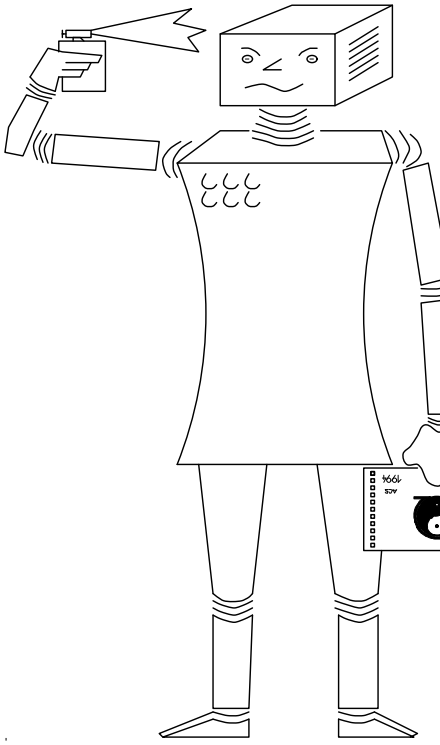

SEKI Report

Analogy in Verification of State-Based Specifications. First Results

Erica Melis[1], Claus Sengler[2]

SEKI Report SR-96-13

Analogy in Verification of State-Based Specifications. First Results

Erica Melis[‡], Claus Sengler[§]
Fachbereich Informatik, Universität des Saarlandes
66041 Saarbrücken, Germany

Abstract

The amount of user interaction is the prime cause of costs in interactive program verification. This paper describes an internal analogy technique that reuses subproofs in the verification of state-based specifications. It identifies common patterns of subproofs and their justifications in order to reuse these subproofs; thus significant savings on the number of user interactions in a verification proof are achievable.

*The first author was supported by the Deutsche Forschungsgemeinschaft Sonderforschungsbereich SFB 378.

†DFKI Saarbrücken

‡The first author was supported by the Deutsche Forschungsgemeinschaft Sonderforschungsbereich SFB 378.

§DFKI Saarbrücken

1 Introduction

Software verification is the job of taming complexity: in order to verify, say one hundred thousand lines of source code, several ten thousands proof obligations have to be shown, some of which may require formal proofs of up to eight or ten thousand steps. Usually these long proofs consist of a considerable number of relatively simple subproofs to be established. Even for a small percentage of interactive steps, i.e., those steps the user has to supply as opposed to those steps that are generated automatically by the system (in the VSE system currently, about 10 to 20% are user supplied), the interaction amounts to quite an effort for proofs with ten thousands of proof steps. Consequently, a major problem in software verification is the tremendous amount of user interaction needed that causes costs and a long development time. To minimize user interaction, is therefore a primary goal in order to reduce the cost of verified software. Again, from the experience with industrial applications of the VSE-system, the cost of verified code may be between twice and ten times the costs of ordinary software. The reuse of user-guided subproofs can contribute to that goal.

An important class of real world software verification problems have state-based specifications. State-based means that an invariant, e.g. a reliability statement, has to be proved for an initial state p_0 and for all states that can be reached by certain (admissible) state transitions T_i. Put formally, the theorem to be proved is

$$Inv(p_0) \bigwedge_i (Inv(p) \rightarrow Inv(T_i(p))), \tag{1}$$

where usually, the invariant $Inv(X)$ is a conjunction of *many* conjuncts and the proof of $Inv(p) \rightarrow Inv(T_i(p))$ may be similar for the different state transitions T_i. Therefore, a decomposition of the theorem leads to many similar proof obligations for one verification task and naturally suggests a reuse of these subproofs.

Reusing proofs has been addressed in different settings. A reuse of proofs in program verification after slightly changing the program (e.g., after a bug has been fixed) is addressed in [10]. External analogy, i.e., analogy between proofs of *different* theorems has been described in [6] and reuse of generalized rewrite proofs is described in [4]. As far as we know, internal analogy for verifying state-based specifications has not been tackled before.

This paper is organized as follows. First we describe the internal analogy paradigm suitable for reusing subproofs within the same large proof attempt. In particular, the reuse in verifying state-based specifications is addressed. Then we illustrate the usage of internal analogy with an example that is taken from a case study that, among others, verified the state-based specification of a communication filter.

Notation

We work with a sequent calculus; for other calculi we believe the procedure can be adapted appropriately. $H_1, \ldots, H_n \vdash C_1, \ldots, C_m$ abbreviates the sequent

$H_1 \wedge \ldots \wedge H_n \vdash C_1 \vee \ldots \vee C_m$. A *normal form* (NF) of this sequent is the set $\{H_1, \ldots, H_n, \neg C_1, \ldots, \neg C_m\}$. Note that this normal form does not distinguish between variants having,e.g., H as hypothesis and having $\neg H$ in the conclusion, respectively. A proof obligation is *provable* if H and $\neg H$ belong to its NF for some formula H. Variables are $\forall$-quantified if not stated otherwise.

2 Internal Derivational Analogy in the Verification of State-Based Specifications

Problem solving by analogy transfers the solution or the problem solving experience of a *source* problem to guide the search for a problem solution of a similar *target* problem. In general, the process of reasoning by analogy can be described as follows: A case base is kept of previously solved problems with accompanying solutions. When a new target problem is encountered, a similar problem is retrieved from this case base and its solution is used as a guide to the solution of the new problem by analogical replay.

Analogy requires (i) to map, and sometimes to reformulate, the source problem to the target problem, (ii) to extend the mapping and reformulation to the solutions, (iii) to replay, and finally to adapt the solution to the requirements of the target.

Derivational analogy [1] guides the target solution by replaying decisions of the source problem solving *process*, and it uses information about reasons for the decisions (*justifications*).

Internal analogy [2, 8] is a process that transfers experience from a completed subgoal (source) in the *same* problem solving process to solve a current subgoal (target). That is, in internal analogy the source and the target are subproblems of a single problem. Therefore, this technique does not require the effort to set up a permanent case base and needs relatively little search for the retrieval of a source, as opposed to analogy in general. Furthermore, little or no effort at all is required for the mapping because corresponding subgoals in one proof are very similar.

Still, internal analogy needs some extra effort for storing the justifications and for the mapping and hence, internal analogy pays in particular when it replaces search-intensive subtasks or interaction-intensive subtasks, see [8].

2.1 Internal Analogy for State-Based Verification

Internal analogy in software verification can sometimes be used to replace interaction-intensive subtasks. The accumulation of a library of cases is not required in our internal analogy. Usually, the subproofs need only to be cached, and often the source terms need not to be mapped into different target terms.

The internal analogy has two steps, retrieval and replay. Two modes of the retrieval are possible for the internal analogy as described in this paper:

- The retrieval of a source is done automatically.

input: source goal, guiding source subplan, target goal
output: (partial) target plan

1. Let C be the guiding subplan and c_i the current step in C.
2. Terminate if the target goal is proved.
3. Check of justifications: If the justification of c_i that corresponds to a subplan π holds in the target, then replay π.
4. Advance the case C to the next usable step c_j; $i \leftarrow j$; goto 2.

Table 1: Outline of the analogical replay

- The source is provided interactively.

In the first case, the analogy procedure includes searching for a source which is left to the user in the second case. The automated retrieval searches for (source) nodes in the proof plan the proof obligations of which are proved already and that have justifications holding in the current (target) node. For instance, as described in sections 2.2 and 3, the search for a reusable subproof automatically compares the `essence` justification of source nodes with the NF of the target problem. An efficient retrieval can be achieved by (lexicographically) ordering the formulae in the justifications and in the NFs. Henceforth, we use "NF" for ordered NF.

The analogical replay is an automated one in any case. It is given in a nutshell in Table 1. The justifications are checked in order to perform a warranted analogical transfer only. Its check of justifications is also simplified by ordering the formulae in the NF. The replayed subplan π may consist of a single step c_i, of several steps, or even of the whole source subplan. The "next usable step" depends on the satisfied justification j of c_i in C. All the steps that belong to the subproof corresponding to j are skipped. A generalization of the retrieval and the replay to multiple source subplans is possible.

In order to use internal derivational analogy, we have to store justifications of the source proof plan steps which we are going to replay.

2.2 Justifications

Justifications represent reasons for proof decisions. It is a non-trivial task to select appropriate justifications in a proof planning environment. For inductive theorem proving this task and a set of appropriate justifications are described in [5] and [8].

Our justifications are represented in a data structure attached to the proof plan nodes. This justification structure has different slots that store different kinds of justifications, as explained below. The justifications are checked during the replay. Only if at least one justification holds, the corresponding step or the subplan can be replayed.

For the verification of state-based specifications we analyzed the most common proof patterns and associated them with appropriate justifications. Frequent proof patterns are: (i) the reduction to small essential proof obligations by extracting *relevant* subformulae, (ii) the use of derived lemmata, and (iii) term generalization. These patterns can be combined in a proof.

In order to make the effort for the proof by analogy that includes checking the justifications as small as possible, we need to

- store all the information *relevant* for the replay but not more,
- store it in a form that is *available* during the source solution process and that can be *easily interpreted* in the target.

Taking into consideration the two requirements, we identified the following justifications for state-based specifications.

1. The user reduces the problem to essential proof obligations: If a proof obligation at a root node N_0

   $$H_1, \ldots, H_n \vdash C_1, \ldots, C_m$$

   is reduced to a proof of a sequent

   $$H_{i_1}, \ldots, H_{i_l} \vdash C_{j_1}, \ldots, C_{j_k}$$

   for $i_1, i_l \in \{1, \ldots, n\}$ and $j_1, j_k \in \{1, \ldots, m\}$, then the NF of the reduced sequent, called *essence*, is stored as a justification in the `essence` slot, e.g., (`essence`: $\{H_{i_1}, \ldots, H_{i_l}, \neg C_{j_1}, \ldots, \neg C_{j_k}\}$). `essence` contains all the relevant subformulae. Note that `essence` is a justification for a whole subproof rather than for single proof steps. Therefore, this justification is stored after the subproof has been completed. It is computed by goal regression [9] over the whole subproof.

   For a new subproblem in a node N it can be checked automatically whether its NF is a superset of N_0's `essence`. That is, it is checked whether the source and the target problem differ in irrelevant subformulae only. If yes, the *subproof* at N_0 can be fully replayed. In a target node, the `essence` is the justification checked first.

   Example: The essence of 5.3. below is a subset of the NF of 5.8. The rest of the proof obligation does not matter, and so the subproof of 5.3. can be completely replayed.

   Interpretation: If the NF of a target proof obligation is a superset of the NF in the justification slot `essence`, then this justification holds, and the source subproof can be replayed.

   Even in cases where no reduction was performed in the source, it is reasonable to store the essence of a subproof in order to be able to discover a similar essence of a target problem later on.

2. The user provides a lemma in the source that enables or considerably simplifies the proof. For instance in the subproofs of the example below, the lemma $x \in insert(Y, Z) \wedge x \notin Z \Rightarrow x \equiv Y$ is provided[1] that helps to complete several subproofs. The subset of (generalized) elements of the source NF that is needed to apply the lemma is stored as a justification in the `lemma` justification. `lemma` is computed by goal regression (backward) from the lemma application node N_l. The current value of the regressed goal is stored as `lemma` justification at each node visited by the goal regression.

   Example: In example 5.3. below, the `lemma` justification at node N_0 is: $\{x \in err, err \equiv insert(next(in'), err'), \neg(x \in err')\}$ because the goal regression yields $\{x \in insert(next(in'), err'), \neg(x \in err')\}$ in the first step and $\{x \in err, err \equiv insert(next(in'), err'), \neg(x \in err')\}$ in the second step.

   Interpretation: If the NF of the target problem is a superset of the source node's `lemma` justification, then the justification holds, and the lemma can be applied in the target.

   `lemma` is a justification for several steps rather than for a large subproof.

3. An extended form of the justification check does not require the source essence to be an exact subset of the target NF but additionally allows for a substitution of variables or even a mapping of terms. This more general `g-lemma` justification is produced by

   (a) in `lemma` replacing the substitution terms by the variables of the lemma they are substituted for and

   (b) replacing other constants not occurring in the lemma by variables.

   When we replace the `lemma` subset by the `g-lemma` subset in `essence`, we also obtain a more general `g-essence`.

   Example: From the `lemma` justification above and from the substitution $[Y \leftarrow next(in'), Z \leftarrow err']$, the `g-lemma` justification $\{x \in B, B \equiv insert(Y, Z),$
   $\neg(x \in Z)\}$ is produced.

   Interpretation: If `g-lemma` of a source node *matches* a subset of the NF of the target problem, then the justification holds and the lemma can be applied in the target node again. If the `g-essence` of a source matches a subset of the NF of the target problem, then the justification holds and the source subproof can be replayed.

[1] The semantics of the functions does not play a role at this moment. It will be explained in section 3.

4. Often, the theorem provers of a verification system are not able to prove a proof obligation without a user supplied generalization. Automated generalization is a very difficult task and, therefore, most often left to the user.

   The justifications `gen-essence` and `gen-lemma`, stored at a generalization node N_G of a plan, is produced by computing the `essence` and `lemma` of the *generalized* goal, respectively.

   Example: The proof of 6.3. in section 3 includes at node N_G the term generalization $max_value(sender(next(in')), clients')$ to X and of $value(next(in'))$ to Y. The `gen-essence` for the node is $\{(X < Y), \neg(Y > X)\}$. Note that this is a justification for the subplan with root N_G.

   Interpretation: If a subset of the NF of a target goal *matches* the `gen-essence` of a source node N, then the substitution provided by the match is used for the term generalization in the target, and the subproof for the goal at N can be replayed. If a subset of the NF of a target goal (node) *matches* the `gen-lemma` of a source node only, then the substitution provided by the match is taken for a generalization, and then the lemma application can be replayed.

3 Example: Proofs of Invariants

The following example stems from a case study performed with VSE, a verification support environment [3] at the DFKI in Saarbrücken. The goal of this case study is to model a communication filter: From an input queue a message is checked for certain properties. If these properties hold, the message is sent to an output queue. In case the properties do not hold, it is sent to an error queue.

A message is a compound object of several components: the addressee, the sender, the subject, and the message text. The input queue (*in*), the output queue (*out*), as well as the error queue (*err*) are first-in-first-out queues with the following functions:

- $nil :\rightarrow queue$ generating the empty queue,
- $insert : message \times queue \rightarrow queue$ inserts a message into a queue,
- $\in: message \times queue \rightarrow bool$ determining whether a message is contained in a queue,
- $next : queue \rightarrow message$ returning the message from the queue which is handled next, and
- $rest : queue \rightarrow queue$ deletes the message that is handled next from the queue and leaving all other entries unchanged.

In addition, for the communication system there is a data base (*base*) of all clients.

The check whether a message can pass the filter is done in two steps: First, it is checked whether the sender is a legal client. A function $known : name \times data_base \rightarrow bool$ returns true if for the name there is an entry in the data base. Secondly, the message is evaluated, and a natural number is computed, $value : message \rightarrow nat$. Moreover, for each client in the data base there is a maximal value, $max_value : name \times data_base \rightarrow nat$. If the value of a message does not exceed the maximal value associated with the sender, then the respective message is allowed to pass. As values one could imagine, for instance, the allowed lengths of a message text.

For this scenario a state-based specification [11, 3] was used: We have several state variables for the different queues and for the client data base. Furthermore, some state transitions were specified for the insertion operation on queues, and for the check whether a message can pass the filter. Each state transition is specified by the details of the changes they produce, i.e., by defining the precondition and the postcondition of a state transition. In these pre- and postconditions a state variable prior to the execution of the state transition is quoted as opposed to the state variable after the transition has been performed. For example, in' denotes the input queue before the transition has been performed, and in is the input queue afterwards.

A state-based specification is called *correct*, if a first-order formula – the *invariant* – holds for all reachable states. Hence, this invariant has to be proved for the initial state, and for all states that can be reached from the initial state. The invariant from our case study is:

$$\begin{array}{lcl} x \in out & \rightarrow & \left(\begin{array}{c} known(sender(x), base) \wedge \\ value(x) \leq max_value(sender(x), base) \end{array} \right) \\ x \in err & \rightarrow & \left(\begin{array}{c} \neg known(sender(x), base) \vee \\ value(x) > max_value(sender(x), base) \end{array} \right) \end{array} \quad \wedge$$

During the verification process the original large proof obligation has been decomposed into seven smaller proof obligations denoted by proofinv-i for $i = 1, \ldots, 7$. By simplifications and equation applications each proofinv-i is decomposed into several simpler proof obligations. For instance, proofinv-5, proofinv-6, proofinv-7 are each reduced to eight subgoals. We shall have a look at the proofs of these subgoals. In the following examples the shaded parts of the proof obligations indicate the *relevant* parts of the proof obligations. Note, how these relevant parts occur in several proof obligations giving rise to a reuse of proofs. **proofinv-5** is a rather large proof obligation:

$$\begin{array}{l} in' \not\equiv nil, x \in out' \rightarrow known(sender(x), base') \wedge \\ value(x) \leq max_value(sender(x), base'), \\ x \in err' \rightarrow \neg known(sender(x), base') \\ \vee value(x) > max_value(sender(x), base'), \neg known(sender(next(in')), base), \\ err \equiv insert(next(in'), err') \wedge out \equiv out' \wedge in \equiv rest(in') \vdash \\ (x \in out \rightarrow known(sender(x), base') \wedge \end{array}$$

$value(x) \leq max_value(sender(x), base')) \wedge (x \in err \rightarrow$
$\neg known(sender(x), base') \vee value(x) > max_value(sender(x), base'))$.

All but the resulting third and eighth subgoal can easily be simplified and proved. Originally, for the 19 proof steps of proofinv-5 10 user interactions were needed . By internal analogy approximately 50% of the interactions can be saved.

5.3. $known(sender(x), base')$, $x \in err$, $err \equiv insert(next(in'), err')$, $out \equiv out', in \equiv rest(in') \vdash$
$value(x) > max_value(sender(x), base')$, $x \in err'$, $x \in out'$, $known(sender(next(in')), base')$, $in' \equiv nil$
is proved by

- *manually* suggesting the lemma

$$x \in insert(Y, Z) \wedge x \notin Z \Rightarrow x \equiv Y. \tag{2}$$

  With the substitution $[Y \leftarrow next(in'), Z \leftarrow err']$ the application of this lemma gives

$$x \equiv next(in'). \tag{3}$$

- By simplification with (3) we obtain a subgoal $\ldots, H, \ldots \vdash \ldots, H, \ldots$ where H abbreviates $known(sender(x), base')$.

The justifications at the root node of 5.3. look as follows:
`essence` made up from all the shaded formulae.
`lemma`: $\{x \in err, err \equiv insert(next(in'), err'), \neg(x \in err')\}$ is constructed from $x \in err$, $err \equiv insert(next(in'), err')$ at the left hand side of the proof obligation and $x \in err'$ at the right hand side. `lemma` provides the elements of the `essence` relevant for the lemma application. The other shaded formulae are relevant for the remaining proof steps.
`g-lemma`: $\{x \in B, B \equiv insert(Y, A), \neg(x \in A)\}$.

5.8. $known(sender(x), base')$, $x \in err$, $err \equiv insert(next(in'), err')$, $out \equiv out'$, $in \equiv rest(in'), known(sender(x), base')$, $value(x) \leq max_value(sender(x), base') \vdash$
$value(x) > max_value(sender(x), base')$, $x \in err'$, $known(sender(next(in')), base')$, $in' \equiv nil$
can be proved by analogy to proof obligation 5.3. because the `essence` of 5.3. is a subset of the NF of 5.8. as well.

proofinv-6 is decomposed into eight proof obligations. All but the resulting third and eighth subgoal can be immediately simplified and proved automatically. The more complicated subproofs are outlined below. Originally, for the 22 proof steps of proofinv-6 13 user interactions were needed . By internal analogy approximately 80% of the interactions can be saved.

6.3. $known(sender(x), base'), x \in err, out \equiv out', in \equiv rest(in')$
$max_value(sender(next(in')), base') < value(next(in'))$,
$known(sender(next(in'), base'))), err \equiv insert(next(in'), err'), \vdash$
$value(x) > max_value(sender(x), base'), x \in err', x \in out', in' \equiv nil.$
is proved by

- reusing the lemma application from 5.3. because the `lemma` justification holds in 6.3. The rest of the subproof differs though.
- Then interactively generalizing $max_value(sender(x), clients')$ to X and $value(x)$ to Y at node N_G results in the problem $\ldots, X < Y, \ldots \vdash \ldots, Y > X, \ldots$. This goal can be proved automatically.
- This subproof automatically uses the lemma $X < Y \leftrightarrow Y > X$.

`essence` at the root node of 6.3. is provided by all the shaded formulae.
`gen-essence` at N_G is $\{X < Y, \neg(Y > X)\}$.

6.8. $known(sender(x), base'), x \in err, in \equiv rest(in'), known(sender(x), base'),$
$max_value(sender(next(in')), base') < value(next(in'))$,
$known(sender(next(in'), base'))), err \equiv insert(next(in'), err'), out \equiv out',$
$value(x) \leq max_value(sender(x), base') \vdash$
$value(x) > max_value(sender(x), base'), x \in err', in' \equiv nil,$
is proved by reusing the proof of 6.3. because `essence` of 6.3. is a subset of 6.8.'s NF.

Only the third and eighth subgoal of **proofinv**-7 can be simplified and proved immediately. The other goals are proved by analogy. Originally, for the 40 proof steps of proofinv-7 31 user interactions were needed . By internal analogy approximately 90% of the interactions can be saved.

7.1. $x \in out, known(sender(next(in')), base'), err \equiv err',$
$out \equiv insert(next(in'), out'), in \equiv rest(in') \vdash$
$known(sender(x), base'), x \in err', x \in out'$
$max_value(sender(next(in')), base') < value(next(in'))in' \equiv nil$
is proved by reusing the subproof of 5.3.

7.2. $x \in out, known(sender(next(in')), base'),$
$err \equiv err', out \equiv insert(next(in'), out'), in \equiv rest(in') \vdash$
$value(x) \leq max_value(sender(x), base'), x \in err', x \in out',$
$max_value(sender(next(in')), base') < value(next(in')))in' \equiv nil$
The lemma application of 7.1. is reused.
Then at N_G, interactive generalization yields $\ldots \vdash \ldots, X \leq Y, Y < X, \ldots$
which can be proved automatically.
This provides the `gen-essence` $\{\neg(X \leq Y), \neg(Y < X)\}$ for $[X \leftarrow value(x), Y \leftarrow max_value(sender(x), base')]$.

7.4. For $x \in out, known(sender(next(in')), base'), err \equiv err',$
$out \equiv insert(next(in'), out'), in \equiv rest(in') \vdash$

$known(sender(x), base')$, $known(sender(x), base')$, $x \in out'$,
$max_value(sender(next(in')), base') < value(next(in')), in' \equiv nil$.

the subproof of 7.1 can be reused.

7.5. For $x \in out$, $known(sender(next(in')), base')$, $err \equiv err'$,
$out \equiv insert(next(in'), out')$, $in \equiv rest(in') \vdash$
$value(x) \leq max_value(sender(x), base')$, $known(sender(x), base')$,
$x \in out'$, $max_value(sender(next(in')), base') < value(next(in')), in' \equiv nil$.

the subproof of 7.1 can be reused.

7.6. For $x \in out$, $value(x) > max_value(sender(x), base')$,
$known(sender(next(in')), base)$, $err \equiv err'$
$out \equiv insert(next(in'), out')$, $in \equiv rest(in') \vdash$
$known(sender(x), base')$, $x \in out'$,
$max_value(sender(next(in')), base') < value(next(in'))in' \equiv nil$.

the subproof of 7.1 can be reused.

7.7. $x \in out$, $value(x) > max_value(sender(x), base')$,
$known(sender(next(in')), base')$, $err \equiv err'$,
$out \equiv insert(next(in'), out')$, $in \equiv rest(in') \vdash$
$value(x) \leq max_value(sender(x), base')$, $x \in out'$,
$max_value(sender(next(in')), base') < value(next(in'))$, $in' \equiv nil$.

The lemma application of 7.1. can be reused.
Then the resulting subgoal is proved by automatically applying the lemma $X > Y \leftrightarrow \neg(X \leq Y)$. The first step of 7.1. can be replayed because its `lemma` justification holds.

4 Conclusion and Future Work

Since user interaction accounts for the lions share of the costs for the formal proofs in program verification, there is every incentive to reduce these costs by a higher degree of automation. This paper has addressed the problem of saving user interaction in the verification of state-based specifications.

From the given examples it is clear that and how whole subproofs, generalizations, and lemma applications can be reused if the same justifications hold for the target subproblem. In our example the savings of user interactions achieved by internal analogy sums up to about 80%.

Our technique is based on the general idea of internal analogy that transfers source subproofs to target subproofs in the same proof attempt. It turns out that state-based specifications give rise to many similar proof obligations in their verification. We identified common patterns of subproofs and their justifications in order to employ them for the reuse of subproofs and proof steps.

The presented techniques are just a beginning: More elaborate justifications and mapping techniques will be explored in the future in order reuse more and

even larger proofs. In particular, retrieval and replay have to be extended to handle multiple sources.

References

[1] J.G. Carbonell. Derivational analogy: A theory of reconstructive problem solving and expertise acquisition. In R.S. Michalsky, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, pages 371–392. Morgan Kaufmann Publ., Los Altos, 1986.

[2] Angela K. Hickman, Peter Shell, and Jaime G. Carbonell. Internal analogy: Reducing search during problem solving. In C. Copetas, editor, *The Computer Science Research Review 1990*. The School of Computer Science, Carnegie Mellon University, 1990.

[3] Dieter Hutter, Bruno Langenstein, Claus Sengler, Jörg H. Siekmann, Werner Stephan, and Andreas Wolpers. Deduction in the Verification Support Environment (VSE). In Marie-Claude Gaudel and James Woodcock, editors, *Proceedings of the Third International Symposium of Formal Methods Europe*, pages 268–286, Oxford, England, 1996.

[4] Th. Kolbe and Chr. Walther. Reusing Proofs. In *Proceedings of 11th European Conference on Artificial Intelligence (ECAI-94)*, Amsterdam, 1994.

[5] E. Melis. Analogy in CLAM. Technical Report DAI Research Paper No 766, University of Edinburgh, AI Dept, Dept. of Artificial Intelligence, Edinburgh, 1995. available from http://jswww.cs.uni-sb.de/~melis/.

[6] E. Melis. A model of analogy-driven proof-plan construction. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 182–189, Montreal, 1995.

[7] E. Melis and C. Sengler. Analogy in verification of state-based specifications: First results Seki report, SR-96-13, 1996. available from http://jswww.cs.uni-sb.de/pub/seki/.

[8] E. Melis and J. Whittle. Internal analogy in inductive theorem proving. In M.A.McRobbie and J.K. Slaney, editors, *Proceedings of the 13th Conference on Automated Deduction (CADE-96)*, Lecture Notes in Artificial Intelligence, 1104, pages 92–105, Berlin, New York, 1996. Springer.

[9] T.M. Mitchell and R.M. Keller and S.T. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning* 1, pages 47–80, 1986.

[10] W. Reif and K. Stenzel. Reuse of proofs in software verification. In R.K. Shyamasundar, editor, *Proc. 13th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 761 of *LNCS*. Springer, 1993.

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