Deduction in the Verification Support Environment (VSE)

Dieter Hutter, Bruno Langenstein, Claus Sengler,
Jörg H. Siekmann, Werner Stephan, Andreas Wolpers

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH
Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany
⟨*name*⟩@dfki.uni-sb.de

Abstract. The reliability of complex software systems is becoming increasingly important for the technical systems they are embedded in. In order to assure the highest levels of trustworthiness of software formal methods for the development of software are required. The VSE-tool was developed by a consortium of German universities and industry to make a tool available which supports this formal development process.
VSE is based on a particular method for programming in the large. This method is embodied in an administration system to edit and maintain formal developments. A deduction component is integrated into this administration system in order to provide proof support for the formal concepts.
In parallel to the development of the system itself, two large case studies were conducted in close collaboration with an industrial partner. In both cases components of systems previously developed by the industry were re-developed from scratch, starting with a formal specification derived from the original documents.
This paper focuses on the deduction component and its integration. We use a part of one of the industrial case studies in order to illustrate the important aspects of the deduction component: We argue that a close integration which makes the structure of developments visible for the theorem prover is necessary for an efficient treatment of changes and an indispensable structuring of the deduction process itself. Also we commend an architecture for interactive strategic theorem proving which has turned out to be adequate for applications in the context of formal program development. The last one of the three main sections addresses the important point of detecting bugs in implementations and specifications.

1 Introduction

The reliability of complex software systems is becoming increasingly important for the technical systems they are embedded in. Malfunctioning of software systems caused by design flaws or faulty implementations may lead to loss or garbling of data, breach of security, danger to life and limb, and, in almost all cases, severe economic losses.

Led by the *German Information Security Agency* (Bundesamt für Sicherheit in der Informationstechnik, BSI) a catalog of criteria for the evaluation of the security of information technology systems has been developed in Germany [IT-89]. In the assessment of a system's trustworthiness, the *development process* plays a major role. Requirements to the development process break down into aspects of (the formulation of) security and/or safety requirements, the overall structure of the system, and the implementation. The highest levels of quality require, to a varying extent, the use of *formal methods*. The BSI thus decided on the development of a tool to support the use of formal methods during all stages of the design process. Starting in 1991 the VSE system [BCC+92] was developed by a consortium consisting of Dornier/DASA, Friedrichshafen, the German Research Center for Artificial Intelligence (DFKI), Saarbrücken, the Gesellschaft für Prozeßrechner-Programmierung (GPP), München, the University of Karlsruhe, and the University of Ulm.

In August 1994 the first prototype was delivered, tested, and finally accepted by the BSI.

VSE is based on a particular method for *programming in the large*. This method is embodied in an *administration system* to edit and maintain formal developments. A *deduction component* is integrated into this administration system in order to provide proof support for the formal concepts.

This paper focuses on the deduction component and its integration. A part of an industrial case study called PERSEUS is used to illustrate the main ideas. The case study is part of an access control system for nuclear power plants. The task was to guarantee that only authorized staff is present in each area of the plant. The part we will be looking at is concerned with the manipulation of the rights of persons to enter different areas of the plant.

The paper is organized as follows. In the following section we give a survey of the VSE system and its underlying formal concepts. In section three we give a more detailed description of the part of the case study which is used in the following. We then discuss the integration of the deduction component into the administration system. We argue that a close integration which makes the structure of developments visible for the theorem prover is necessary for an efficient treatment of changes and an indispensable structuring of the deduction process itself. The next section describes what could be called interactive strategic theorem proving. This architecture has turned out to be adequate for applications in the context of formal program development. The last one of the three main sections addresses the important point of detecting bugs in implementations and specifications.

2 The VSE System

We begin with a survey of the VSE system introducing some underlying formal notions as well as the basic constituents of the systems in its technical realization. We will also mention the two industrial case studies that were carried out in the project.

2.1 The General Method

VSE is based on a method for the top-down development of *structured, formal specifications* and their *stepwise implementation* (refinement) using abstract intermediate layers represented by specifications.

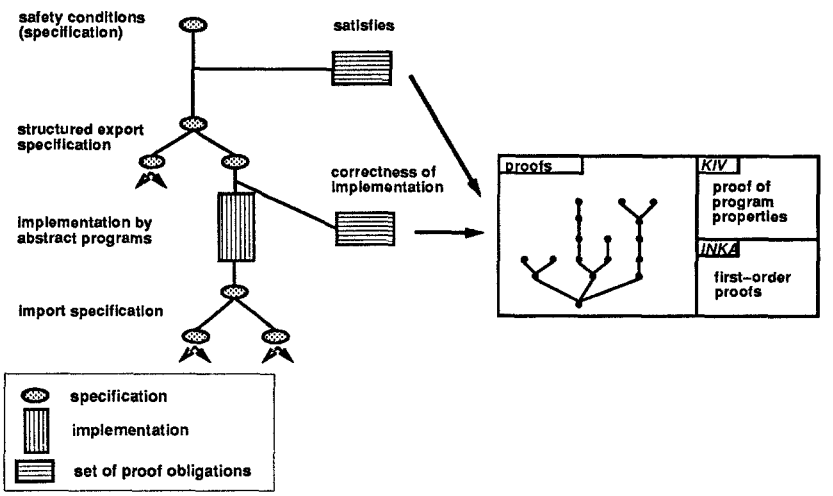

Fig. 1. The VSE Method

Refinement steps are specified by means of *abstract programs* that use concepts from the lower (import) level in order to implement the more abstract ones on the export level. The bottom layer is given by a collection of predefined concepts that can directly be realized in a target programming language. *Modularity* in this context means that sub-specifications can be implemented separately.

At each level, additional safety and/or security requirements can be formulated in addition to the system specification. These requirements are formalized in a separate specification which can be mapped onto the systems specification, such a connection is called a *satisfies*-link between the two.

Most of the development steps lead to so-called *proof obligations*. These formalized assertions are handed over to the deduction component where the actual verification takes place.

Figure 1 shows the general method for formal software development in VSE.

The concrete instance of this general method depends on the formal concepts used for modeling the desired system. Specification concepts covered in the current (first) version of the VSE system include *abstract data types* and *state transition systems*.

Abstract Data Types Abstract Data Types (ADTs) provide a view on a system as an *algebra*, given by a (typed) collection of sets of data objects and a

(typed) collection of operations which manipulate these data.

Elementary specifications (of algebras), so-called *theories* are made up of a *signature* and a set of *axioms*. The signature part introduces the vocabulary of the data type. It consists of a set of types and a set of typed function and predicate symbols. VSE-SL allows full first-order logic to describe the semantics of these symbols in the axiom part.

In general there will be many (non-isomorphic) algebras that satisfy a given specification. VSE-SL allows to restrict the class of models to (term) *generated* or even to *freely generated* models. These restrictions lead to *induction principles* that are used in the verification process.

An example of a theory is given below:

```
THEORY Rights
 USING : Topology;
         Persons
 TYPES : AccessRel =
         GENERATED BY emptyRel |
                      grant(Person,Area,AccessRel) |
                      refuse(Person,Area,AccessRel)
 PREDICATES : hasAccess : Person,Area,AccessRel;
              inRel : Person,Area,AccessRel
 VARS : p, p0, p1 : Person;
        b, b0, b1 : Area;
        m : AccessRel
 AXIOMS : NOT inRel(p0, b0, emptyRel);
          inRel(p0,b0,grant(p0, b0, m));
          NOT inRel(p0, b0, refuse(p0, b0, m));
          b0 /= b1 OR p0 /= p1
          -> (inRel(p0, b0, m) <->
              inRel(p0, b0, grant(p1, b1, m)));
          b0 /= b1 OR p0 /= p1
          -> (inRel(p0, b0, m) <->
              inRel(p0, b0, refuse(p1, b1, m)));
          hasAccess(p, b, m) <->
          (b = extern
           OR visitor(p) AND inRel(leader(p), b, m)
           OR NOT visitor (p) AND
              (inRel(p, b, m) OR hasAccess(p, next(b), m)))
THEORYEND
```

The theory `Rights` corresponds to representation of the rights of persons to enter different areas of the plant. It refers to the (sub-) theories `Topology` (e.g. using the function `next`) which represents the topology of the plant and `Persons` (e.g. using the predicate `visitor`) modelling the different (types of) persons like staff members or visitors. Among other operations this kind of *enrichment* gives rise to a (horizontal) structure of abstract data type specifications.

In the refinement process functions and predicates from the export specification are implemented by (recursive) procedures that use function and predicate symbols from a given import specification. The relation between concepts from the export specification and parts of the implementation is given by a *mapping*. The theory underlying these concepts has been described in [Rei92b], [Rei92a].

Specification of State Transition Systems State transition systems are used to describe systems where the history of operations executed has an effect on the results produced. Here the operations cause *side effects* on a global system *state*.

State transition systems in VSE-SL are given by so-called *objects*. As an example, we present below a simple specification of a state transition system which is, like the theory presented in the last section, taken from the case study PERSEUS conducted in the VSE project.

```
OBJECT PlantAccess
 USING : Rights
 DATA : Matrix : AccessRel
 VARS : p, q : person;
        b : area
 OPERATIONS :
 PROC access(p : Person,b : Area) : bool
      ENSURES IF hasAccess (p, b, Matrix)
              THEN RESULT = T
              ELSE RESULT = F
              FI
 PROC setaccess(p : Person,b : Area,z : bool)
      MODIFIES Matrix
      ENSURES IF z = T
              THEN Matrix = grant(p, b, Matrix')
              ELSE Matrix = refuse(p, b, Matrix')
              FI
 INITIAL : leader(p) = leader(q) ->
           (hasAccess(p, b, Matrix) <->
            hasAccess (q, b, Matrix))
OBJECTEND
```

The *state space* is given implicitly through the items in the DATA-slot. The variables declared here are state-dependent and may be changed by the operations. Their types (sorts) are taken from the *theory* mentioned in the USING-slot. Additional local variables may be declared in a VARS-slot. The behavior of *Operations* is described by *pre- and postconditions*. In the postcondition a variable x' denotes the value of x prior to the execution of an operation. The preconditions form a complete case-distinction. In the RESULT-slot the value returned by an operation is specified. By an additional REQUIRES-slot one might restrict the situations in which the operations may be executed. These requirements have to

be respected if an operation is used for example as an import operation in some implementation.

Consider the example of the procedure `setaccess`. `Matrix'` denotes the value of `Matrix` prior to the execution of the operation `setaccess`. Thus, in case the operation is called with `z` equal to `T` the new value of `Matrix` is computed from its current value by applying `grant` with additional parameters `p` and `b` on it.

The set of *initial* states (and perhaps also a global *invariant* of the system) is specified by first-order formulae. As with ADTs there are also *generic* state transition systems, and *unions* of such systems. Export operations of state transition systems are *implemented* in the VSE system by pieces of abstract programs that use operations from some import specification.

An introduction to state transition systems of this kind can be found in [RvHO91] while a comprehensive description of the syntax and semantics of state transition systems is given in [VSE94].

2.2 System Support

VSE is an integrated system that supports the user at all stages of the development process following the method outlined above. Basically there are two kinds of activities, *editing* and *proving*, that are in fact interleaved. This means that the user might edit (some part of) a specification, prove certain safety properties, edit an import specification and a refinement, prove the refinement correct, and then continue his work for example by a further refinement step.

The work is organized via a so-called *development graph* that displays the representation of a (partial) development to the user and allows him to continue his work at a node he wants to expand. The development graph also gives the user access to *status information* that is maintained by the administration system. This *correctness management* controls the work in the various development units and their proof obligations. It keeps track of dependencies between a proof and those parts of the specification which are used during the proof. Changing these parts of the specification will invalidate the related proofs while other proofs may not be affected.

The main system support w.r.t. editing specifications and implementations is *syntactical analysis* including type-checking. If a development step is completely edited and successfully checked, a *logical database* related to this step is created by the system. It is initialized by the axioms computed from the specifications involved and the proof obligations that correspond to the development step. The verification process then takes place in the context of this database where all kinds of logical information, like axioms, lemmata, and proofs are stored. When a branch of the development is completed, the actual code of the target language is generated by the system.

2.3 Case Studies

In parallel to the development of the system itself, two large case studies were conducted in close collaboration with Dornier. In both cases components of sys-

tems previously developed by Dornier were *re-developed* from scratch, starting with a formal specification derived from the original documents. The selection of the case studies was oriented along the specification paradigms supported by the VSE system.

The first case study deals with a system that controls the exchange of programs between radio stations, including the booking of leased communication lines. Within this case study, the kernel of the booking system was re-developed. It contains a complicated algorithm to re-schedule already booked transmissions in the case of conflicts. The booking system was modeled as an abstract data type, i.e. the booking system can be viewed as a structured collection of operations. The entire case study consists of about 5000 lines of specifications (including the security model), and 8000 lines of implemented source code.

The second case study named PERSEUS demonstrates the use of state transition systems. It is (part of) an *access control system* for nuclear power plants. Again the safety relevant kernel of the system, which supports the tracking and control of movements within a plant, was re-developed. This case study, which begun later, comprises currently about 3000 lines of specifications, which were partially implemented in 2000 lines of code.

2.4 Deduction

A distinguishing feature of the VSE system is that it offers deductive support for all formal concepts. This takes into account the experience that one of the main limiting factors for the application of formal methods in an industrial context is machine assistance for the construction of proofs.

Apart from *actualizations of generic* specifications, there are two major kinds of links in development graphs that lead to *proof obligations*, *satisfies-links* between systems specifications and additional *safety and/or security properties* and *refinement steps*. Safety and/or security properties lead to assertions in first-order predicate logic while in the case of refinements we have to prove assertions about programs. These are formulated in a variant of Dynamic Logic (DL), see [Pra76].

The deduction component is made up of two closely integrated provers: the KIV system (Karlsruhe Interactive Verifier) and the INKA system (INduction-prover KArlsruhe), see [HRS90] and [BHHW86]. KIV is mainly used for proofs in DL while INKA provides strategies for first-order assertions, in particular for inductive proofs. Both systems follow the paradigm of *interactive, strategic theorem proving*.

3 PERSEUS

As mentioned before we will use a part of the second case study PERSEUS to demonstrate the main features of the deductive component and its integration into the administration system. While in the first case study the safety requirements were concerned with the problem of exclusive use on lines, here the task

was to guarantee that only authorized staff is present in each area of a nuclear power plant. A small part of the development graph concerning the access rights is given in Figure 2.

The part we will be looking at is concerned with the manipulation of an *access matrix* that encodes the rights of persons to enter different areas of the plant. Persons are either member of the staff or parts of visitor groups who are always escorted by a member of the staff (the so-called leader of a group). Visitors have same access rights to the areas as their leader (but may only enter an area together with their leader). In order to simplify matters staff members are always their own leader regardless whether they escort a visitor group or not.

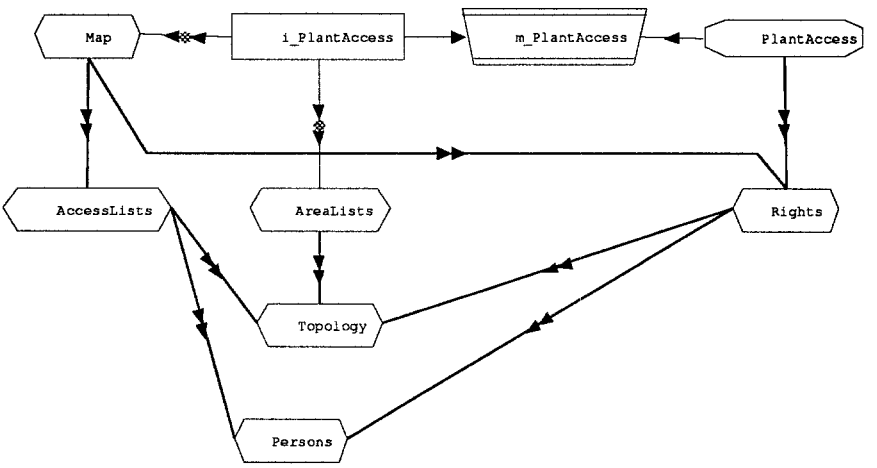

Fig. 2. A Development Graph

On the top-level we have the specification of an object `PlantAccess` which provides operations for looking up (`access`) and changing entries (`setaccess`) in the access matrix (`Matrix`). It uses *abstract data types* for matrixes (`AccessRel`), persons (`person`), and areas (`area`) given by the structured *theory* `Rights`.

In a typical refinement step we re-specify the theory `PlantAccess` on a lower specification level making some implementation decisions. For instance, we implement the access relation by a property list and define procedures `Get_access` and `Set_access` which operate on property lists and correspond to the functions `access` and setaccess specified in the theory `Rights`.

Next, we implement the access matrix in the theory `AccessLists`. A datatype `AccessList` is specified as a property list with persons as items and areas as values. An entry of a person p with value a denotes that p has access to a. Persons may occur several times in the property list specifying access rights to different areas.

```
THEORY AccessLists
 USING : Persons;
         Topology
 TYPES : AccessList = FREELY GENERATED BY
          nil |
          blst(getPerson : Person,getArea : area,rest : AccessList)
 FUNCTIONS : DEL : Person,Area,AccessList -> AccessList
 PREDICATES : in_list : Person,Area,AccessList
 VARS : b : area; p : person; l : AccessList
 ALGORITHMS : DEFPRED in_list(p, b, l) <->
              SWITCH l IN
              CASE nil : FALSE
              CASE blst : IF getPerson(l) = p
                          THEN IF getArea(l) = b
                               THEN TRUE
                               ELSE in_list(p, b, rest (l))
                               FI
                          ELSE in_list(p, b, rest(l))
                          FI
              NI;
 ...
THEORYEND
```

List of areas are defined in the theory `AreaLists`. Especially, the theory provides a function `higherAreas` of an area `a` which provides a list of all areas with higher security level than `a`.

```
THEORY AreaLists
 USING : Topology
 TYPES : areaList =
         FREELY GENERATED BY no_area |
                             add_area(first:area, rest:areaList)
 FUNCTIONS : higherAreas : area -> areaList
 PREDICATES : elem : area, areaList;
              _<_ : areaList, areaList
 VARS : b, b1, b2 : area;
        l, l1 : areaList
 ALGORITHMS : DEFPRED elem(b, l) <->
              SWITCH l IN
               CASE no_area  : FALSE
               CASE add_area : IF first(l) = b THEN TRUE
                               ELSE elem(b, rest(l)) FI
               NI;
              DEFPRED l < l1 <->
              SWITCH l IN
               CASE no_area  : l1 /= no_area
```

```
            CASE add_area : IF l1 = no_area THEN FALSE
                            ELSE rest(l) < rest(l1) FI
          NI
 AXIOMS : b2 elem higherAreas(b1) <-> (b2 > b1 OR b2 = b1)
THEORYEND
```

Based on the theories **AccessLists** and **AreaLists** the procedures **Get_Access** and **Set_Access** are defined which implement the functions **access** and **setaccess**. Thus, **Get_Access** implements the lookup whether a person is allowed to enter a specific area. It takes into consideration the access rights of visitors and an additional requirement that persons who have access to a specific area have also access to less secure areas. **extern** denotes the area outside the plant which is accessible by all persons.

```
FUNCTION Get_Access
 PARAMS : p_person : IN Person;
          p_area : IN Area
 RESULT : BOOL
 BODY   : DECLARE
          v_res: Bool:=F;
         IF p_area = extern THEN v_res := T
         ELSE IF visitor(p_person)
              THEN IF AccessToArea(leader(p_person),
                                   higherAreas(p_area)) = t
                   THEN v_res := T ELSE v_res := F FI
              ELSE IF AccessToArea(p_person,
                                   higherAreas(p_area)) = t
                   THEN v_res := T ELSE v_res := F FI
              FI
         FI;
         RETURN v_res
FUNCTIONEND
```

The procedure **Set_Access** grants or refuses (depending on the value of **p_entry**) a person access to a specific area.

```
PROCEDURE Set_Access
 PARAMS : p_person : IN person;
          p_area : IN area;
          p_entry : IN BOOL
 BODY : IF p_entry = t
        THEN IF NOT in_list(p_person, p_area, List)
             THEN List := blst(p_person, p_area, List)
             FI
        ELSE DelAccessToAreas(p_person, higherAreas(p_area))
        FI
PROCEDUREEND
```

In a next step we have to link both specifications, the theory PlantAccess and its implementation. This is done by so-called *Refinement steps* as it is shown in Figure 2. It consist of a *mapping* and a *module*. Modules, as the module i_PlantAccess below, contain slots for the import specifications, e.g. Map or AreaLists, the state dependent variables, e.g. list, and (references to) procedures that implement the abstract operations, e.g. Get_Access or Set_Access.

```
MODULE i_PlantAccess
 IMPORTSPEC : Map;AreaLists
 DATA : list : AccessList
 ELEMENTS : Get_Access, Set_Access, AccessToArea, DelAccessToAreas
MODULEEND
```

The theory Map provides the relation between the access relations and its implementation as a property list. Based on AccessLists the algorithm list2rel maps each access list into an access matrix:

```
THEORY Map
 USING : AccessLists;
         Rights
 FUNCTIONS : list2rel : AccessList -> AccessRel
 VARS : l : AccessList
 ALGORITHMS :
   DEFFUNC list2rel(l) =
           IF l = nil
           THEN emptyRel
           ELSE grant (getPerson(l), getArea(l),
                                     list2rel(rest(l)))
           FI
THEORYEND
```

The actual correspondence between the above procedures and the operations of the specification in PlantAccess is given by the mapping m_PlantAccess. In our case the access matrix is implemented by an access list while the operations access and setaccess are implemented by the procedures Get_Access and Set_Access.

```
MAPPING m_PlantAccess
 EXPORTSPEC : PlantAccess
 IMPLEMENTATION  : i_PlantAccess
 MAPS : Get_Access IMPLEMENTS access;
        Set_Access IMPLEMENTS setaccess;
        list2rel(List) IMPLEMENTS matrix
MAPPINGEND
```

4 Structural Deduction

Applying formal methods to industrial case studies results in thousands of lines of specifications which represent the logic database with the help of which various proof obligations have to be established. As mentioned above this specification is structured within a development graph (e.g. refer Figure 2 showing the development-graph of `PlantAccess`) and thus, the development graph represents also an structuring of the axiomatization. In Figure 2 doubled arrows indicate that the target theory of an arrow is (virtually) part of the specification of the source theory. This so-called theory-graph (the subgraph of the development graph wrt. double arrows) specifies which axiomatizations are part within a theory. E.g. `PlantAccess` uses the specification of `Rights` and thus, the axioms of `Rights` are visible within the theory of `PlantAccess`.

Proof obligations are always located at elements - e.g. mappings - of the development graph. They have to be established wrt. the specifications of theories accessible from this element via the arrows of the theory-graph. In case of `m_PlantAccess` the proof of some obligation may use the specifications of `i_PlantAccess`, `PlantAccess`, and the sub-theories visible in both. Hence, besides being a notion for structured specification the development graph represents also a structured database for the deduction system. Depending on the location of the proof obligation the corresponding subgraph of the development graph specify the axiomatization to be used for the proof.

Also each theory contains a set of lemmata formalizing properties on functions or operations which have been proven within the denoted subgraph. E.g. `Rights` may contain lemmata proven with the help of the theories `Rights`, `Topology`, and `Persons`.

4.1 Correctness Management

Verifying proof obligations with respect to the actual theories ensures the soundness of the corresponding development step. Since in general specifications contain errors which are usually detected during establishing the proof obligations there is an interleaved process of adapting the specification and proving theorems. In general changing the specification would invalid all proofs but with the help of the development graph the impacts of changing the specification can be restricted to specific parts of the graph while the soundness of proofs in other parts remains unaffected.

In general changing the axiomatization of a theory $\mathcal{T}$ will invalid the proofs (and thus also the lemmata) of all theories using T while underlying theories used by $\mathcal{T}$ are not affected. This work can be done without knowing a single proof just by a static analysis of the development graph. A more elaborated correctness management can be achieved by a dynamic analysis if we save for each lemma and proof obligation the set of axioms and lemmata used during the proof and build up a more elaborated dependency graph of lemmata and proof obligations.

Changing the implementation of a procedure (e.g. `Get_Access` in module `i_PlantAccess`) has no impacts to other parts of the theory-graph (e.g. the theories `PlantAccess` or `Map`) but will invalid lemmata in the implementation module `i_PlantAccess` and cause new proof obligations in the mapping `m_PlantAccess`. In order to reduce the amount of new proof obligations which occur after changing a procedure we use the implementation dependencies of the procedures in order to determine the subset of proof obligations which are affected by the change of the procedure.

4.2 Guiding Proofs

The first benefit of development graphs to the guidance of proofs is the reduction of search space by reducing the number of axioms and lemmata available during the proof. According to the theory-graph only a subset of the overall specification is available while establishing a proof obligation attached to an element of the graph.

Second, the development graph defines a partial ordering on theories and thus, also a partial ordering on functions and predicates defined within these theories which is used as a skeleton for a simplification ordering on formulas. E.g. `hasAccess` is defined within the theory `Rights` while `visitor` is specified within `Persons`. Thus, the system will orient appropriate equivalences as rewrite rules in order to replace occurrences of `hasAccess` in favor of `visitor` whenever possible.

Besides the explicit structure given by the development graph each axiomatization of a theory is structured again by itself. The axiomatization is build up by type declarations, specification of algorithms or procedures, arbitrary first-order formulas etc. Each of these types of specification result in a set of formulas which are part of the theory axiom. They differ in the way the deduction system uses them in order to guide a proof. E.g. algorithms (which are proved to be terminating) and type declarations are used to create induction schemes. Additionally, algorithms are used to refine the simplification ordering and to generate new simplification rules in order to enable symbolic evaluation.

Consider the freely generated datatype `areaList` inside the theory `AreaLists`. Its definition introduces a new (structural) induction scheme:

$$\Phi(no\_area) \wedge \forall x : areaList\ x \neq no\_area \rightarrow \Phi(rest(x)) \rightarrow \Phi(x)$$

$$\rightarrow \forall x : areaList\ \Phi(x)$$

Also the definitions of `elem` and `<` of the theory `AreaLists` which are proved to be terminating suggest (using their recursion orderings) induction schemes which are (in these cases) identical to the previous structural induction scheme. The algorithm of `list2rel` creates the simplification rules

$$list2rel(nil) \rightarrow emptyRel \text{ and}$$

$$list2rel(blst(p, a, tl)) \rightarrow grant(p, a, list2rel(tl))$$

5 Automation

Passive proof assistance where the system only *controls* the user (proof-checking) is not sufficient in situations where we are faced with proofs of thousands of deduction steps. On the other hand conventional fully automated theorem proving systems that carry out an exhaustive search following some complete (and therefore problem independent) strategy also turned out to be inadequate for our kind of applications. One reason for this is the limited possibility for user interaction where the activities of the user are restricted to certain preparations (formatting the input, choosing the search strategy and additional parameters) *before* the system is run.

Proof construction in VSE therefore follows the paradigm of *interactive strategic theorem proving*. A suitable representation of problem-specific knowledge which often is available in this context enables the system to exhibit an *active behavior*. It is this way that a routine generation of large and often technically complex proofs becomes possible. The main prerequisite for a strategic user interaction is an architecture that allows the user (and of course also the system) to analyze the *state of a proof*. In our architecture a proof state is given by a *partial proof tree*. The leaves of such a tree are open subgoals, lemmata or axioms. The basic activity is to refine a given subgoal by applying (backwards) proof steps thereby generating new subgoals. These proof steps are given by the general formalism and by local axioms. The user may also give more precise hints for a proof plan, which includes directives to apply a certain rule as the next step if the system gets stuck or he may replace hints. *Backtracking*, initialized either by the user or the system itself, allows proof steps to be undone and a previous situation to be restored. Thus, at each stage of the proof synthesis, the human user can revise the proof attempt specified so far or to give advice how to fill in the gaps of the proof tree.

Specifying, for instance, the mapping `m_PlantAccess` the system comes up with a proof obligation

$$\langle \texttt{Get\_Access(p, b, res)} \rangle \mathit{true}$$

which states that the procedure `Get_Access` terminates on all inputs p, b, and res. `Get_Access` uses a procedure `AccessToArea` which is defined as follows:

```
FUNCTION AccessToArea
 PARAMS : p_person : IN person;
          p_areaList : IN areaList
 RESULT : bool
 BODY : DECLARE
        v_res2: bool:=f;
        WHILE (NOT p_areaList = no_area) AND v_res2 = f DO
           IF in_list(p_person, first(p_areaList), List)
           THEN v_res2 := t
           ELSE p_areaList := rest(p_areaList) FI
        OD;
        RETURN v_res2
FUNCTIONEND
```

The deduction system reduces the proof obligation and comes up with a partial proof tree which contains a single open goal concerning the termination of the *while*-loop in `AccessToArea` (called by `Get_Access`). Now, it is up to the user to select an appropriate induction rule. After that the system continues and finish the proof automatically. During the proof several first order subgoals have to be proven like for instance,

$$\forall x : areaList\, x \neq no\_area \rightarrow rest(x) < x$$

This goal is proven automatically by induction. The system uses the knowledge of the termination ordering of $<$ in the theory `AreaLists` in order to synthesize an appropriate induction scheme which results in this case in a structural induction. In case of

$$rest(x) = add\_area(first(rest(x)), rest(rest(x)))$$

we obtain an induction step. The system analyses the syntactical differences between the induction hypothesis

$$\underline{rest(x)} \neq no\_area \rightarrow rest(\underline{rest(x)}) < \underline{rest(x)}$$

and the induction conclusion

$$x \neq no\_area \rightarrow rest(x) < x.$$

These differences (which are underlined above) guide the prover in order to enable the use of the hypothesis [Hut90]. In this case the prover uses an axiom

$$\begin{aligned}\forall u, v : areaList\, u = add\_area(first(u), rest(u)) \wedge v = add\_area(first(v), rest(v)) \\ \rightarrow (u < v) \leftrightarrow (rest(u) < rest(v))\end{aligned}$$

derived from the algorithm defined in the theory `AreaLists` to minimize the syntactical differences and we obtain:

$$x \neq no\_area \rightarrow rest(rest(x)) < rest(x).$$

Using the above precondition of the induction step the proof of the induction step is finished.

6 Fixing Bugs

So far, we have looked at the question of how to create proofs for *correct* theorems. In general, however, we will be confronted with errors in specifications, implementations, or both. Unless we have introduced "corresponding" errors in specifications and implementations, such errors will result in proof obligations which are not provable, i.e. we will reach open premises in the proof tree which can't be closed. As opposed to e.g. resolution based proofs, one can derive information about the potential problem by investigating the open premise and the path from the root of the proof tree to that premise.

As an example, consider the proof obligation for the `Set_Access` operation:

$$l = l' \vdash \langle \texttt{Set\_Access}(p, b, z) \rangle (z = t \rightarrow \mathit{list2rel}(l) = \mathit{grant}(p, b, \mathit{list2rel}(l'))) \\ \wedge (z \neq t \rightarrow \mathit{list2rel}(l) = \mathit{refuse}(p, b, \mathit{list2rel}(l'))) \ ,$$

As mentioned before, *list2rel* maps access lists to the access matrix. This asserts that `Set_Access` will terminate, and afterwards the new value of the access relation, $\mathit{list2rel}(l)$, will correspond to the old value, $\mathit{list2rel}(l')$, with p granted or refused access to b according to the flag z.

While working on the proof, we will encounter an open goal

$$\mathit{in\_list}(p, a, l) \vdash \mathit{list2rel}(l) = \mathit{grant}(p, a, \mathit{list2rel}(l)) \ ,$$

since the implementation will add an entry only if it is not yet there (i.e. if $\neg \mathit{in\_list}(p, a, l)$ is true), while the specification does not treat that case separately. The goal is true if granting p access to a a second time (since if it is already in l, it will be in $\mathit{list2rel}(l)$ as well) has no effect. To prove this goal, we need axioms for the equality on relations. A quick look at the theory `Rights` reveals that such axioms have been forgotten.

Here, a sophisticated correctness management pays off. If we add the missing axioms to the theory `Rights`, the validity of the proofs for `m_PlantAccess` is not affected (since we only enlarge the axiom base, and our logic is monotonic). The change does, however, influence the correctness of an implementation of `Rights`. As a whole, it becomes invalid. The actual consequence of the change is a larger set of proof obligations, so that the existing proofs can remain valid. If we have to change the implementation as well to satisfy the additional proof obligations, even (some of) the proofs have to become invalid.

In general, an unprovable premise like the one above may hint at an error in any of the involved procedures or specifications, and a careful analysis is needed to find out the true reason for the problem.

A large number of bugs were uncovered during the verification process, in specifications as well as in implementations. In both case studies, bugs in specifications were more common, and were discovered when proving the security model or the correctness of an implementation. Also, aspects which were only vaguely described in the original documents and also difficult to formalize, proved to be those which significantly delayed the original projects. For the PERSEUS case study, for example, this included the treatment of visitor groups, the reaction to messages which are inconsistent with the current system state (e.g. staff members being detected in areas to which they have no access), and combinations of these problems (e.g. members of a visitor group who 'got lost'). The formal treatment of these parts uncovered intricate problems which had been overlooked in the original design.

7 Related Work, Current Research

In order to compare VSE to other methods and systems we recall the most important features of the system. The VSE methodology combines data type

specifications with an abstract machine notation. Abstract machines are based on a notion of persistent states. The current version of VSE allows to specify operations causing state transitions as shown in section 2.1 . In that aspect VSE is similar to Z [Spi92] and systems like EHDM [RvHO91], VDM [Jon90], and the B Tool [ALN+91].

The methodology for developing abstract machines in B corresponds to that for state based systems in VSE . In VSE there is, however, a distinction between the problem of proving the *implementation* of a state based system correct, for which dynamic logic proof obligations are generated, and proving that one specification of a state based systems entails another. The latter is reduced to predicate logic proof obligations, and used to verify safety conditions and/or security models developed separately from a specification.

Reasoning about state based systems in the current version of VSE is restricted to invariance properties (given by a separate safety/security model) and the proof of refinements. However, the approach is compatible for example with Lamport's TLA, [Lam94]. Current work is concerned with an embedding of the specification technique into a temporal framework in order to cover for example eventuality properties and provide more general refinement concepts.

The data type part of VSE is based on abstract data types as compared to model based techniques like Z and VDM. Here and also in the case of abstract machines VSE supports algorithmic elements. In particular this concerns recursive definitions in constructive specifications and refinements. The correctness proof for an implementation, which usually involves assertions about the interplay between recursive procedures, also guarantees the consistency of the implemented data types relative to the consistency of the data types used for the implementation.

The algorithmic constructs allow for bridging the gap between concepts in abstract specifications and the concepts available in real programming languages. Proof support heavily relies on techniques for proving program properties and for guiding inductive proofs. In these aspects there is a different emphasis compared to Z-based systems.

From a more technical point of view considerable effort was made with respect to the integration of different proof techniques and the integration of the entire deduction component into a system for editing and storing formal developments.

Powerful proof support relies on special tailor-made calculi and an adequate representation of domain specific knowledge. The deduction component of the VSE system offers extensive proof support beyond the level of experimental or ad hoc systems. Up to now it is a heterogeneous system in that it combines two separated theorem proving systems. In the next version the various strategies will share a common deductive mechanism but still use different additional structures that allow for an efficient and highly automated proof generation.

VSE differs from systems with a loose integration of the deduction component or stands-alone deduction systems, like HOL, [GM93], or the Boyer-Moore prover, [BM79], not only by an automatic generation of proof obligations but also by offering a correctness management that maintains a consistent state

of the development including the databases of logical objects. This correctness management is indispensable for an efficient treatment of changes. [BTo94]

With respect to both quantitative as well as qualitative aspects VSE is adequate for many industrial applications. However, this does not mean that no improvements are necessary. The VSE consortium is currently working on an enhanced version of the system to take notions like object orientation, concurrency, and real time into account.

Research at the DFKI is concerned with the extension of the formal basis of VSE and a further development of the deduction component. State transition systems as used in VSE are extended to cover *concurrent* and *embedded* systems based on an interleaving semantics. Temporal logics will be used in order to reason about these execution sequences.

With respect to the deduction component, we are working on a closer integration of KIV and INKA and a framework for proof planning which provides generic concepts and a more explicit representation of proof plans.

References

[ALN+91] J.R. Abrial, M.K.O. Lee, D.S. Neilson, P.N. Scharbach, and I.H. Sorensen. The B-method (software development). In W.J. Prehn, S.; Toetenel, editor, *VDM 91. Formal Software Development Methods. 4th International Symposium of VDM Europe Proceedings.*, volume 2, pages 398–405. BP Res., Sunbury Res. Centre, Sunbury-on-Thames, UK, Springer-Verlag, Berlin, Germany, October 1991.

[BCC+92] P. Baur, E. Canver, J. Cleve, R. Drexler, R. Förster, P. Göhner, H. Hauff, D. Hutter, P. Kejwal, D. Loevenich, W. Reif, C. Sengler, W. Stephan, M. Ullmann, and A. Wolpers. The Verification Support Environment VSE. In *Safety of Computer Control Systems 1992 (SAFECOMP'92)*, 1992.

[BHHW86] Susanne Biundo, Birgit Hummel, Dieter Hutter, and Christoph Walther. The Karlsruhe Induction Theorem Proving System. In Jörg H. Siekmann, editor, *Proceedings 8^{th} International Conference on Automated Deduction (CADE), Lecture Notes in Computer Science (LNCS) 230*, pages 672 – 674, Oxford, England, 1986. Springer-Verlag, Berlin, Germany.

[BM79] R. S. Boyer and J Strother Moore. *A Computational Logic.* Academic Press, London, England, 1979.

[BTo94] The B-Toolkit. B-Core(UK) Limited, October 1994.

[GM93] M.J.C. Gordon and T.F. Melham. *Introduction to HOL.* Cambridge University Press, 1993.

[HRS90] Maritta Heisel, Wolfgang Reif, and Werner Stephan. Tactical Theorem Proving in Program Verification. In *Proceedings of the 10^{th} International Conference on Automated Deduction*, volume 449 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 115–131. Springer-Verlag, Berlin, Germany, 1990.

[Hut90] Dieter Hutter. Guiding induction proofs. In Mark E. Stickel, editor, *Proceedings 10^{th} International Conference on Automated Deduction (CADE), Lecture Notes in Artificial Intelligence (LNAI) 449*, pages 147–161, Kaiserslautern, Germany, July 1990. Springer-Verlag, Berlin, Germany.

[IT-89] IT-Sicherheitskriterien. Bundesanzeiger, 1989.

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

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3), 1994.

[Pra76] V. R. Pratt. Semantical Considerations on Floyd-Hoare Logic. In *Proc. 17th IEEE Symp. on Foundations of Computer Science*, pages 109–121, October 1976.

[Rei92a] Wolfgang Reif. Correctness of Generic Modules. In Nerode and Taitslin, editors, *Symposium on Software Technology and Theoretical Computer Science*, volume 620 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, Berlin, Germany, 1992. Tver, Russia.

[Rei92b] Wolfgang Reif. Verification of Large Software Systems. In Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 652 of *LNCS*. Springer-Verlag, Berlin, Germany, 1992. New Dehli, India.

[RvHO91] J. Rushby, F. von Henke, and S. Owre. An Introduction to Formal Specification and Verification using EHDM. Technical report, SRI International, March 1991.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual.* Series in Computer Science. Prentice Hall International, 2nd edition, 1992.

[VSE94] *Sprachbeschreibung VSE-SL*, 1994. Version 1.