Three selected case studies in verification

Thomas Fuchß, Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel

Angaben zur Veröffentlichung / Publication details:

Fuchß, Thomas, Wolfgang Reif, Gerhard Schellhorn, and Kurt Stenzel. 1995. "Three selected case studies in verification." In KORSO: methods, languages, and tools for the construction of correct software; final report, edited by Manfred Broy and Stefan Jähnichen, 369–87. Berlin: Springer. https://doi.org/10.1007/bfb0015472.

PART V

Case Studies

Case studies are badly needed to demonstrate, test, and evaluate new approaches to software development. They illustrate the proceeding and help to get some idea of the required overhead. With some medium sized case studies the KORSO project demonstrated that formal methods are ready for the use in industrial practice.

The first paper of this part, **Three Selected Case Studies in Verification**, gives a description and motivation of the usage of the KIV system. By reference to three small examples for theorem proving a detailed productivity analysis for the KIV system is given.

Most of the work in this field was carried out on the case study "Production Cell". Nearly all the partners were involved in this study, so that a separate book is devoted to it. The authors of the **Case Study "Production Cell": A Comparative Study in Formal Specification and Verification** therefore confine themselves to giving an overview of this work, while attempting to put the results into perspective.

Even the last paper in this volume, **The KORSO Case Study for Software Engineering with Formal Methods: A Medical Information System**, is a survey on work done in the biggest case study of the KORSO project. It focuses on requirements analysis supported by formal specification techniques. And we are proud of the main result of this case study: Formal methods scale up in a pleasant way.

Three Selected Case Studies in Verification

Thomas Fuchß, Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel

Institut für Logik, Komplexität und Deduktionssysteme
Universität Karlsruhe

Abstract. In this paper we highlight three verification case studies with the KIV system (*Karlsruhe Interactive Verifier*, [HRS90]). The case studies pursue different aims and illustrate different aspects of verification. The paper reports on our experiences, presents a detailed productivity analysis of the KIV system, and gives an impression of the typical verification problems that show up in practical applications. Furthermore, these case studies can serve as a challenge for other verification systems.

1 Introduction

The three case studies are concerned with three different applications: data security, translation of E/R-diagrams to formal specifications, and efficient data access using dynamic hashing.

Data security has been selected since it is a typical application where formal methods can be extremely helpful to detect and to avoid errors. Verification is required both for the algorithms that implement the access control mechanism, as well as for the requirements of the security model. The access control model sketched in the paper is generic, hence reusable, and was applied to the KORSO case study HDMS-A (see [CHL95] in this volume).

In contrast to data security, the translation of E/R-diagrams to formal specifications is a nonstandard application of verification. The original idea of such a translation is to establish a bridge between semi-formal specification methods (E/R-diagrams) and formal specifications. The problem, however, is to prove the consistency of the resulting specification. This question is turned into a verification problem by the following technique: Given an E/R-diagram, the translation not only generates a structured specification but also a structured implementation for all the types and operations in a uniform way. Then consistency reduces to the correctness of the implementation with respect to the specification. This case study was also inspired by the HDMS-A activities in KORSO.

Finally, efficient data access using dynamic hashing was selected for this paper because the verification is very difficult. We think that this case study is a big challenge for every verification system. With the help of this case study we explain some of the reasons that make verification a highly non-trivial job. Furthermore, the selected studies are quite demanding for special components of a verification system: They require a highly automated deduction component, elaborate proof engineering facilities, powerful simplification mechanisms, and an intelligent correctness management.

It is clearly impossible to communicate every detail of three case studies. Therefore, we concentrate on certain aspects of them, and refer to other papers for the details. In the following sections, we give for each case study a short introduction to the problem, describe its formalization, give detailed performance and productivity figures for the verification in the KIV system, and report on our experiences.

This paper is not completely self-contained. The formalization of the case studies is based on the KIV terminology (notions like *modules, hierarchy respecting modular systems, proof steps, degree of automatization* etc.) presented in [Rei95] and the methodology described in [PW+95] (both in this volume).

2 A Verified Access Control Model

In this case study [Ste93], which is part of the KORSO case study HDMS-A (see [CHL95] in this volume for an overview) a data access control model is specified and implemented. The correctness of the implementation as well as properties of the security model are proven.

This overview concentrates on the general ideas of the security model, presents extensive statistical figures for the verification, and comments on the results and experiences of the case study.

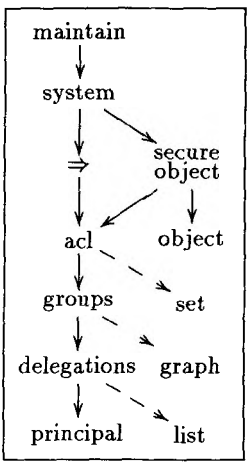
2.1 Ingredients of Access Control

In the considered model based on [ABLP91], acting subjects — called *principals* — are not atomic but structured. Principals are built up from atomic *proper principals* that denote active components of a computing environment: real persons, groups, terminals, etc. Structured principals are constructed by a conjunction of principals, by the adoption of a role, and by speaking for another principal. The conjunction of two principals A and B, A & B, models a "joint signature": Neither A nor B alone can gain access, but a request from A & B is granted. The rights of A & B are a superset of the rights of A and B. A principal A in a role B, A *as* B with B again a principal, lowers his rights to reduce the consequences of mistakes. A *as* B gains access if and only if A and B gain access. A principal may adopt any role. A principal B is able to delegate his rights to another principal A. Then A speaks for B, A *for* B, and has the rights of B. However, the existence of a delegation from B to A is checked on each request.

Further ingredients for the security model are the valid *delegations*, groups forming a *group hierarchy*, *access control lists* (ACLs) and *secure objects*. Each object of the system (files, data base entries etc.) together with an ACL forms a *secure object*. In order to determine the right to access an object, its ACL is inspected. An ACL contains principals who should gain access. However, it is not necessary to explicitly list in the ACL *all* principals whose requests are granted. Structured principals are reduced to their components, i. e. *proper* principals. Proper principals gain access if they are member of the ACL or if they are member of a group that appears in the ACL. Here the group hierarchy plays

an important role. Groups are proper principals; the hierarchy of groups forms a directed graph with edges denoting group membership. The list of valid delegations holds the currently valid (i. e. existing) delegations. The access control algorithm will use this list to determine the validity of principals *A for B*.

Formally, *delegations*, *group hierarchy*, and ACLs are just actualizations of the standard specifications of lists, directed graphs and sets[1], respectively. Figure 1 shows the structure of the specification.



$$(\forall\, A.A \in L \vee A \in R \rightarrow \mathit{is\_proper\_principal}(\mathrm{A})) \rightarrow (L \Rightarrow_{D,G} R \leftrightarrow \exists\, p_1 \in L, p_2 \in R \wedge p_1 \mathit{member}_G\, p_2)$$

$$(A \,\&\, B + L \Rightarrow_{D,G} R) \leftrightarrow (A + B + L \Rightarrow_{D,G} R)$$
$$(A \text{ as } B + L \Rightarrow_{D,G} R) \leftrightarrow (A + L \Rightarrow_{D,G} R) \wedge (B + L \Rightarrow_{D,G} R)$$

$$(L \Rightarrow_{D,G} A \,\&\, B + R) \leftrightarrow (L \Rightarrow_{D,G} A + R) \wedge (L \Rightarrow_{D,G} B + L)$$
$$(L \Rightarrow_{D,G} A \text{ as } B + R) \leftrightarrow (L \Rightarrow_{D,G} A + B + R)$$

$$A \text{ for } B \in D \rightarrow ((A \text{ for } B + L \Rightarrow_{D,G} R) \leftrightarrow (B + L \Rightarrow_{D,G} R))$$
$$\neg\ A \text{ for } B \in D \rightarrow ((A \text{ for } B + L \Rightarrow_{D,G} R) \leftrightarrow (L \Rightarrow_{D,G} R))$$

$$A \text{ for } B \in D \rightarrow ((L \Rightarrow_{D,G} A \text{ for } B + R) \leftrightarrow (L \Rightarrow_{D,G} B + R))$$
$$\neg\ A \text{ for } B \in D \rightarrow ((L \Rightarrow_{D,G} A \text{ for } B + R) \leftrightarrow (L \Rightarrow_{D,G} R))$$

Fig. 1. structure of the specification (left) and specification of the access control predicate $\Rightarrow$ (right)

2.2 Determining Access

A principal A may access a secure object *obj* with ACL *acl* under current delegations D and group hierarchy G if and only if $\{A\} \Rightarrow_{D,G} acl$ is true. The specification of $\Rightarrow$ (on the right of Fig. 1) describes the main access control mechanism[2].

This specification is monomorphic, so that all properties of the model are specified. The specification contains a "base case" where all principals of both sets are proper principals, and "reduction steps" for each principal building operation. In this way, structured principals are reduced to proper principals. In the base case, $L \Rightarrow_{D,G} R$ is true if and only if one proper principal of L is member of one proper principal in R with respect to G. The reduction steps conform to the intuitive meaning of the structuring operations on principals stated above.

[1] Though we speak of access control *lists*, only the properties of sets are of interest, the order of principals or duplicates are ignored.

[2] Delegations are treated differently from [ABLP91].

Properties are formulated as lemmas, e. g. the fact that $A \ \& \ B$ has more rights than A is stated as $\forall\ acl, D, G.\ \{A\} \Rightarrow_{D,G} acl \rightarrow \{A \ \& \ B\} \Rightarrow_{D,G} acl$.

The secure *system* is a triple consisting of the group hierarchy, the list of valid delegations, and a data base of secure objects.

2.3 The Security Policy

The security policy determines the functionality of administrative functions related to security: Who may when modify access rights, the group structure, add or delete objects etc. Of particular interest are the operations for adding and deleting subjects, adding and deleting objects, modifying the group hierarchy, the valid delegations, and ACLs. These functions are specified in a uniform way in *maintain* (Fig. 1). A predicate *may-...* determines if the calling principal may perform the operation with the given arguments. The predicate takes the same arguments as the operation, the calling principal, specific arguments for the operation and the system. If the principal may perform the operation the system is modified accordingly, otherwise it remains unchanged.

In the considered model for a general computing environment, the data of the single user is protected, but no restrictions are posed on the ability of a user to make his own data accessible to others. The main security property states that *a principal cannot increase his rights without outside help* by using the administrative functions. This property can be formulated and formally proven.

First an invariant *Inv* of the system *sys* for all administrative functions is formulated and proven. The invariant contains properties like *the group hierarchy forms an* acyclic *graph*. The security is then stated in the form *if a principal A has no access to an object and receives no outside help, then after calling an administrative function he still has no access to the object and receives no outside help*. Together with a secure (in the above sense) initial system *initsys* and the assumption that no other functions modify security relevant data, the security of the system follows. The scheme is for each administrative function f:

- Inv(initsys)
- Inv(sys) $\rightarrow$ Inv(f(A,...,sys))
- Inv(sys) $\wedge \neg$ A may-access$_{sys}$ obj $\wedge \neg$ A gets-outside-help sys
  $\rightarrow \neg$ A may-access$_{f(A,...,sys)}$ obj $\wedge \neg$ A gets-outside-help f(A,...,sys)

2.4 Implementation and Verification

The implementation of the access control model can be divided in four major parts: the access control algorithm, the implementation of graphs, the administrative functions, and several miscellaneous functions. Furthermore, additional data structures are needed. Table 1 gives some figures.

The access control algorithm implementing the predicate $\Rightarrow$ basically works by reducing principals to proper principals. For proper principals group membership is checked. The problem is comparable to a prover for propositional logic (the conjunction of principals corresponds to the conjunction of two formulas,

Table 1. Overview over the specifications and implementations

Specification of access control	
Specifications	25
Operations	111
Axioms	206
Properties	47

Additional data structures	
Specifications	15
Operations	35
Axioms	57

Implementation	
Modules	10
Lines of Code	1195
Proof obligations	222

adopting a role corresponds to the disjunction of formulas). Of course, several different implementations are possible. The selected implementation avoids case distinctions (introduced by & on the right and *as* on the left side of $\Rightarrow$) as long as possible, and checks for membership at several points during the computation.

The test for membership is a time-critical operation. Viewing the group hierarchy as a graph, membership becomes the existence of a path between two vertices. The graph implementation is chosen to allow for an efficient algorithm for this problem. A graph is implemented by an array of the vertices and their adjacency lists containing indices of the array. The correspondence between vertices and array indices is computed by binary search. The requirement for a sorted array makes the addition and deletion of vertices, i. e. proper principals, rather complicated.

Table 2 lists the figures for the verification of the implementation. The proofs of the properties of the security model and other lemmas used during the verification of the implementation complete the verification task. The first three lines depict the lines of code for every module, the number of proof obligations which must be proved to verify the correctness of the module and the number of additional lemmas which had to be invented during the verification process. The number of proof steps given refers to the number of applied tactics of the proof strategy. Typical steps are 'unfold procedure call', 'apply induction hypothesis', 'insert a lemma' or 'simplify first order formulas'. The last two lines show the percentage of proof steps automatically found by the system and the time needed to verify the modules. The table *Properties of the Specifications* encompasses all purely first order proofs, including lemmas used for simplification and security properties.

2.5 Experiences

The special features of the case study can be described in a few sentences. The specification has a rich structure, and most of the standard data types are used (naturals, lists, sets, tuples, graphs, arrays, etc. in different versions). The verification encompasses pure predicate logic proofs, and proofs involving programs. The number of lines of code divided by the time needed for verification plus the time for the development of specifications and implementations (4 weeks) gives a ratio of approx. 20 lines of verified code per day. This extremely good performance has several reasons:

Table 2. Data for the various verification tasks

Access control algorithm	
Lines of code	180
Proof obligations	48
Lemmas	98
Proof steps	9344
Automatization	75.4 %
Absolute time	3 weeks

Directed graphs	
Lines of code	287
Proof obligations	35
Lemmas	157
Proof steps	8753
Automatization	80.6 %
Absolute time	4 weeks

Administrative Functions	
Lines of code	391
Proof obligations	84
Lemmas	0
Proof steps	874
Automatization	100 %
Absolute time	5 min.

Other functions	
Lines of code	337
Proof obligations	109
Lemmas	14
Proof steps	3948
Automatization	88 %
Absolute time	4 days

Properties of the Specifications	
Operations	160
Axioms	279
Lemmas	568
Proof steps	7160
Automatization	68.1 %
Absolute time	2 weeks

Summary	
Lines of Code	1195
Proof obligations	276
Lemmas	837
Proof steps	30079
Automatization	77.5 %
Absolute time	10 weeks

- The implementation is structured into ten separate modules of rather small size: A strong modularization keeps the verification tasks manageable.
- The implementation contains some complex algorithms, but also rather trivial (though safety-critical) parts that are easy to verify — this mixture seems to be a typical feature of larger case studies.
- The use of generic specifications of standard data types saves verification effort (the same specification is used in different instances), and reduces errors because these specifications are well understood.

However, the most prominent feature of the case study was the large number of modifications of specifications, implementations, and lemmas at all points of the development and verification. The access control algorithm was implemented and verified before the administrative functions were specified or anything else implemented or proved. The specification of the administrative functions has been changed considerably several times (either due to errors — some principals could increase their rights — or to make the specification more lucid and expressive). During the verification about a dozen errors were found with consequences for specifications, implementations and already proved statements. That these problems (which arise in every non-trivial application) did not lengthen the verification considerably, is due to two other features of the KIV system:

- An effective correctness management keeps track on a very detailed level which proofs are affected by a modification.
- The automatic reuse of proofs (failed proof attempts are used in an intelligent way to guide the proofs after a correction, see [RS93] for details) minimizes the loss of work caused by modifications.

3 Translating E/R-diagrams into Consistent Database Specifications

Within the framework of the KORSO case study HDMS-A ([CHL95], this volume) we carried out the proof of consistency for an entity relationship database specification. In comparison to the other case studies presented in this paper a completely different task was tackled by the KIV approach. Not the development of correct programs was to the fore, but the examination of the suitability of the module concept as a basis for consistency proofs of specifications. The chosen method and the overall motivation is described here, for details see [Fuc94].

3.1 Translation of Entity Relationship Diagrams

In the database community entity relationships are well known structures for modeling connections between real world objects [Bek92]. Entity relationship diagrams (E/R-diagrams) contain entities of several types as objects under consideration. They are described through attributes which can be mandatory or not. Each entity type contains at least one mandatory key attribute as distinguishing feature. Between the entity types in an E/R-diagram binary relations exist. As a basis for our task we used the translation of E/R-diagrams to a structured SPECTRUM specification (see [Het93]) of a relational database, by our Munich KORSO Partners. Entity types are specified as sets of tuples of attributes with an additional key function and the relationships as sets of pairs. By this translation E/R-Diagrams should be given a formal semantic. Of course, this requires the consistency of every resulting specification.

In contrast to SPECTRUM, KIV uses a first order specification language and requires all function to be total. Therefore an explicit error handling was introduced and all sorts of the original SPECTRUM specification were enhanced by error constants.

3.2 The Proof of Consistency

To carry out the consistency proof of the database specification we used the module concept of the KIV approach for the development of correct large software systems (see [Rei92] or [Rei95]). Because module correctness guarantees the consistency of the export specification with respect to the import, it is possible to prove the consistency of a specification by an implementation on the basis of specifications for which consistency is a known fact. Therefore the output of our translation is not only a structured database specification. It is an instantiation of the template of a modular system containing specifications and prototypical implementations given in Fig. 2. The correctness of an instantiated modular system only depends on the correctness of the implemented modules, because all the other structuring operations on specifications preserve correctness. In this way we get a database specification (*ER-DB*(*Attributes*)) parameterized in the attribute specification (*Attributes*), which is consistent for every concrete and consistent instantiation of *Attributes*.

The automatic translation of an E/R-diagram to a modular system was implemented in the KIV system. The translation generates an instance of the scheme shown in Fig. 2. The correctness of every instance depends on a set of verification conditions. These conditions are also generated automatically from the components of the modular system and can be given schematically. This allows to prove that for all E/R-diagrams the proof obligations will be provable in general. The result was derived by hand in [Fuc94] and gives a "plan" how to automate the proofs for every instance in the KIV system. We are working on integrating this result into the translation program, so that appropriate lemmas and proof "plans" needed for the verification will be generated. Thereby we gain a verified modular system independent of the translation.

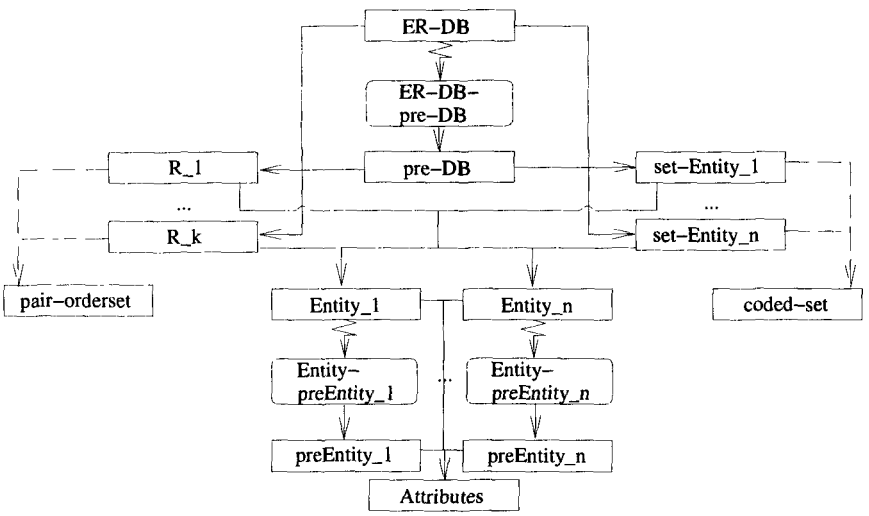


Fig. 2. Modular system: The output of the translation.

3.3 The Modular System and the Details of Translation

The modular system which is generated by the translation of an E/R-diagram is schematically shown in Fig. 2. Solid arrows from one specification to another represent the fact that the specification at the arrowhead is used in the specification at the base (or used in a renamed version). Dashed arrows point to generic specifications while the specification at the base of the arrow is an actualized version. The triple of two rectangular boxes and a box with rounded corners represent a module with export specification (at the top), implementation (in the middle) and import specification (at the bottom).

The modular system contains two specifications, which are independent of the E/R-diagram to translate. Both are assumed to be implemented in a library and therefore to be consistent. The first specification assumed to be given is

pair-orderset. This is a specification of sets of pairs, which is generic in the two parameters of the pair. This specification is used as a template for the relations $R(1\text{–}k)$ of the E/R-diagram. Relations are represented as sets of pairs of keys. To get a concrete relation both parameters åre actualized with the appropriate entity types. The other translation-independent specification is *coded-set*. It is a generic specification of sets of elements with keys. It is guaranteed that the keys of two elements are always pairwise different. Both the target sort *set* and the parameter sort *elem* contain an error element. In addition there is a selector function *sel* : *key* × *set* → *elem*. *sel(k,s)* yields the element with key *k* from set *s*, if there is one in the set. Otherwise it yields the error element. The parameter of this specification is actualized with the different entity types *Entity*(1–*n*) to give the specifications *set-Entity*(1–*n*), which define sets of entities.

Apart from these presupposed specifications the other specifications and modules are generated dependent on the given E/R-diagram by instantiating templates. The specification *Attributes* at the bottom of Fig. 2 is the parameter of the whole system. It fixes the sort, an error constant and a special *absent* value for every attribute. For each entity type (out of *n*) in the given E/R-diagram one module *Entity-preEntity*(1–*n*) is generated. Every such module exports a specification *Entity* which provides the sort *E* of the entity type. For ensuring the consistency of *Entity* an implementation is generated automatically on the basis of the specification *preEntity*, which provides the sort *pre-E* freely generated by the constructors *p-mk-E* and *p-error-E*, an error constant. *p-mk-E* builds an entity as a tuple of attributes. Since the specification corresponds to a "data"-specification in SPECTRUM it is consistent by construction (the KIV system allows such data type declarations in specifications and generates the corresponding axioms automatically).

Similar to the sort *pre-E* in the specification *preEntity*, the sort *E* of the specification *Entity* is generated by *create-E* and *error-E*, but not freely. The function *create-E* builds a tuple only if the following condition is satisfied: All attributes are different from the error attribute and no mandatory attribute has the special value *absent*. The functions of the specification *Entity* can now be implemented by simple programs in the module Entity-preEntity. Finally, the top-level database specification *ER-DB* is built up from a data type specification *pre-DB* in an analogous way to the construction of *Entity* from *preEntity*.

3.4 Application: Cardiac-Catheterisation

As an application of the generic translation of E/R-diagrams to a correct modular system, the consistency proof for the Cardiac-Catheterisation E/R-diagram taken from [HHMNS93] was carried out with the KIV system. The diagram contains five entity types, doctors, patients, catheterisation orders, entities containing the data of the catheterisation and entities for the findings of the catheterisation. Between these entities six relationships exist. Three between doctors and the entities related to the catheterisation parts (orders, data and findings), one between patients and data, one between orders and data, and one between data and findings. The 5 entities and 6 relations yielded 33 specifications with 345

specified operations and more than 500 axioms. The implementation contains more than 2200 lines of code. This large number of axioms is due to the explicit error handling. The use of partial functions in the SPECTRUM version [Het93] does not avoid the axioms but allows a more compact notation using strictness clauses. The size of this translation shows that the use of the semi-formal specification method E/R-diagram can not be replaced by a formal algebraic specification without losing clarity. However we win a verified prototypical implementation.

4 Putting Verification to the Acid Test — Dynamic Hashing

This case study is concerned with the verification of a store structure by dynamic (or extendible) hash tables. These hash tables were introduced in [Lar78]. Dynamic hash tables allow an arbitrary number of elements to be inserted and are used mainly in large file systems with direct access stored on disk. Their main characteristic is that they never need to be reorganized, although collision lists of bounded size are used.

4.1 Dynamic Hashing

We consider hash tables, which store pairs of keys and data. For simplicity keys are natural numbers. Other introductions to hash tables assume that there is an encoding function from keys to natural numbers, or that keys are computed from the data. Both variants are easily convertible to the one presented here. Typical applications of hash tables are file systems with names as keys and contents as data, entities with their key attributes in E/R-diagrams of databases and booking systems.

The key of a pair is used to compute an array index where the pair is stored. Dynamic hashing uses collision lists (as opposed to the open address-technique) to resolve conflicting hash codes. Figure 3 shows a typical dynamic hash table with some elements already inserted.

The upper vector, called 'generation vector', represents the address space. Its length is always a power of two, 2^{maxgen}, where 'maxgen' is called the maximal generation. In the diagram we have maxgen = 2. key-data pairs are stored in the collision lists in the lower half of the diagram. Collision lists have a fixed maximal size 'maxsize' (here maxsize = 5). In a real implementation, maxsize is chosen, such that a collision list will fit into a block (or page) of physical memory. Therefore collision lists are called 'blocks' in the following. Blocks contain both data and their keys, but since data items do not matter here, we omit them in the figures.

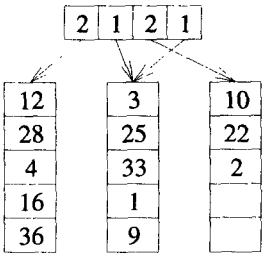


Fig. 3.

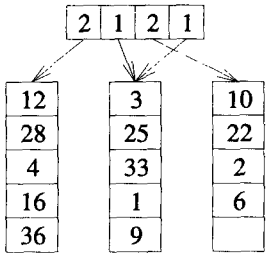


Fig. 4.

To find the right block in which to store a datum with key k, we compute an index j := k mod 2^{maxgen}, and follow the pointer below the j^{th} entry of the generation vector. The indirect block allocation avoids the necessity to use one block for every hash code. In the diagram the keys, which are congruent 1 modulo 4 ($\equiv$ 1 mod 2^2) and $\equiv$ 3 mod 2^2 both are stored in the middle, shared block. The keys in this block are exactly those which are $\equiv$ 1 mod 2^1. The exponent 1 is called the generation of the block and stored in the generation vector.

Inserting a new key calling an INSERT procedure is simple, if it is placed in a block which is not full, e.g. the rightmost block in Fig. 3. Inserting a datum with key 6 in Fig. 3 will result in Fig. 4.

Inserting in a full block requires a case distinction. One case is, when the selected block has generation < maxgen, such as the middle one. In this case the block is split into two blocks with incremented generation and data items are distributed among them, according to their keys. A subprocedure SPLIT is used for this task. Then INSERT is called recursively on the result. Adding 7 divides the entries of the middle block in those $\equiv$ 1 mod 4 and those $\equiv$ 3 mod 4 and gives Fig. 5 as result.

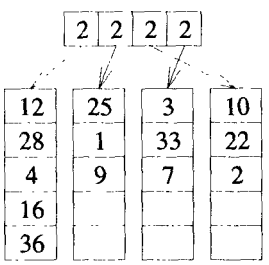


Fig. 5.

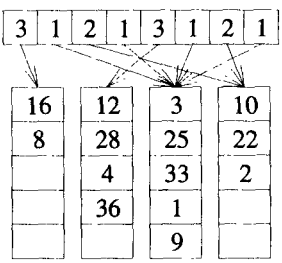


Fig. 6.

The other case is, when the generation of the block has already reached maxgen. Then we double the whole generation vector including the pointers with

a DOUBLE procedure, thereby increasing the maximal generation. Subsequently we split the block under consideration and call the INSERT procedure as in the previous case. As an example, adding 8 in Fig. 3 will lead to Fig. 6. Note that it may be necessary to repeat the process of splitting (and eventually doubling), if all data of the split block including the one to insert, happen to be hashed into the same block.

Deleting a data item proceeds as follows: First, the key-data pair is removed from the appropriate block. If the resulting block is empty and may be eliminated, a 'merge'-operation is called, which is the inverse of splitting two blocks. Note that an empty block cannot always be eliminated. Deleting an empty block is possible, if its 'dual' block was not split repeatedly by insertions, where the two blocks resulting from a split-operation are called dual. If all generations in the generation vector become smaller than the maximal generation, the table is halved.

4.2 The Modular System

Viewed as part of a modular system, the algorithms of dynamic hash tables realize an efficient implementation of a store structure by elementary data types, such as arrays and records. The following specification describes the store structure:

generic specification Store
parameter Data
using Nat
sorts store
functions

emptystore :		→ store
insert	: nat × data × store	→ store
lookup	: nat × store	→ data
delete	: nat × store	→ store

variables s, s_1, s_2: store; d : data
axioms
store **generated by** emptystore, insert;
lookup(m, emptystore) = nodata,
lookup(m, insert(m, d, s)) = d,
$m \neq n \rightarrow$ lookup(m, insert(n, d, s)) = lookup(m, s),
delete(m, s) = insert(m, nodata, s),
$s_1 = s_2 \leftrightarrow (\forall$ m.lookup(m, s_1) = lookup(m, s_2))
end

The specification uses the natural numbers Nat as keys. It is parameterized by the data to be stored. The constant 'nodata' is used to denote failure of the lookup function. It should be different from the regular data to be stored in the table. Stores are built up from emptystore, using the insert-function. The axioms should be self-explanatory, note that the insert-function overwrites old entries with the same key.

The store specification is a typical example for information hiding. Possible implementations include simple association lists as well as complex variants of tree structures (like AVL-trees or B-trees) and of course hash tables, as described here. A software system which uses the store specification as an intermediate level of abstraction may choose freely among these implementations.

The implementation represents stores by dynamic hash tables. The functions emptystore, insert, delete and lookup are implemented by the procedures EMPTYSTORE, INSERT, DELETE and LOOKUP, computing on the hash tables. They comprise 314 lines of code. Correctness of the implementation is assured by suitable verification conditions. The abstract data type 'dynamic hash table' as visualized in Figs. 3 – 6 is formalized as a record (or tuple) of 6 components:

First the generation vector, which is an array of natural numbers. Second the heap, containing the blocks of the hash table. Since we wanted to mimic disk storage, where pages are numbered, we chose to model the heap as an array of blocks. The third component, pointers to the blocks are realized as an array of indices (natural numbers), the 'pointer vector'. The fourth component needed is the maximal generation, and to speed up the decision, if halving of the table is possible, we introduced a counter for the number of entries in the generation vector equal to the maximal generation. To optimize the usage of heap storage, we always allocate the block with the least index, which is not already used. This is supported by the last component of dynamic hash tables, a set of indices of blocks currently used. It is implemented by lists of intervals in a separate module with 107 lines of code. The representation guarantees access in O(1) to the least index not in the set.

Blocks are specified as an abstract data type similar to stores, but with a maximal size maxsize (an unspecified natural number). Inserting a datum with new key in a block of maximal size will leave it unchanged. Their implementation in another module by association lists comprises 139 lines of code. Altogether 30 specifications are built up from basic specifications by enrichment, union, renaming and actualization. Figure 7 gives a simplified version of the development graph: Solid arrows denote the using relation, dashed arrows mean actualization. Serrated arrows denote the implementation relation and boxes with rounded corners contain the algorithms of the module. The specifications on the bottom line are assumed to describe data types which are available on the target computer.

4.3 Verification

The modular system described in the previous section is hierarchy respecting (see [Rei95], this volume). Therefore it is possible to verify each of the three modules independently. Table 3 gives an overview over the verification. At present verification has been done excluding the delete function.

Comparing the verification efforts of the modules, it becomes obvious that the first two modules are rather straightforward, while the main module is a lot of hard work. Several reasons contribute to the complexity of the verification:

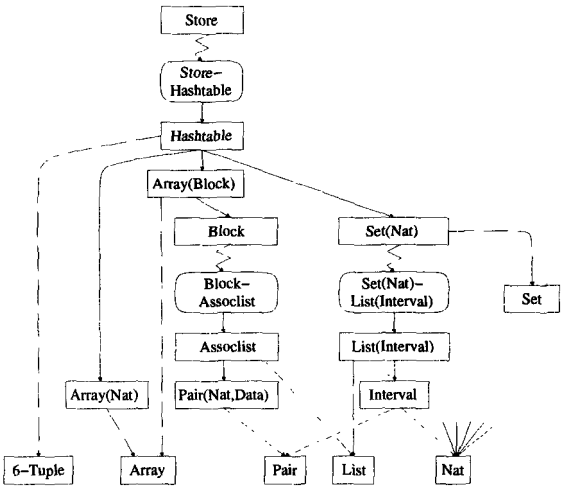


Fig. 7. The simplified development graph

Table 3. Overview over the verification.

Name of module	block-assoclist	set(nat)-list(interval)	store-hash table
Lines of code	139	107	314
Proof obligations	34	17	17
Lemmas	15	9	71
Proof steps	5272	3470	3422
Automation	94.5 %	86.3 %	51.8 %
Absolute time	$\approx$ 3 days	$\approx$ 3 days	$\approx$ 2 months

- As can be seen from the description above, dynamic hash tables involve a lot of different data structures: arrays, blocks, natural numbers, records, lists and sets. This makes simplification of first order formulas rather expensive. Currently over 250 rewrite rules are used for this task. Especially natural numbers with modulo-operation and exponentiation need a lot of lemmas like:

  $$x \bmod 2^{n+1} \neq x \bmod 2^n \rightarrow\ x \bmod 2^{n+1} = (x \bmod 2^n) + 2^n$$

  Order relations, frequently used here (e.g. in inductive proofs), pose an additional problem, since transitivity cannot be handled by rewriting. Altogether, simplification consumes about 80% of the system run time in the proofs and is therefore the crucial point for the efficiency of the proof strategy.
- Arrays lead to formulas involving quantification over their elements. Quantifier instantiation is the main obstacle to reduce the need for interaction in proofs.
- Because of the great number of conditionals in procedure bodies, a lot of cases have to be considered, resulting in large proofs. Since all unnecessary

case distinctions double the effort to prove a goal, they must be avoided as much as possible. This lessens the degree of automatization, since fewer heuristics can be used.
- A lot of lemmas are involved in the correctness proof. Their formulation is difficult and they are often subject to modifications.

These problems indicate that a wide variety of difficulties has to be overcome before verification becomes a manageable task. For this reason we feel justified to speak of this example as an "acid test" for verification.

In the KIV system the first two problems listed above have been subject to substantial improvements, which were incorporated during the work on this case study. Efficiency of simplification was improved by using a shared data structure for simplifier rules. A routine handling transitivity was added. For quantifier instantiation a systematic procedure based on pattern matching and rewriting was developed, which finds about 70 % of the instantiations automatically.

Apart from these improvements, the *proof engineering* facilities implemented in the KIV system were heavily exploited. These allow to view and manipulate proof trees[3] comfortably. As an example, an unnecessary case distinction often become obvious only when the proof of the first case is already done and the second case is considered. It can be undone by nonchronological backtracking, which is realized by simply pruning away the branch from the proof tree. The branch can then be used either to formulate a lemma or to reconstruct the proof by a replay mechanism, which can handle a lot of differences in the goals (here resulting from the omission of the case distinction). Replaying "similar" proofs is also helpful when a lemma has to be corrected.

Still lemma formulation is the main task in verification. To demonstrate the relevant problems, we consider one of the proof obligations. The example involves the restriction formula R, which describes which of the 6-tuples are used as representations of stores. This formula is a conjunction of 2 formulas: $R = C \wedge M$, where C indicates correctness of the hash table (no wrong keys in a block etc.) and M guarantees minimality, i.e. no superfluous empty blocks. The proof obligation states that the INSERT procedure terminates and R is an invariant. As a formula of Dynamic Logic this is written as

$$R(ht) \rightarrow \langle \mathrm{INSERT}(k, d, ht; ht_1) \rangle R(ht_1)$$

Since INSERT is recursive, this goal must be proved by induction. Since we need a criterion for termination and since $R(ht)$ does not hold prior in recursive calls (minimality is violated), the induction hypothesis has to be generalized, i.e formulas R_1, R_2 have to be found such that

$$R_1(ht) \rightarrow \langle \mathrm{INSERT}(k, d, ht; ht_1) \rangle R_2(ht_1) \qquad (1)$$

is provable by induction. Since INSERT calls the subprocedures SPLIT to split blocks and DOUBLE to double the hash table, we also need lemmas

$$R_3(ht) \rightarrow \langle \mathrm{SPLIT}(j, ht; ht_1) \rangle R_4(ht_1) \qquad (2)$$

[3] proof trees are represented as an explicit data structures in the KIV system

$$R_5(ht) \rightarrow \langle \mathrm{DOUBLE}(ht; ht_1) \rangle R_6(ht_1) \tag{3}$$

for these subprocedures and have to choose formulas R_3, R_4, R_5, R_6. Now consider again the two cases for the INSERT procedure from 4.1, where it was necessary to call the two subprocedures. In the first, where the generation of the block to be split is not maximal, $R_1(ht) \rightarrow R_3(ht)$ and $R_4(ht) \rightarrow R_1(ht)$ must hold (the latter because of the recursive call to INSERT after splitting the block). In the second case, where the generation of the block is maximal and DOUBLE is called, $R_1(ht) \rightarrow R_5(ht)$, $R_6(ht) \rightarrow R_3(ht)$ and $R_4(ht) \rightarrow R_1(ht)$ must be provable. Therefore, none of the R_i (i = 1 ...6) can be chosen independently, and to find versions, which make all three lemmas fit together, is a difficult problem. The final version of lemma (2) is

$$\begin{aligned}
&\mathrm{C}(ht) \wedge \mathrm{WM}(k, ht) \\
&\wedge \mathrm{get}(\mathrm{genvec}(ht), k \bmod 2^{\mathrm{maxgen}(ht)}) \neq \mathrm{maxgen}(ht) \\
&\wedge \mathrm{size}(\mathrm{get}(\mathrm{heap}(ht), \mathrm{get}(\mathrm{ptrvec}(ht), k \bmod 2^{\mathrm{maxgen}(ht)}))) = \mathrm{maxsize} \\
&\wedge \mathrm{find}(k, \mathrm{get}(\mathrm{heap}(ht), \mathrm{get}(\mathrm{ptrvec}(ht), k \bmod 2^{\mathrm{maxgen}(ht)}))) = \mathrm{nodata} \\
\rightarrow\ &\langle \mathrm{SPLIT}(k \bmod 2^{\mathrm{maxgen}(ht)}, ht; ht_1) \rangle \\
&\quad (\quad \mathrm{C}(ht_1) \wedge \mathrm{WM}(k, ht_1) \\
&\quad\ \wedge \mathrm{maxgenctr}(ht_1) \geq \mathrm{maxgenctr}(ht) \\
&\quad\ \wedge \mathrm{card}(\mathrm{usedblocks}(ht_1)) = \mathrm{card}(\mathrm{usedblocks}(ht)) + 1)
\end{aligned}$$

where WM(k,ht) is weak minimality excluding the block where key k will be inserted. 'genvec', 'ptrvec', 'heap', 'maxgen', 'maxgenctr' and 'usedblocks' are the selectors on the 6-tuple. 'size' is the number of key-data pairs in a block and 'find' on blocks is the analogue to 'lookup' on stores. 'get(i,a)' selects the i^{th} component from an array a and 'card' is cardinality on sets.

To find and prove versions of all three lemmas which fit together required approximately 3 weeks of work. The lemmas were changed several times in this process, when proof attempts showed their unprovability or their use revealed insufficiency. Here the possibility of replaying proofs and proof attempts was indispensable.

5 Conclusion

In this paper we presented three case studies with the KIV system which showed different aspects of verification. It is an attempt to give at least some answers to the two questions *What is verification good for?* and *What are the prerequisites to be successful?*. Several other case studies carried out with the KIV system (larger, equally sized and smaller ones) yielded comparable results and confirm the experiences made.

Obviously, verification serves to prove the correctness of an implementation with respect to a specification. This is useful for complex algorithms with a high degree of reusability (as in the case of dynamic hashing) or for security relevant programs (as in the case of access control). Furthermore, verification can be used

to prove properties of specifications, and — in special cases — to validate the correct formalization of semiformal methods (the translation of E/R-diagrams gives an example). Experiences have shown that verification typically reveals errors in both programs and specifications. Their correction improves the quality of the specifications and leads finally to correct programs.

A powerful deduction component (including fast simplification and a high degree of automatization), enriched by proof engineering facilities (like the possibilities for manipulating proof trees), must interact with a fine tuned correctness management with special support for modifications and intelligent mechanisms for a reuse of proofs. Their combination, together with a suitable methodology that allows for modular verification and includes evolutionary aspects, makes large-scale verification possible.

References

[ABLP91] M. Abadi, M. Burrows, B. Lampson and G. Plotkin. A Calculus for Access Control in Distributed Systems. In *CRYPTO 91*, Proceedings. 1991.

[Bek92] J. ter Bekke. *Semantic Data Modeling.* PRENTICE HALL, 1992.

[CHL95] F. Cornelius, H. Hußmann and M. Löwe. The KORSO Case Study for Software Engineering with Formal Methods: A Medical Information System. In Broy, Jähnichen (eds.), KORSO, *Correct Software by Formal Methods.* Springer LNCS 1995. (In this volume).

[Fuc94] T. Fuchß. Translating E/R-diagrams into Consistent Database Specifications. Technical Report 2/94, Fakultät für Informatik, Universität Karlsruhe, Germany, 1994.

[Het93] R. Hettler. Übersetzung von E/R-Modellen nach SPECTRUM. Technical Report I9333, TU München, Germany, 1993 (in german).

[HHMNS93] R. Hettler, H. Hußmann, S. Merz, F. Nickel and O. Slotosch. Die funktionale Essenz von HDMS-A. Technical Report I9335, Fakultät für Informatik, TU München, Germany, 1993 (in german).

[HRS90] M. Heisel, W. Reif and W. Stephan. Tactical Theorem Proving in Program Verification. 10th International Conference on Automated Deduction, Kaiserslautern, FRG, Springer LNCS 1990.

[Lar78] P.A. Larson. Dynamic Hashing. BIT 18 (1978).

[PW+95] P. Pepper, M. Wirsing, et al. A Method for the Development of Correct Software. In Broy, Jähnichen (eds.), KORSO, *Correct Software by Formal Methods.* Springer LNCS 1995. (In this volume).

[Rei92] W. Reif. Verification of Large Software Systems. In Shyamasundar (ed.), *Foundations of Software Technology and Theoretical Computer Science*, Proceedings, New Dehli, India. Springer LNCS 1992.

[Rei95] W. Reif. The KIV-Approach to Software Verification. In Broy, Jähnichen (eds.), KORSO, *Correct Software by Formal Methods.* Springer LNCS 1995. (In this volume).

[RS93] W. Reif and K. Stenzel. Reuse of Proofs in Software Verification. In Shyamasundar (ed.), *Foundation of Software Technology and Theoretical Computer Science*, Proceedings, Bombay, India. Springer LNCS 1993.

[Ste93] K. Stenzel. A Verified Access Control Model. Technical Report 26/93, Fakultät für Informatik, Universität Karlsruhe, Germany, 1993.