dependently typed logic to type partial functions like down. This way the type system of PVS becomes undecidable. As a solution, PVS generates special proof obligations (type correctness conditions or TCC's). So for every use of down, the user must help the typechecker and prove that the argument of down is of form $\mathsf{up}(a)$.

Recall that partial functions into Addr correspond to (total) functions into Lift[Addr].

Of course, attempts to read or write at an virtual address might produce a page fault, therefore read and write also use the typeconstructor Lift.

The assertion v_p in line 15 in Figure 5.3 is the most important one in this specification. It defines the result of virt_to_phy in terms of physical read accesses to the memory. The assertion simulates exactly the algorithm that is described in Section 4.1. To make the assertion readable, I defined some functions for bit manipulations in theory Page (file page.pvs).

The function pde_addr takes the contents of a PDBR register and a virtual address and computes the address of the page directory entry for this virtual address. The functions VALID_PDE, VALID_PTE, and SUPER_PAGE take a bitvector and test for the corresponding status bit in it. If the page directory entry in pde is not valid, then the assertion requires that the method virt_to_phy fails with result bot. If pde describes a superpage entry, then the function super_addr merges the page directory entry with the virtual address together. The result is wrapped in an up, signaling successful termination. If pde is not a superpage entry, then the second indirection is done with the functions pte_addr (computing the address of the page table entry) and norm_addr (merging a page table entry with the lower bits of the virtual address).

The other assertions restrict the behavior of the other methods. They state the obvious requirements for memories, for instance the assertion virt_read says that if it is possible to transform the virtual address v into a physical address, then reading at v produces no page fault and delivers the same result as reading at the corresponding physical address.

The last two assertions regulate the relation of the PDBR attribute and the memory itself. They state that changing one of them does not effect the other.

Again it is a good idea to build a model for the specification VirtMem. The PVS sources contain the file VirtMem_Model.pvs. It proves that pairing the final model for PhyMem with a bitvector of length 32 (the PDBR) forms the final model for VirtMem.

5.3 The Specification of Space_t

It is now time to come back to the original aim, a formalized version of the superpage challenge on page 28. The full specification is in Listing 5.5 on page 44, but it is quite complicated. Before I explain it, I have to discuss a few points.

First, there is a major difference between the specifications of the memories in the previous sections and the specification Space_t. The specification PhyMem and VirtMem are complete specifications on their abstraction level. That is, all models of these specifications are correct memories. The specification Space_t is deliberately too loose. There are models of this specification that cannot be considered as correct implementations. But the specification will contain a formalized version of the superpage challenge, so the original aim can be achieved by proving that the FIASCO sources give raise to a model of Space_t.

Second, the FIASCO sources contain an enumeration type status_t, which is used as

```
BEGIN Status_t : ADT
  CONSTRUCTOR
    Insert_ok : CARRIER;
    Insert_warn_exists : CARRIER;
    Insert_warn_rw_upgrade : CARRIER;
    Insert_err_nomem : CARRIER;
    Insert_err_exists : CARRIER;
END Status_t
```

Listing 5.4: The enumeration type status_t (file mem.beh)

result of the method v_insert. In this case study I formalize it as a non–recursive abstract datatype, see Listing 5.4. Besides class specifications CCSL allows for abstract data type specifications. Similar to class specifications, abstract datatype specifications define a new type, its semantics is given by the initial model of the data type specification. In Listing 5.4 the keyword **CARRIER** is used as a placeholder for the abstract datatype being defined. The initial semantics ensures that the type Status_t has exactly five elements. I will use the names of the constructors in Listing 5.4 to denote these elements.

The third point is about modification of state. In a theorem prover it is not possible to modify objects *in place* as it is usually done in programming.[4] Instead one has to use transition functions that deliver the modified object as result. An example for this technique is the method write in both memory specifications. In the Space_t specification the use of transition functions becomes really striking, because the assertions state properties about a sequence of three method applications. For every following method application the successor states must be extracted and bound to a new variable. The resulting code looks difficult and leads to complicated proof obligations during the verification. This shows one of the major deficiencies of the current CCSL version. Although it is possible to verify imperative programs with it, it gives only minimal support. Ideally one would like to write assertions as a sequence of statements and let the CCSL compiler construct transition functions from it and connect them appropriately. I will return to this point in the conclusions.

The fourth point concerns the virtual memory. As described in Chapter 3 I consider the specification VirtMem as the specification of a new type. Because it is a class specification, its semantics is given by the final model of the specification. Therefore I can use the type symbol VirtMem in type expressions. This approach is directly supported by the CCSL compiler.

The fifth point is about the relation of the virtual memory and the specification Space_t. In reality objects of the class space_t *live in* a virtual memory. If such an object is modified, then also a special portion of the virtual memory is modified and vice versa. If we develop coalgebraic specifications for both, then they are seen as independent black boxes. The specification Space_t contains special structure to connect Space_t and the virtual memory, which I explain in the following.

[4] Assume $x = y$ and a function f, which modifies x *in place*. Then after applying f we can prove $x \neq y$, so we have both $x = y$ and $x \neq y$, a contradiction.

When I developed the specification, I decided to leave the connection between the virtual memory and the Space_t specification as weak as possible. This way I can work out which properties are essential for the sources of FIASCO. There are the following two points:

- It is easy to imagine an object m of class space_t (i.e., an address space) which does not live in a given virtual memory v. That means, every page of the memory v, when considered as a page directory, realizes an address space different to m. It does not make any sense to check if the superpage challenge holds for the pair of m and v. Therefore the specification Space_t contains an additional method consistent? of type Self $\times$ VirtMem$\longrightarrow$bool, whose behavior is left completely open in the specification. The intention is, that this method delivers true for a pair of m and v if m lives in v. All assertions of Space_t have an assumption of the form consistent?(m, v).
- As I said above, changes in the address space are reflected in the virtual memory. Therefore all methods in the specification take an additional argument of VirtMem. It is seen as the state of virtual memory in which the method is called. There is also an additional result, the state of virtual memory, when the computation of the method is finished.

Now it is possible to discuss the specification in Listing 5.5. It is also parameterized by the amount max of available main memory. Apart from the method consistent? (see above) only the two methods v_insert and switchin_context are necessary for the formalization of the superpage challenge. Both methods change the state of the current object and for both it is possible, that a page fault occurs in the middle of the method. This explains, why both take two arguments of Self and VirtMem and produce a result of type Lift[[Self, VirtMem]].[5] The method v_insert takes additionally a physical address, a virtual address, and a size. It also produces an additional result of type Status_t.

Because I only consider insertions of superpage mappings, all calls to v_insert in the assertions have a fixed size argument of SUPER_PAGE_SIZE.[6] The first assertion insert_up states that in case x lives in the virtual memory v, then the method v_insert terminates correctly for sensible arguments phys. The assertion switchin_context_up states the same for the method switchin_context.

The other assertions state properties of the successor state of the two methods. It is therefore necessary to use the function down to extract the successor states from the results of the methods. However, this makes only sense if the methods terminate correctly (without page fault). So we have a situation where an assertion can only be interpreted if a different assertion is fulfilled. Such dependent assertions are not (yet) supported by the CCSL compiler. But there is the following workaround.

Assertions depend on a coalgebra (for interpreting the methods of the class) and contain one free variable of type Self (the one declared with the keyword **SELFVAR**).

[5] This is the Lift typeconstructor applied to [Self, VirtMem]—the Cartesian product Self $\times$ VirtMem.

[6] SUPER_PAGE_SIZE is defined as 2^{20}, which corresponds to 2^{22} Bytes, because the memory counts in words of 32 bits lenght.

```
BEGIN Space_t[ max : Addr ] : CLASSSPEC
 METHOD
  consistent? : [Self,VirtMem[max]] -> bool;
  v_insert : [Self,VirtMem[max],Addr,Addr,nat] -> Lift[[Self,VirtMem[max],Status_t]];
  switchin_context : [Self,VirtMem[max]] -> Lift[[Self,VirtMem[max]]]

 ASSERTION              SELFVAR x : Self
  insert_up : PVS FORALL( v : FVirtMem, phys, virt : Addr ) :
      phys + SUPERPAGE_SIZE + 1 <= max AND
      consistent?(x,v) IMPLIES
          up?(v_insert(x,v,phys,virt,SUPERPAGE_SIZE))
  ENDPVS

  switchin_context_up : PVS  FORALL( v : FVirtMem) :
      consistent?(x,v)    IMPLIES    up?(switchin_context(x,v))
  ENDPVS

  insert_arg : PVS FORALL( v : FVirtMem, phys, virt : Addr ) :
      % collect some other assertions for TCCS:
          insert_up?(c)(x) AND
      % now the real assertion
      phys + SUPERPAGE_SIZE + 1 <= max AND
      consistent?(x,v) AND
      PROJ_3(down(v_insert(x,v,phys,virt,SUPERPAGE_SIZE))) = Insert_ok IMPLIES
              virt + SUPERPAGE_SIZE <= exp2(30)
  ENDPVS

  insert_consistent : PVS FORALL( v : FVirtMem, phys, virt : Addr ) :
      % collect some other assertions for TCCS:
          insert_up?(c)(x) AND
      % now the real assertion
      phys + SUPERPAGE_SIZE + 1 <= max AND
      consistent?(x,v) IMPLIES
      let v1 = Proj_2(down(v_insert(x,v, phys, virt, SUPERPAGE_SIZE))),
          x1 = Proj_1(down(v_insert(x,v, phys, virt, SUPERPAGE_SIZE)))
      IN
          consistent?(x1,v1)
  ENDPVS

  insert_ok : PVS FORALL( v : FVirtMem, phys, virt : Addr ) :
      % collect some other assertions for TCCS:
          insert_up?(c)(x) AND
          (Forall( y : Self ) : switchin_context_up?(c)(y)) AND
          insert_arg?(c)(x) AND
          insert_consistent?(c)(x) AND
      % now the real assertion
      phys + SUPERPAGE_SIZE + 1 <= max AND
      consistent?(x,v) AND
      Proj_3(down(v_insert(x,v,phys,virt, SUPERPAGE_SIZE))) = Insert_ok
    IMPLIES
      LET v1 = Proj_2(down(v_insert(x,v, phys, virt, SUPERPAGE_SIZE))),
          x1 = Proj_1(down(v_insert(x,v, phys, virt, SUPERPAGE_SIZE))),
          v2 = Proj_2(down(switchin_context(x1,v1)))
      IN  FORALL(o : below(SUPERPAGE_SIZE)) :
              virt_to_phy(v2,virt + o) = up(phys + o)
  ENDPVS
END Space_t
```

Listing 5.5: Specification of space_t (file mem.beh)

The CCSL compiler turns every assertion into a predicate on the state space of the class. The name of the predicate is derived from the name of the assertion by appending a question mark. So in the assertion insert_arg the first condition insert_up?$(c)(x)$ is true, if for the coalgebra c that is used to interpret the methods, and for x of type Self the assertion insert_up is true. In the rest of the assertion I can now safely apply down to the result of v_insert and use the third projection to access the status flag.[7] This assertion states that v_insert only terminates with Insert_ok if the argument virt fulfills some basic consistency conditions.

The assertion insert_consistent also depends on the assertion insert_up. It states that if v_insert is called with a consistent pair (x, v) then also the successor state of Space_t, and the successor state of the virtual memory form a consistent pair.

The last assertion insert_ok is now the formalization of the superpage challenge. It depends on all three previous assertions. The assertion states, that if the method v_insert terminates with Insert_ok for a consistent pair (x, v), then for the virtual memory v_2, which is obtained by applying switchin_context to the successor states of v_insert, the following holds: For all offsets[8] o less than 2^{20} the virtual memory v_2 maps the virtual address virt $+\ o$ to the physical address phys $+\ o$, where virt and phys are the original arguments of the method v_insert. To put it a bit simpler, the assertion insert_ok states that v_insert correctly inserts superpage mappings.

[7] The additional assumption of insert_up?$(c)(x)$ is not a formality. Every time the assertion insert_up is used in PVS, the theorem prover requires the user to prove, that the argument of down (i.e., the result of method v_insert) is of the form up$(-)$. So without the additional assumption, it would not be possible to complete the proofs.

[8] The typeconstructor below(n) defines the type of natural numbers less than n.

6 The Verification

In this chapter I translate the FIASCO sources of Section 4.2 into a model of the specification Space_t from the previous section. According to Definition 3.11 (page 21) this translation comprises two tasks. First I have to define a model of the signature to interpret the methods of Space_t. Then I have to prove that the assertions hold for this model.

Technically I proceed as follows.[1] First I run the CCSL compiler on the whole specification. For every class specification the compiler outputs a _basic.pvs file. It formalizes the signature, the definition of bisimulation, the predicates corresponding to the assertions, and many general lemmas in the higher-order logic of PVS. The theories that are generated are described in [HHJT98]. The compiler also generates a file _basic.prf containing some proofs for the lemmas. For the specification VirtMem the compiler generates a new type FVirtMem and a (uninterpreted) function final_VirtMem together with axioms that state that the coalgebra final_VirtMem on state space FVirtMem is is the final model of the specification VirtMem. Here I only use that FVirtMem is a model of VirtMem. I do not exploit its finality.

Internally the CCSL compiler handles only one coalgebra per class specification. For the specification Space_t the CCSL compiler generates also a definition of what consitutes a model of the specification Space_t. This definition of model assumes that all three methods consistent?, v_insert, and switchin_context of Space_t are combined into one coalgebra. In the following section I will define a Space_t coalgebra (see Listings 6.1, 6.2 and 6.3). Such a combined coalgebra can be considered as the implementation of a class. The compiler also generates utility functions that allow to access single methods. If c is a Space_t coalgebra, then switchin_context$(c)(x, v)$ extracts the method switchin_context from c and applies it to the arguments x and v.

I should note that the three specifications and the final models for PhyMem and for VirtMem were relatively easy and straightforward to develop. Altogether it took me only about three weeks, counting from the point where I started to learn about virtual memory and FIASCO. The model and its verification were the hard part of the case study. It cost me additional three months, to complete the case study as it is described in this paper.

[1]See Figure 5.1 on page 36 for an overview over the relevant files.

```
Space_tModel[(IMPORTING MemTypes) max : Addr] : THEORY
BEGIN
 Space_tState : TYPE = Addr
 IMPORTING Space_t[Space_tState , max]
 IMPORTING Kmem[FVirtMem, max]

 space_consistent? : [Space_tState, FVirtMem -> bool] =
   LAMBDA(x : Space_tState, v: FVirtMem ) :
       % current mapping lays in physical memory
       phy_pd(final_VirtMem)(v)(pdbr(final_VirtMem)(v) ^ PAGE_FRAME) AND          10
       % mapping x lays in physical memory
       up?(virt_to_phy(final_VirtMem)(v,x)) AND
       phy_pd(final_VirtMem)(v)
              (addr2bv(down(virt_to_phy(final_VirtMem)(v,x))) ^ PAGE_FRAME)

 % Enter coalgebra structure for Space_t

 Space_t_c : [Space_tState -> Space_tIFace[Space_tState , max]] =
   (LAMBDA(x: Space_tState) :
     (#                                                                           20
       % consistent?(x) : [[FVirtMem[max]] -> bool]
       consistent? := (LAMBDA(v : FVirtMem[max]) :  space_consistent?(x,v)),
```

Listing 6.1: A model for Space_t (file Space_t_model.pvs), Part I

6.1 A Space_t Coalgebra

It is necessary to write PVS input now. PVS organizes its material in *theories*: each of them has a unique name and can be parameterized by types or constants. The body of a theory contains functions, constant definitions, and lemmas. To work with a theory, one has to load the file into PVS and typecheck it. After that, one can attempt to prove a lemma. PVS is an interactive theorem prover. This means it does not attempt to prove a lemma by its own, it rather presents the current proof state to the user and asks for a proof command. The commands range from simple propositional rules to very powerful strategies with automatic instantiation of quantifiers. If a proof has been finished, PVS stores it in the prf file. It can be rerun in later sessions without user interaction. For each lemma PVS records, which other lemmas have been used in its proof and also, if these other lemmas are proved yet.

Together with all utility definitions, the model that I developed is spread over several files and comprises about 800 lines of PVS code. It is impossible to explain every detail here, I rather concentrate on the main points.

Listings 6.1, 6.2, and 6.3 contain (parts of) the PVS source code from the file Space_t_model.pvs. The name of the theory is Space_tModel. It takes the amount max of available main memory as parameter. Because the type Addr is user defined, it is necessary, to insert an appropriate **IMPORTING** statement before its first use. PVS uses the same comment syntax as CCSL: comments start with a percent sign % and reach till the end of the line.

The first declaration in Line 3, Listing 6.1, defines the state space of this model. In principle, an address space consists of a hierarchy of frames for the page directory and the page tables. However the page tables that belong to an address space can be deduced from the page directory and the page directory can be identified by its starting address. Therefore I simply use the type Addr of virtual addresses as state space. This is similar to the use of references in C++ for refering to objects.

This matches the C++ point of view on objects of space_t. Internally the C++ compiler represents them as pointers to their data area (the page directory).

Now it is possible to instantiate the generated output from the CCSL compiler in Line 4 and to import the top level theory for the class specification Space_t. I also import the theory Kmem and indirectly via Kmem the theory VirtMemTheory. The theory Kmem defines and axiomatizes the three methods of the class kmem and the function memset on which v_insert depends. The theory VirtMemTheory contains a number of utility functions and many useful lemmas for virtual memory. Everything in there is proved directly from the specification VirtMem.

As next I could define the coalgebra that combines the three methods of my model. It will contain the method consistent that formalizes under which conditions a virtual address is considered as a valid object of Space_t in a given virtual memory. But this property of being a valid object of class Space_t is needed several times in the other two methods. Therefore I define a utility predicate space_consistent? in Line 7.[2] The predicate space_consistent? requires three properties from the object x of Space_t and the virtual memory v. First, the current address space of the virtual memory v itself must be consistent. The predicate phy_pd from theory VirtMemTheory (file VirtMem_theory.pvs) formalizes consistency of address spaces. It requires that all frames that can be reached via valid page directory entries and valid page table entries lay completely within the available main memory. If as is a consistent address space, then the following holds: If as converts a virtual address successful into a physical address, then this physical address is less than max.

The second condition of space_consistent? is that x must be a valid virtual address in the current address space of v (i.e., it can be transformed into a physical address without a page fault). For the third condition I regard x as a pointer to a page directory. I require that also the address space that corresponds to this page directory is consistent in the above sense.

Note, that I do not require, that the address x is size-aligned. Instead I always use only the first twenty bits of x. There is another situation, which would be considered as a terrible programming error in practice. Imagine a page directory where one of its entries points to the frame of the page directory itself, so that the page directory is used as a page table too. However, the verification showed that the FIASCO sources are robust enough, to deal with such page directories, at least with respect to the superpage challenge.

The next declaration is the coalgebra Space_t_c, which realizes all three methods of the class Space_t. The type Space_tIFace is the PVS representation of the Space_t

[2]The PVS sources in Listing 6.1 use the operator ^ from the bitvectors library. It takes a bitvector and two indices and delivers the subvector between the indices. If b is a bitvector of length 32, then b ^ PAGE_FRAME are the twenty most significant bits. PAGE_FRAME is defined as the pair $(31, 12)$ in theory Page (file page.pvs).

signature. It is generated by the CCSL compiler from the specification as a record with one field for each method.[3] In PVS record definitions start with (#. The fields of the records are named after the methods. The first field is consistent?, it simply calls the predicate, which I defined before.

The method v_insert is in Listing 6.2. It is a *hand translation* of the C++ sources in Listings 4.5 and 4.6 on pages 30 and 31. There are the following peculiarities. First, since I do not deal with access permissions, the Sections C, D, and F from the Listings 4.5 and 4.6 either vanish or are only present in a translated return statement. Second, there is no sequencing of statements in PVS. This makes it impossible to join different control paths. In the C++ sources some control paths are joined in Section F and G again. In the PVS translation, I have to copy these sections for every path of control that contains them. Third, the construction of a result of the form up($-$) corresponds to a return statement in the C++ source. For instance the return statement in line 21, Listing 4.5 returns control after writing one value to the virtual memory. In the PVS translation, a new virtual memory v_2 is obtained by the write in line 11, Listing 6.2. Then in line 13, this is packed together with the flag Insert_ok to return the result together with the successor states.

Fourth, page faults and a failure of the assert statement (line 20, Listing 4.6) are modeled by returning bot. For instance, the $*$p in line 9, Listing 4.5 is an access to the virtual memory at address p. The PVS translation first tests in line 5, Listing 6.2 if the access to address p produces a page fault. If not, the contents of the memory cell is load into the PVS variable star_p. In line 11, the test if writing at address p fails is omitted, because I can prove from the assertions of VirtMem, that at this point the write always succeeds.[4]

Finally, I do not use symbolic names for the positions of the status bits in the page directory and page table entries. Here, only the superpage flag (at bit number seven) and the "valid entry flag" (at bit number zero) are taken into account. They are set for instance in line 12 with the **WITH** expression of PVS.[5]

The translation of the method switchin_context is in Listing 6.3. It only checks if x can be translated into a physical address in the virtual memory v. If so, it sets the page directory base register of v.

This completes the definition of a model of the Space_t signature. The CCSL compiler combines all assertions of Space_t into one predicate Space_tAssert?. So the Theorem assert_ok in line 11 in Figure 6.3 states that for all possible states in Space_tState, all assertions from Space_t hold for the methods in Space_t_c.

My approach of using a *hand translation* of the original FIASCO sources needs certainly some justification. Strictly speaking the results of the case study apply only to the

[3] The typeconstructor Space_tIFace can be seen as a functor. Then, Space_t_c fits into the categorical definition of coalgebras.

[4] Recall that I disregard memory access permissions in this case study.

[5] Bitvectors of length 32 are functions from $[0,\ldots,31]$ to $\{0,1\}$. If f is a function, then f **WITH** $[\ (n) := a\ ]$ denotes the same function like f, except that it maps n to a. So if b is a bitvector, then b **WITH** $[\ (0) := 1\ ]$ is b with bit 0 set to 1.

```
% v_insert(x) : [[FVirtMem[max] , Addr , Addr , int]
%                 -> Lift[[Space_tState , FVirtMem[max] , Status_t]]]
v_insert := (LAMBDA(v: FVirtMem[max] , phys: Addr, virt: Addr , size: nat) :
  LET p = pde_addr(addr2bv(x),virt)
  IN  IF bot?(read(v,p)) THEN bot                                % Section A
  ELSE LET star_p = down(read(v,p))
   IN IF NOT VALID_PDE(star_p) THEN
       (IF ((size = SUPERPAGE_SIZE)
         AND ((addr2bv(virt) ^ SUPER_OFF) = bvec0[20])
         AND ((addr2bv(phys) ^ SUPER_OFF) = bvec0[20]))
     THEN  LET v2 = down(write(v,p, addr2bv(phys)
                            WITH  [(0) := 1, (7) := 1]))
          IN up(x,v2,Insert_ok : Status_t)
       ELSE   % can't map a Superpage - alloc new page table        Section B
         LET new_pt = kmem_page_alloc(v,x) IN
         IF( new_pt = bvec0[32] )
         THEN up(x,v,Insert_err_nomem : Status_t)
         ELSE LET
                v2 = down(memset(final_VirtMem) (v,new_pt, bvec0[32], PAGE_SIZE)),
                v3 = down(write(v2,p,
                          kmem_v_t_p(final_VirtMem)(v2,new_pt)  WITH [0 := 1])),
                e = pte_addr(new_pt, virt) IN
           IF( NOT size = PAGE_SIZE ) OR ( bot?(read(v3,e)))
           THEN bot
           ELSE LET star_e = down(read(v3,e)) IN                   % Section F
             IF VALID_PTE(star_e)
             THEN up(x,v, Insert_err_exists : Status_t )
             ELSE                                                  % Section G
               LET v4 = down(write(v3,e, addr2bv(phys) WITH [0 := 1]))
               IN up(x,v4,Insert_ok : Status_t)
             ENDIF
           ENDIF
         ENDIF
       ENDIF)
     ELSIF SUPER_PAGE(star_p)                                      % Section C
     THEN up(x,v,Insert_warn_exists : Status_t)
     ELSE                                                          % Section E
       let e = pte_addr( kmem_p_t_v(final_VirtMem)(v,star_p),  virt) IN
       IF( NOT size = PAGE_SIZE ) OR ( bot?(read(v,e)))
       THEN bot
       ELSE LET star_e = down(read(v,e)) IN                        % Section F
         IF VALID_PTE(star_e)
         THEN up(x,v, Insert_err_exists : Status_t )
         ELSE                                                      % Section G
           LET v4 = down(write(v,e, addr2bv(phys) WITH [0 := 1]))
           IN up(x,v4,Insert_ok : Status_t)
         ENDIF
       ENDIF
     ENDIF
  ENDIF
 ),
```

Listing 6.2: A model for Space_t (file Space_t_model.pvs), Part II

```
    % switchin_context(x) : [[FVirtMem[max]] --> Lift[[Space_tState , FVirtMem[max]]]]
    switchin_context := (LAMBDA(v: FVirtMem[max]) :
        IF up?(virt_to_phy(v,x)) THEN
            up(x,set_pdbr(v,addr2bv(down(virt_to_phy(v,x)))))
        ELSE
          bot
        ENDIF
    )
  #))

 assert_ok : THEOREM Space_tAssert?(Space_t_c)
END Space_tModel
```

Listing 6.3: A model for Space_t (file Space_t_model.pvs), Part III

translated PVS code of this section and not to the original C++ sources. Ideally one should directly verify the sources of the microkernel, or even better the compiled object code. Working with the object code is not an option, because one loses the abstraction level of the original source.

To verify the sources of FIASCO one would need a formal semantics of C++. However, C++ is so complicated that a formal semantics of it—if it can be developed at all—would be unusable. There are two possible solutions. First one could require that the microkernel is rewritten in a language, for which a formal semantics exists or can be easily developed. An example of such a language is SML [MTH91, MT91]. This approach is followed in Ensemble project [Hay98, LKvR+99]. Although a verified operating system written in a functional programming language would be an interesting academic project, it is unlikely that such a system would ever be used in practice. The second solution is to define a well–behaved subset of C++ and develop a semantics just for this subset. On the basis of this semantics, it would be possible to write a compiler that translates restricted C++ sources into PVS.

For this case study there is no feasible alternative to translating the FIASCO sources by hand into PVS. The amount of translated code is small and together with a structure chart of the C++ sources it is easy to verify the translation.

6.2 The Proof

From the discussion about the FIASCO source on page 32 we already know that the proof of assert_ok cannot be completed, because for certain parameters and states of the virtual memory, the assert statement in line 20, Listing 4.6 (page 31) fails and the method v_insert does not terminate normally. This was not known to me prior to the verification attempt. It became apparent when I tried to prove that the coalgebra Space_t_c fulfills the assertion insert_up (see Listing 5.5 on page 5.5).

However, all the other assertions hold. I also tried to localize the problem for the abnormal termination of v_insert. The theory Space_tModel (file Space_t_model.pvs) contains three lemmas, which prove within PVS the following proposition.

Proposition 6.1 *Assuming, that the Space_t object x and the virtual memory v form a consistent pair and that v_insert is called with the arguments phys and virt, and size = SUPERPAGE_SIZE. Then the assert statement fails in the following cases.*

1. *The page directory entry for virt is valid, but does not describe a superpage mapping.*
2. *The page directory entry for virt is invalid, but the physical address phys is not size-aligned for superpages and the memory allocation via kmem::page_alloc is successful.*
3. *Similar to Item 2, the page directory entry for virt is invalid, but this time the virtual address virt is not size-aligned for superpages and the memory allocation via kmem::page_alloc is successful.* □

Under the additional hypothesis that v_insert terminates normally in the listed three cases, it is possible finish the proof of the assertion insert_up. From this we can conclude the following.

Theorem 6.2 (Superpage Challenge) *The method v_insert inserts superpage mappings correctly, unless it is called in one of the conditions of Lemma 6.1. In these cases it does not terminate normally.* □

The PVS sources use about 230 utility lemmas for this proof and there are about 150 type correctness conditions to discharge. About one third of the utility lemmas was generated from the CCSL compiler. The compiler also contributed some of their proofs. All proofs of the case study contain about $4,000$ PVS proof commands. It is fair to say that this case study was not only a challenge for coalgebraic specification. The size of the PVS sources and the complicated type correctness conditions resulted in about 10 bug reports for PVS. Originally the case study was developed with PVS version 2.2 patch level 1.46. At the time of writing, PVS 2.3 patch level 1.2.2.36 is available. But this newest version of PVS still coredumps on the case study.

It is clearly impossible to give even an outline of the whole proof here. In the following I give an overview over the involved theories. Then I present three selected problems in detail.

During the case study I found some flaws in the bitvector library that is shipped with PVS 2.2. The theory bv_extra_manipulations (file bv_extra_manipulations.pvs) corrects a few lemmas of the bitvectors library and contains some more general results about bitvectors, which were necessary in my proofs.

The theory Page (file page.pvs) defines the bit windows in virtual addresses that are used as index in the page directory and in the page tables. It defines the utility functions for the VirtMem specification and contains trivial results about these functions.

The theory VirtMemTheory (file VirtMem_theory.pvs) contains rewrite lemmas for the VirtMem specifications. Rewrite lemmas are lemmas in the form of a conditional equation. In this special form they can be used very efficiently with builtin strategies of PVS to simplify proof goals or even to prove them. The file pvs-strategies contains new proof commands for PVS that allow the user to load the assertions of the VirtMem specification as rewrite lemmas.

The theory Kmem axiomatizes the method page_alloc and the function memset and defines the methods virt_to_phys and phys_to_virt of class kmem. Such axiomatizations are always problematic because they can introduce inconsistencies. However, for this work I had to make a compromise with respect to the time that I could spend on the whole case study. For the axioms I decided to use basic properties about the frames and physical addresses that these functions manipulate. The same theory Kmem contains lemmas that derive properties about whole address spaces from the axioms. These more abstract lemmas are then used in the proofs of the Space_t assertions.

In the remaining subsections I focus on a few particular problems to give you a more concrete impression about the actual verification.

6.2.1 The Assertion switchin_context_up

The assertion switchin_context_up states that the method switchin_context does not produce a page fault (see Listing 5.5 on page 44 and the discussion on page 43). The theory Space_tModel contains the following proposition that states that the coalgebra Space_t_c fulfills the assertion switchin_context_up.

switchin_context_up_ok : **PROPOSITION FORALL** (x: Space_tState):
    switchin_context_up?(Space_t_c)(x)

This is very easy to prove, because it follows almost directly from the assumption about consistency in the assertion. Two PVS commands are sufficient to prove switchin_context_up in PVS. First I install a set of auto rewrite lemmas, then I invoke REDUCE.[6]

6.2.2 Consistency of Address Mappings

Consider line 11 in Listing 6.2 on page 50 (or alternatively line 16 in Listing 4.5 on page 30). In these lines, we have an object x of Space_t and a virtual memory v and write something into the page directory of x at address p, thereby changing v. For clarity, let as denote the address space, which is currently in use in the virtual memory v. So in Line 11 in Listing 6.2 the page directory and page tables of as are used to transform p into a physical address, then the memory cell at this physical address is updated. Because as is located in the virtual memory v itself, it is not obvious that this write operation does not change as. If as were indeed changed, then it would also be possible that x refers to different frames before and after the write operation. Then it would be impossible to prove that the code inserts superpage mappings correctly.

It is therefore a crucial part of the whole correctness proof to show that this write operation does not change the current address space as. A helpful observation is that

[6] REDUCE is one of the most powerful PVS proof commands. Among other things it does repeatedly skolemization, instantiation, if–lifting, and simplification.

the write operation under discussion only happens, if the memory cell at p contains an invalid page directory entry. Because valid page table entries are recognized on the same bit (number zero), the memory cell at address p cannot be used as page directory entry or page table entry in as while translating p into a physical address. This observation can be generalized to the following.

Proposition 6.3 *Assume, that the method v_insert executes the lines 16 to 19 in Listing 4.5. If the currently used address space as maps a virtual address b to a physical address d before the execution of lines 16 to 19, then after the execution, as maps b still to d.*

Note that the lemma does not exclude that the current address space as is extended during the execution of Lines 16 to 19 of Listing 4.5. The PVS version of this lemma is called map_super and is contained in theory Space_tModel (file Space_t_model.pvs). It reads as follows.

```
map_super : PROPOSITION
  FORALL( x : Space_tState, v : FVirtMem, a : Addr, d : bvec[32] ) :
      space_consistent?(x,v) AND
      VALID_PDE( d ) AND
      up?(virt_to_phy(final_VirtMem)(v,a)) AND
      NOT VALID_PDE( down(read(final_VirtMem) (v, a )))
  IMPLIES
      virt_to_phy(final_VirtMem)
            (down(write(final_VirtMem)(v, a, d)), x) =
        virt_to_phy(final_VirtMem)(v,x)
```

Actually the PVS version is a bit more general then Lemma 6.3, in that it does not refer to the code, but states the property merely in terms of the methods of VirtMem. The PVS proof follows the idea above and needs only 40 proof commands, because it can use powerful characterizations of virt_to_phy from theory VirtMemTheory, file VirtMem_theory.pvs. The lemma map_super is used in other utility lemmas and in the proof of the assertion insert_consistent. It simplifies the proof state considerably in these other lemmas.

6.2.3 Ordering of Bitvectors

As last example I show how difficult it can be to convince a theorem prover of evident things. Consider addresses $a \leq b \leq c$ and the function addr2bv, which converts addresses (i.e., numbers less than 2^{30}) into their bitvector representation. From the representation of binary numbers, we can conclude immediately that if the 20 most significant bits of addr2bv(a) equal the 20 most significant bits of addr2bv(c) then also addr2bv(b) has the same 20 most significant bits. In PVS this looks so.

```
msb_equal : LEMMA FORALL(a, b, c : Addr) :
```

```
a <= b AND b <= c AND
addr2bv(a) ^ PAGE_FRAME = addr2bv(c) ^ PAGE_FRAME
  IMPLIES
    addr2bv(a) ^ PAGE_FRAME = addr2bv(b) ^ PAGE_FRAME
```

Recall, that PAGE_FRAME is defined as the interval $(31, 12)$ and that ^ cuts the selected bits out of a bitvector. The Lemma msb_equal is used to show that the current address space as does not change during the memset function. Assume a and c denote the start and the end of a frame, which is initialized with zeros by memset. From the properties of kmem::page_alloc I know that a and c refer to a frame that is not used in the address space as. If b is the address of a page directory entry or a page table entry, then I know that the 20 most significant bits of b are not equal to the 20 most significant bits of a. Turning Lemma msb_equal on its head[7] I can conclude that b does not lay between a and c, so the memory contents at address b is not changed.

```
% Addresses correspond to bitvectors
addr2bv( a : Addr ) : bvec[32] = nat2bv[30](a) o bvec0[2]

bv2addr( b : bvec[32] ) : Addr = bv2nat[30](b ^ RELEVANT)

addr2bv_inv : LEMMA FORALL( b : bvec[32] ) :
   addr2bv(bv2addr(b)) ^ RELEVANT = b ^ RELEVANT

bv2addr_inv : LEMMA FORALL( a : Addr ) :
      bv2addr(addr2bv(a)) = a
```

Listing 6.4: Definition of bv2addr and addr2bv (file page.pvs)

The rest of this section demonstrates in some detail, how msb_equal is proved in PVS. Readers not interested in PVS details can directly jump to the conclusions.

For the proof, I have to explain how the function addr2bv works. Its definition is in Listing 6.4. The design decision to model only a memory of words and to understand the addresses as addresses to memory cells of 32 bits length manifests itself in Listing 6.4 (compare the discussion in Chapter 5 on page 35).

To compute the bitvector from an address, the address is first transformed into a bitvector of length 30 with the function nat2bv[30] from the bitvectors library. The parameter instanciation [30] defines the length of the bitvector on which nat2bv works. Then two zero bits are appended.[8] The function addr2bv has an inverse, bv2addr. It takes a bitvector of length 32, corresponding to an byte address, cuts off the two least

[7] If P implies Q, then $\neg Q$ implies $\neg P$. Using the latter instead of the former is usually called indirect proof.

[8] The operator $\circ$ concatenates bitvectors and bvec0 is the bitvector which contains only zeros.

significant bits[9] and uses the inverse of nat2bv, the function bv2nat to compute the corresponding natural number. The two Lemmas addr2bv_inv and bv2addr_inv state that the two functions are inverse of each other. They are relatively easy to prove.

In the proof of msb_equal I will need this lemma.

```
msb_less : LEMMA FORALL( ah, al, bh, bl : nat ) :
  al < exp2(10)   AND   bl < exp2(10) AND
  ah * exp2(10) + al  <=  bh * exp2(10) + bl  IMPLIES
      ah <= bh
```

The term exp2(10) is the ASCII representation of 2^{10}. Think of a and b as bitvectors, and let al and bl denote the value of the least significant ten bits. Let ah and bh be the value of the other more significant bits. Then the value of a equals $\mathsf{ah} \times 2^{10} + \mathsf{al}$. The Lemma msb_less states the triviality that in case the value of a is less or equal then the value of b, then also the value of the more significant bits ah must be less or equal than the value of the more significant bits bh. PVS has builtin decision procedures for linear inequations, so the single proof command GRIND[10] can prove msb_less.

Now I invoke the prover on msb_equal. PVS opens a new window and shows the initial proof goal.

```
msb_equal :

  |-------
{1}  FORALL (a: Addr, b: Addr, c: Addr):
       a <= b AND b <= c AND
       addr2bv(a) ^ PAGE_FRAME = addr2bv(c) ^ PAGE_FRAME
         IMPLIES addr2bv(a) ^ PAGE_FRAME = addr2bv(b) ^ PAGE_FRAME
```

PVS uses a sequent calculus (see [SORSC98] for details). The single sequents are numbered on the left hand side. The current assumptions (antecedents) are above the dashed line and what we have to prove is below (consequents). The full tree of the completed proof is shown in Figure 6.5. In the following I discuss most of its branches.

To start the proof I first install some definitions and auto-rewrite formulae, introduce meta constants for the all–quantified variables (this is called skolemization in PVS) and simplify the goal. This yields

```
[-1]   a!1 <= b!1
[-2]   b!1 <= c!1
{-3}    nat2bv[30](a!1) ^ (29, 10) = nat2bv[30](c!1) ^ (29, 10)
  |-------
{1}    nat2bv[30](a!1) ^ (29, 10) = nat2bv[30](b!1) ^ (29, 10)
```

The variables a!1, b!1, c!1 are the meta variables, which have been introduced into the proof. The antecedents still contain the inequation and the additional assumption, that

[9] The interval RELEVANT is defined as $(31, 2)$.

[10] The super–duper strategy of PVS.

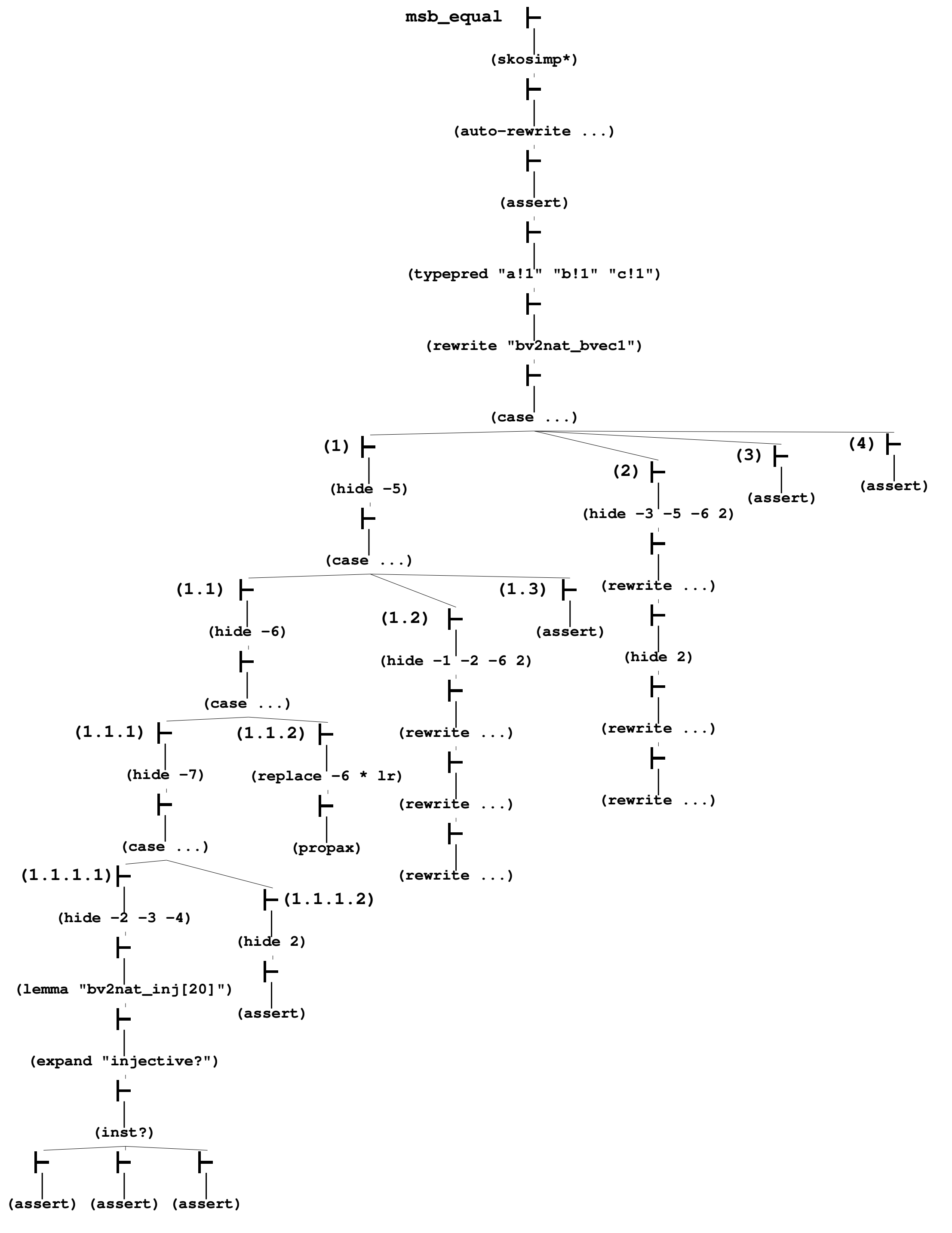


Figure 6.5: Prooftree of the proof for Lemma msb_equal

the 20 most significant bits of a and c are equal. In the remainder of the proof, the side conditions that a, b, and c are numbers less than 2^{30} is needed a number of times, so I introduce these side conditions as assumptions into the proof. This yields

```
{-1}   a!1 <= exp2(30) - 1
{-2}   b!1 <= exp2(30) - 1
{-3}   c!1 <= exp2(30) - 1
[-4]   a!1 <= b!1
[-5]   b!1 <= c!1
[-6]   nat2bv[30](a!1) ^ (29, 10) = nat2bv[30](c!1) ^ (29, 10)
  |-------
[1]    nat2bv[30](a!1) ^ (29, 10) = nat2bv[30](b!1) ^ (29, 10)
```

The strategy to complete the proof is now as follows. As I said before, PVS has powerful builtin decision procedures for inequations. But the sequents [−6] and [1] are equalities of bitvectors of length 20. So the aim is to transfer the proof goal into a proof obligation on natural numbers. This can be done by applying `bv2nat` to both sides of [−6] and [1]. To prove the resulting equation, I also need the inequations [−4] and [−5] in similar form. I introduce the needed sequents with case distinctions. A *case distinction* inserts a new formula into the proof. The proof is split into at least two new branches, the first contains the new formula as antecedent, so it is possible to use it as an assumption in what follows. In the second branch the formula is inserted as subsequent and the user has to prove that the new formula follows from the current proof state.

The first case distinction is on

```
bv2nat[20](nat2bv[30](a!1) ^ (29, 10)) <= bv2nat[20](nat2bv[30](b!1) ^ (29, 10))
```

This generates four subgoals. The additional two stem from the typechecker, which could not deduce that the above formula is type correct. Let me discuss the new subgoals in the converse order. The fourth new subgoal is

```
msb_equal.4 (TCC):

[-1]   a!1 <= exp2(30) - 1
[-2]   b!1 <= exp2(30) - 1
[-3]   c!1 <= exp2(30) - 1
[-4]   a!1 <= b!1
[-5]   b!1 <= c!1
[-6]   nat2bv[30](a!1) ^ (29, 10) = nat2bv[30](c!1) ^ (29, 10)
  |-------
{1}    a!1 < exp2(30)
[2]    nat2bv[30](a!1) ^ (29, 10) = nat2bv[30](b!1) ^ (29, 10)
```

The curly braces[11] tell us that this proof goal is a type correctness condition (TCC) for applying `nat2bv[30]` to `a!1`.[12] This subgoal can be proved with basic properties of linear

[11] PVS marks new and changed sequents with curly braces.

[12] The function `nat2bv[30]` is only defined for numbers that fit into bitvectors of length 30, that is, numbers that are less than 2^{30}.

inequations, so I apply the proof command ASSERT and PVS reports

```
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of msb_equal.4.
```

Subgoal msb_equal.3 is the same type correctness condition for b!1. It is discharged in the same way.

In subgoal msb_equal.2 I have to prove the equation, which I want to introduce into the proof. After deleting sequents that do not contain valuable information in this branch of the proof, the current goal is:

```
msb_equal.2 :

[-1]   a!1 <= exp2(30) - 1
[-2]   b!1 <= exp2(30) - 1
[-3]   a!1 <= b!1
  |-------
[1]    bv2nat[20](nat2bv[30](a!1) ^ (29, 10))
         <= bv2nat[20](nat2bv[30](b!1) ^ (29, 10))
```

If you look carfully at sequent [1], you see that this follows from Lemma msb_less, where ah and bh are instanciated with both terms of sequent [1]. PVS is not clever enough to find the right substitution for al and bl, so I apply the lemma with a partial substitution. The Lemma msb_less has three assumptions. PVS automatically deduces the first two from the current goal. The conclusion of the lemma matches exactly what I have to prove, so the proof is finished if I can establish the third assumtion of Lemma msb_less. PVS does all these bookkeeping jobs for me and when I apply the Lemma, I get the following new proof state.

```
[-1]   a!1 <= exp2(30) - 1
[-2]   b!1 <= exp2(30) - 1
[-3]   a!1 <= b!1
  |-------
{1}    bv2nat[10](nat2bv[30](a!1) ^ (9, 0))
         + bv2nat[20](nat2bv[30](a!1) ^ (29, 10)) * exp2(10)
         <= bv2nat[10](nat2bv[30](b!1) ^ (9, 0))
           + bv2nat[20](nat2bv[30](b!1) ^ (29, 10)) * exp2(10)
[2]    bv2nat[20](nat2bv[30](a!1) ^ (29, 10))
         <= bv2nat[20](nat2bv[30](b!1) ^ (29, 10))
```

Sequent [2] is not important any longer, I concentrate on proving {1}. (In what follows I will silently delete superfluous sequents, to keep the displays short.) Sequent {1} looks a bit difficult, but it is only a verbose form of [−3]. An application of the Lemma bvconcat2nat from the bitvectors library yields

```
[-1]   a!1 <= exp2(30) - 1
[-2]   b!1 <= exp2(30) - 1
[-3]   a!1 <= b!1
  |-------
{1}    a!1 <= bv2nat[10](nat2bv[30](b!1) ^ (9, 0))
           + bv2nat[20](nat2bv[30](b!1) ^ (29, 10)) * exp2(10)
```

A second application of the same lemma solves this branch of the proof.

I can continue now with the first subgoal after the case distinction. It is

```
msb_equal.1 :

[-1]    bv2nat[20](nat2bv[30](a!1) ^ (29, 10))
          <= bv2nat[20](nat2bv[30](b!1) ^ (29, 10))
[-2]    a!1 <= exp2(30) - 1
[-3]    b!1 <= exp2(30) - 1
[-4]    c!1 <= exp2(30) - 1
[-5]    b!1 <= c!1
[-6]    nat2bv[30](a!1) ^ (29, 10) = nat2bv[30](c!1) ^ (29, 10)
  |-------
[1]    nat2bv[30](a!1) ^ (29, 10) = nat2bv[30](b!1) ^ (29, 10)
```

I have to do the same kind of case distinction for the inequation $[-5]$. I concentrate here on the first subgoal, the others are proved in the same way as before.

```
msb_equal.1.1 :

[-1]    bv2nat[20](nat2bv[30](b!1) ^ (29, 10))
          <= bv2nat[20](nat2bv[30](c!1) ^ (29, 10))
[-2]    bv2nat[20](nat2bv[30](a!1) ^ (29, 10))
          <= bv2nat[20](nat2bv[30](b!1) ^ (29, 10))
[-3]    a!1 <= exp2(30) - 1
[-4]    b!1 <= exp2(30) - 1
[-5]    c!1 <= exp2(30) - 1
[-6]    nat2bv[30](a!1) ^ (29, 10) = nat2bv[30](c!1) ^ (29, 10)
  |-------
[1]    nat2bv[30](a!1) ^ (29, 10) = nat2bv[30](b!1) ^ (29, 10)
```

As next I transfer sequent number $[-6]$ into an equality of natural numbers by a case distinction on

```
bv2nat[20](nat2bv[30](a!1) ^ (29, 10))  =  bv2nat[20](nat2bv[30](c!1) ^ (29, 10))
```

This equation follows directly from $[-6]$, by replacing the left hand side of $[-6]$ with its right hand side. This is easily proved in PVS, so I concentrate on the first subgoal.

```
msb_equal.1.1.1 :

[-1]    bv2nat[20](nat2bv[30](a!1) ^ (29, 10))
          = bv2nat[20](nat2bv[30](c!1) ^ (29, 10))
[-2]    bv2nat[20](nat2bv[30](b!1) ^ (29, 10))
          <= bv2nat[20](nat2bv[30](c!1) ^ (29, 10))
[-3]    bv2nat[20](nat2bv[30](a!1) ^ (29, 10))
          <= bv2nat[20](nat2bv[30](b!1) ^ (29, 10))
[-4]    a!1 <= exp2(30) - 1
[-5]    b!1 <= exp2(30) - 1
[-6]    c!1 <= exp2(30) - 1
  |-------
[1]    nat2bv[30](a!1) ^ (29, 10) = nat2bv[30](b!1) ^ (29, 10)
```

Finally I translate [1] into an equation of natural numbers by a case distinction on

```
bv2nat[20](nat2bv[30](a!1) ^ (29, 10)) = bv2nat[20](nat2bv[30](b!1) ^ (29, 10))
```

The first subgoal is

```
msb_equal.1.1.1.1 :

[-1]    bv2nat[20](nat2bv[30](a!1) ^ (29, 10))
          = bv2nat[20](nat2bv[30](b!1) ^ (29, 10))
[-2]    a!1 <= exp2(30) - 1
[-3]    b!1 <= exp2(30) - 1
[-4]    c!1 <= exp2(30) - 1
  |-------
[1]   nat2bv[30](a!1) ^ (29, 10) = nat2bv[30](b!1) ^ (29, 10)
```

I have to show that sequent [1] follows from [−1]. This is easy because bv2nat[20] is an injective function. The corresponding Lemma bv2nat_inj is part of the bitvectors library. The automatic instanciation yields three subgoals; two of them are type correctness conditions. All three subgoals can be finished with the PVS command ASSERT. The only remaining subgoal of the whole proof is:

```
msb_equal.1.1.1.2 :

[-1]    bv2nat[20](nat2bv[30](a!1) ^ (29, 10))
          = bv2nat[20](nat2bv[30](c!1) ^ (29, 10))
[-2]    bv2nat[20](nat2bv[30](b!1) ^ (29, 10))
          <= bv2nat[20](nat2bv[30](c!1) ^ (29, 10))
[-3]    bv2nat[20](nat2bv[30](a!1) ^ (29, 10))
          <= bv2nat[20](nat2bv[30](b!1) ^ (29, 10))
[-4]    a!1 <= exp2(30) - 1
[-5]    b!1 <= exp2(30) - 1
[-6]    c!1 <= exp2(30) - 1
  |-------
[1]   bv2nat[20](nat2bv[30](a!1) ^ (29, 10))
          = bv2nat[20](nat2bv[30](b!1) ^ (29, 10))
```

Note, that this is the original goal of this proof (see the Display of msb_equal on page 56), but everything converted to natural numbers. PVS can prove this with its inequality reasoning by the command ASSERT. This completes the whole proof and I get

```
This completes the proof of msb_equal.1.1.1.2.
This completes the proof of msb_equal.1.1.1.
This completes the proof of msb_equal.1.1.
This completes the proof of msb_equal.1.
Q.E.D.
```

7 Conclusions

This paper reports on a case study of applying coalgebraic specification to *real software*. For one module of the microkernel FIASCO (the memory management) I developed a specification and checked mechanically whether the FIASCO sources fulfill this specification. The whole case study consists of about 5,000 line of source code. About 4,000 PVS commands prove in 230 lemmas and 150 type correctness conditions that the method v_insert of class space_t inserts superpage mappings correctly if its arguments fulfill a side condition. The case study also shows that, in case the arguments violate the side condition, an assert statement in the method v_insert fails. See Proposition 6.1 and Theorem 6.2 on page 52 for details.

The proof relies on a few axioms. Two of them, which state the existence of final models for the specifications PhyMem and VirtMem, follow from the general theory about coalgebras [Rut96]. Nevertheless, theorems stating the same property as the axioms have been proved in the case study; see Section 5.1 and 5.2.[1] The remaining two axioms formalize properties of foreign functions, which are used in the code of v_insert. In principle, also these last two axioms could be proved within PVS if these foreign functions were programmed in PVS. However, this would be a considerable amount of work.

The specifications in this case study are very complex. This is partly reflected in the large number of type correctness conditions. To typecheck the whole specification and to run all the proofs takes more than half an hour on a 500MHz Pentium box.

To complete the case study as it is described in this paper took about four months of work. So it is fair to say that, with extensive use of state of the art tools, it is possible to apply coalgebraic specification and verification techniques to real software. It is also clear however, that for larger applications to imperative programs the specification language CCSL and its compiler must be developed further. Mainly two problems must be solved. First, the translation of the sources into the theorem prover must be automated; see the discussion at the end of Chapter 6 on page 49.

The second problem is that, at the moment, CCSL lacks any support for imperative programming. It is quite cumbersome to state and proof properties about a series of successor states of one object (compare assertion insert_ok in Listing 5.5 on page 44). For a larger application, CCSL should provide syntactic sugar that makes it easy to write such assertions. The state of the object should be propagated automatically, without that the user has to extract it every time from the result of the previous methods.

A widely used tool for such state propagation are *monads* [Wad90]. It is a topic for further research, how monads can be incorporated into CCSL. Although it is not

[1] PVS lacks an appropriate internal notion of theory refinement. Therefore it was not practical to replace the axioms by the theorems.

explicit in [JvdBH+98], the coalgebraic semantics of Java uses a special monad and its composition for the combination of Java statements. From this work it is also known that this approach can yield highly automated proofs in PVS.

8 Bibliography

[BSGM97] R. W. Butler, M. K. Srivas, D. A. Greve, and S. P. Miller. *A new bitvectors library for PVS*, January 1997.

[BW84] M. Barr and C. Wells. *Toposes, Triples and Theories*. Springer–Verlag, 1984. Grundlehren der mathematischen Wissenschafter, Volume 278.

[CF92] R. Cockett and T. Fukushima. About Charity. Yellow Series Report 92/480/18, Dept. of Computer Science, Univ. of Calgary, June 1992.

[Cîr99] C. Cîrstea. A coalgebraic equational approach to specifying observational structures. In B. Jacobs and J. Rutten, editors, *Coalgebraic Methods in Computer Science*, number 19 in ENTCS. Elsevier, Amsterdam, 1999.

[Gum99] H. P. Gumm. Elements of the general theory of coalgebras. Preliminary version, available at http://www.mathematik.uni-marburg.de/ gumm/Papers/publ.html, 1999.

[Hay98] M. G. Hayden. *The Ensemble System*. PhD thesis, Cornell University, 1998.

[HBB+98] H. Härtig, R. Baumgartl, M. Borriss, Cl.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter. DROPS: OS support for distributed multimedia applications. In *Proceedings of the Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.

[Hen99] U. Hensel. *Definition and Proof Principles for Data and Processes*. PhD thesis, Univ. of Dresden, Germany, 1999.

[HHJT98] U. Hensel, M. Huisman, B. Jacobs, and H. Tews. Reasoning about classes in object–oriented languages: Logical models and tools. In Ch. Hankin, editor, *European Symposium on Programming*, number 1381 in Lecture Notes in Computer Science, pages 105–121. Springer, Berlin, 1998.

[HHL+97] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-Kernel-based systems. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, volume 31,5 of *Operating Systems Review*, pages 66–77, New York, October 5–8 1997. ACM Press.

[HJ99] U. Hensel and B. Jacobs. Coalgebraic theories of sequences in PVS. *Journal of Logic and Computation*, 9(4):463–500, 1999.

[HK99] R. Hennicker and A. Kurz. (Ω, Ξ)–Logic: On the algebraic extension of coalgebraic specifications. In B. Jacobs and J. Rutten, editors, *Coalgebraic Methods in Computer Science '99*, volume 19 of *ENTCS*, pages 195–212. Elsevier, 1999.

[Hoh98] M. Hohmuth. The Fiasco kernel: Requirements definition. Technical Report TUD–FI–12, TU Dresden, December 1998. Available from URL: `http://os.inf.tu-dresden.de/papers_ps/fiasco-spec.ps.gz`.

[Hoh00] M. Hohmuth. The Fiasco kernel: System architecture. Unpublished manuscript, 2000.

[Jac96] B. Jacobs. Objects and classes, co–algebraically. In B. Freitag, C.B. Jones, C. Lengauer, and H.-J. Schek, editors, *Object–Orientation with Parallelism and Peristence*, pages 83–103. Kluwer Acad. Publ., 1996.

[Jac97] B. Jacobs. Invariants, bisimulations and the correctness of coalgebraic refinements. In M. Johnson, editor, *Algebraic Methodology and Software Technology*, number 1349 in Lecture Notes in Computer Science, pages 276–291. Springer, Berlin, 1997.

[JR97] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997.

[JvdBH+98] B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about classes in Java (preliminary report). In *Object–Oriented Programming, Systems, Languages and Applications*, pages 329–340. ACM Press, 1998.

[LBR99] G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Techn. Rep. 98-06, Dep. of Comp. Sci., Iowa State Univ. (`http://www.cs.iastate.edu/~leavens/JML.html`), 1999.

[LKvR+99] X. Liu, C. Kreitz, R. van Renesse, J. Hickey, M. Hayden, K. Birman, and R. Constable. Building reliable, high-performance communication systems from components. In *Proceedings of the 17th Symposium on Operating Systems Principles (SOSP-99)*, volume 34,5 of *Operating Systems Review*, pages 80–92. ACM Press, December 1999.

[ML71] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer–Verlag, 1971.

[MT91] R. Milner and M. Tofte. *Commentary on Standard ML*. MIT Press, 1991.

[MTH91] R. Milner, M. Tofte, and R. Harper. *The Definition of Stamdard ML*. MIT Press, Cambridge, MA, 1991.

[ORR+96] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *Computer Aided Verification*, number 1102 in Lecture Notes in Computer Science, pages 411–414. Springer, Berlin, 1996.

[ORSvH95] S. Owre, J.M. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Trans. on Softw. Eng.*, 21(2):107–125, 1995.

[Pau94] L. C. Paulson. *Isabelle: A Generic Theorem Prover*. Number 828 in Lecture Notes in Computer Science. Springer, Berlin, 1994.

[Rei95] H. Reichel. An approach to object semantics based on terminal co–algebras. *Mathematical Structure in Computer Science*, 5:129–152, 1995.

[Röß] M. Rößiger. From modal logic to terminal coalgebras. submitted to Theoretical Computer Science.

[Rot00] J. Rothe. Modal logic for coalgebraic class specification. Diploma thesis, Dresden University of Technology, 2000.

[Rut96] J. Rutten. Universal coalgebra: a theory of systems. CWI Report CS-R9652, 1996.

[SORSC98] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, version 2.2 edition, September 1998.

[Tew00a] H. Tews. *The CCSL Reference manual*, 2000. available via URL http://www.cs.kun.nl/~bart/LOOP/.

[Tew00b] H. Tews. Coalgebras for binary methods. to appear in *Coalgebraic Methods in Computer Science*, Volume 33 of ENTCS, 2000.

[Wad90] P. Wadler. Comprehending monads. In *Conference on Lisp and Functional programming*, pages 61–78, June 90.

[Wir90] M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 13, pages 673–788. Elsevier/MIT Press, 1990.