UNIVERSITÄT DES SAARLANDES
FACHRICHTUNG INFORMATIK
D–66123 SAARBRÜCKEN
GERMANY
WWW: http://www.ags.uni-sb.de/

SEKI-Report ISSN 1437-4447

A Calculus-Independent Proof Data Structure

Dominik Dietrich, Serge Autexier
FR Informatik, Saarland Univ.
dodi|serge@ags.uni-sb.de

SEKI-Report SR–2005–03

This SEKI-Report was internally reviewed by:
Claus-Peter Wirth
FR Informatik, Universität des Saarlandes, D–66123 Saarbrücken, Germany
E-mail: cp@ags.uni-sb.de
WWW: http://www.ags.uni-sb.de/~cp/welcome.html

Editor of SEKI series:
Claus-Peter Wirth
FR Informatik, Universität des Saarlandes, D–66123 Saarbrücken, Germany
E-mail: cp@ags.uni-sb.de
WWW: http://www.ags.uni-sb.de/~cp/welcome.html

A Calculus-Independent Proof Data Structure

Dominik Dietrich, Serge Autexier
FR Informatik, Saarland Univ.
dodi|serge@ags.uni-sb.de

Searchable Online Version
Definitive Print Edition November 30, 2005

Abstract

A practically useful mathematics assistance system requires the sophisticated combination of interaction and automation. Central in such a system is the proof data structure, which has to maintain the current proof state and which has to allow the flexible interplay of various components including the human user. We describe a parameterized proof data structure for the management of proofs, which includes our experience with the development of two proof assistants. It supports and bridges the gap between abstract level proof explanation and low-level proof verification. The proof data structure enables, in particular, the flexible handling of lemmas, the maintenance of different proof alternatives, and the representation of different granularities of proof attempts.

Contents

1 Introduction

The main goal of the ΩMEGA-group is the development of the mathematics assistance system ΩMEGA[21] which aims to support working mathematician to find proofs. ΩMEGA provides a central proof data structure, the so-called $\mathcal{PDS}$[9]. Here proofs are represented and modified during the proof construction with the final goal in finding a proof in a machine checkable calculus. Furthermore, it provides information to generate a proof readable for the user[24].

Given a theorem in the form of a conclusion and its assumptions, a proof in ΩMEGA can be constructed by successively applying inference rules, currently a higher order variant of natural deduction[12] (ND)[1], which represent valid steps of deduction. An inference rule can be applied both forwards, resulting in the introduction of new assumptions, and backwards, reducing the current open goal, which is an unjustified proof node, to a set of new goals. Each inference step is stored as a justification of a proof node.

As these rules only allow very small steps, many interactive systems such as HOL[13] and ISABELLE[19] define the notion of a *tactic* which encapsulates repeatedly occurring sequences of inference rules combined with simple commands such as *repeat*, *then*, *or else* – called tacticals – in a macro step in order to automate proofs of repeatedly occurring subproblems. In a bottom up manner, more and more complex tactics are constructed until an adequate granularity is reached. An application of such a tactic results in the execution of the steps encoded in the tactic and is always sound, provided the underlying calculus is. Such tactics are called LCF tactics named after the proof checker LCF invented by Robin Milner and his group in 1977 (cf. [18]).

ΩMEGA follows an opposite approach by supporting the top-down construction of tactics. Technically, a tactic in ΩMEGA consists of a derivation procedure and an expansion procedure. The first performs a high level deduction step whereas the latter is responsible for the expansion, i.e. the verification of this step. The expansion of a tactic can result in the introduction of other tactics that have to be expanded again.

Tactics in ΩMEGA can be quite powerful, e.g. the call to an external system such as a computer algebra system (CAS), model generators (MG's), automatic theorem provers (ATP's) or constraint solvers (CS) can be represented in a tactic. But this doesn't come without a price: ΩMEGA tactics need not be sound, i.e. the expansion of a tactic can fail, as we cannot guarantee the soundness of the external systems. Furthermore, tactics need not be specified completely. Hence for a proof (plan) build from ΩMEGA tactics to be sound it needs to be fully expanded and checked, that is we need to find a proof using only rules provided by the calculus.

In tactical theorem proving, the user has to select the tactics manually to perform a proof. In ΩMEGA, proof planning is used to automate this process. The key idea of this approach is to augment tactics by pre- and postconditions to get so called methods and use AI-planning techniques to find a sequence of tactics, which constitute a proof.

In general an AI-planning problem consists of an initial state, a goal to be achieved and a set of operators that can be used to transform a state to another state. Applied to a planning problem, a planner tries to find a sequence of instantiated operators, called actions, that transform the initial state to the goal state. Proof planning applies planning techniques to a proof problem. The idea is that a proof is planned with a very coarse granularity using abstract planning operators, so that an outline of the proof is found. The outline is a sequence of basic tactics or complex methods.

[1] an overview of these rules is given in [17]

method PeanoInduction	
premises	$\oplus L1$, $\oplus L2$
conclusions	$\ominus L3$
application.condition	sort(n)=Nat
proof schema	L1 $\vdash P(0)$ L2 $\vdash P(k) \rightarrow P(k+1)$ L3 $\forall n.P(n)$

Figure 1: PeanoInduction method

This outline has to be expanded until the calculus level is reached.

The procedure is as follows: In each step the planner essentially determines the set of applicable methods. If more than one method is applicable, it has to choose among this set. In ΩMEGA, these choice points can be influenced by so called control rules. They order the set of applicable methods or may exclude some methods from this set.

In ΩMEGA, a basic method is defined as a tactic enriched by pre- and postconditions. Technically methods are 4-tuples consisting of premises, conclusions, application conditions and a proof schema. Premises and conclusions are annotated with $\oplus, \ominus$ in a STRIPS style with the following meaning: When applying the method, conclusions annotated with $\ominus$ are deleted as open goals, premises with annotation $\oplus$ are added as new goals, premises annotated with $\ominus$ are deleted as assumptions and conclusions annotated with $\oplus$ are added to the assumptions. They are used to determine if – in a given state – the method matches the existing proof nodes. If so, the method is applicable if additionally the application conditions are fulfilled. The proof schema provides the information for a schematic expansion of the method, i.e. introduces a partial proof plan into the $\mathcal{PDS}$.

Consider for example the ΩMEGA-method *PeanoInduction* shown in Figure 1. It states that in order to show that for all natural numbers n a property $P(n)$ holds $(L3)$, it is sufficient to show that the property holds for the natural number 0 $(L1)$ and whenever it holds for a natural number k, then it also holds for the natural number $k+1$ $(L2)$. So whenever L3 occurs in an open node, it can be closed by introducing two new open subgoals L1 and L2 which have to be proved. Hence we split the proof of $\forall n.P(n)$ to 2 subproofs of $P(0)$ and $P(k) \rightarrow P(k+1)$.

During the planning process, the planner works on the central data structure $\mathcal{PDS}$, which represents the current proof attempt, using its planning operators, i.e. methods. The $\mathcal{PDS}$ is represented as a directed acyclic graph containing open and closed nodes. Open nodes represent problems that need to be solved and closed nodes represent already solved problems. The goal is to reach a state where all nodes are closed, that is, the theorem is proved.

Technically speaking the $\mathcal{PDS}$ is a directed acyclic graph, consisting of open and closed nodes, justifications that connect nodes and hierarchical edges that connect justifications. Hierarchical edges allow us to represent the dependencies of an abstract proof step and its expansion.

Initially, the $\mathcal{PDS}$ consists of the proof assumptions as closed nodes and the theorem to be proved as an open node. The application of a method results in adding new proof nodes to the $\mathcal{PDS}$, corresponding either to the new assumptions introduced by the method, if it is applied in forward-style, or to the subgoals, if it is applied in backward-style. The premises and conclusions of the method are then connected via a justification link. The left side of Figure 2 shows the initial

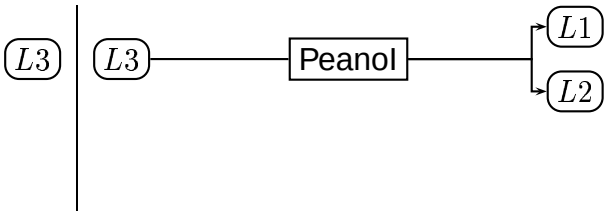


Figure 2: Applying Peano induction to $L3$

$\mathcal{PDS}$, where $L3$ contains the formula $\forall x.P(x)$. The right side shows the $\mathcal{PDS}$ after applying the method PeanoInduction.

Though methods can be defined with a suitable granularity, this approach has some drawbacks: In the current version of the $\mathcal{PDS}$ alternative proof steps cannot be represented. So it is not possible for the user to tackle different proof approaches in parallel within the same proof data structure.

Additionally, as analyzed in [7] the ND-calculus imposes strong restrictions on how methods can be designed. So the order in which the inference rules are applied, e.g. when eliminating existential quantifiers (see [7] for an example), plays an essential role for the success of proof expansion and must be considered at the planning level, which is not desirable. Furthermore, the splitting of a proof tree hampers proof explanation.

A possible solution would be

a) to have a calculus with fewer restrictions

b) to have an independent planning layer

In the new version of the ΩMEGA-system, we do both: The ND-calculus is replaced by the CORE-calculus[2] and an independent planning layer, the so called task-layer[4] is introduced. Basically a task represents a subproblem, i.e. a formula that has to be solved in order to solve the overall problem. Tasks can be manipulated using so called actions: They can split a task into several subtasks that are either conjunctively or disjunctively related, or correspond to lemma applications or inductive arguments. Despite these changes, we keep the hierarchical aspect and argue for a proof plan datastructure that

1. allows speculative or unverified proof steps that can be verified at a later time, independently of the current proof
2. represents different levels of abstractions within one proof object explicitly
3. is independent from the underlying calculus, and
4. can maintain different proof attempts simultaneously.

This paper describes the details and the implementation of the new $\mathcal{PDS}$ designed along these lines[2] and it is organized as follows: In Section 2 we give a technical description of the current $\mathcal{PDS}$ and its extension into the new $\mathcal{PDS}$. In Section 4 we sketch how the technical description can be translated into a software model and define an operational semantics for the most important functions. Finally, Section 5 gives a summary of the work.

2 Theoretical Requirements

In the following section we give a formal definition of the old $\mathcal{PDS}$, which has already been informally introduced in Section 1. Starting from this definition, we generalize this model and close the section with the definition of the new $\mathcal{PDS}$.

2.1 The old $\mathcal{PDS}$

Intuitively we view a proof as a three dimensional proof object that consists of horizontal and vertical structures. A *horizontal* structure is a (partial) proof plan on a specific *level* of abstraction including alternatives, while a *vertical* structure links proof plans at different levels of abstraction.

Definition 2.1 (old $\mathcal{PDS}$) A $\mathcal{PDS}$ graph S is a directed acyclic graph $\langle \mathcal{N}, \mathcal{J}, \mathcal{H} \rangle$ where

- $\mathcal{N}$ is a finite set of nodes. Each node consists of a formula, a set of hypotheses (forming a sequent) and has a unique label.
- $\mathcal{J}$ is a finite set of *justification links* between elements of $\mathcal{N}$. Each justification connects a node $n \in \mathcal{N}$ with a set $T \subset \mathcal{N}$ of nodes $(j = n \stackrel{i}{\rightarrow} T)$ where and i is any inference rule, method or tactic. n is called the source and the elements of T the targets of j. Furthermore we define the set of incoming justifications $I_n := \{j \in \mathcal{J} | n \in target(j)\}$ and the set of outgoing justifications $O_n := \{j \in \mathcal{J} | source(j) = n\}$
- $\mathcal{H}$ is a finite set of *hierarchical edges* between elements of $\mathcal{J}$ $(h = j_1 \stackrel{h}{\longrightarrow} j_2)$. Given a hierarchical edge $j_1 \stackrel{h}{\rightarrow} j_2$, j_1 is called source and j_2 is called target of h.
- If a node $n \in N$ has more than one outgoing justification, then the justifications must be connected by hierarchical edges:

$$\forall n \in \mathcal{N}. \exists j_1, j_2 \in O_n (j_1 \neq j_2 \Rightarrow \exists h \in \mathcal{H} (j_1 \stackrel{h}{\rightarrow} j_2 \vee j_1 \stackrel{h}{\rightarrow} j_1))$$

Furthermore, the following conditions have to hold:

- Hierarchical edges must satisfy

$$\forall h \in \mathcal{H}. source(target(h)) = source(source(h))$$

[2] A short description of the $\mathcal{PDS}$ has been published in [3].

and given $h = (j_1 \xrightarrow{h} j_2) \in \mathcal{H}$

$$\forall x \in target(j_2) : x \text{ is reachable from } j_1 \text{ using only justification links}$$

i.e. horizontal links are only allowed between justifications sharing the same source and the targets of the more abstract nodes have to be reachable from the less abstract ones.

- Each justification has at most one outgoing and one incoming hierarchical edge

In this definition, acyclic means that there are no cycles allowed using justifications or hierarchical edges or both.

Justifications build up the horizontal structure in a proof, i.e. they connect a goal (node) with a set of subgoals that have all to be solved in order to solve the goal. This connection is justified by a calculus rule at the finest granularity of the proof and by tactics, methods or lemmas. Hence justifications can be seen as an AND-relation between the subgoal nodes.

There are two kinds of nodes, namely open and closed nodes. Open nodes correspond to (sub)problems, that still have to be solved and closed nodes to (sub)problems that have already been solved. Initially, the node containing the theorem to be proved is open and nodes storing the assumptions of the theorem are closed. Applying a justification to an open node closes that node; the newly introduced nodes are open. Applying a justification on closed nodes yields again a closed node. Figure 3 shows an example proof represented in the old $\mathcal{PDS}$. Given A and B,

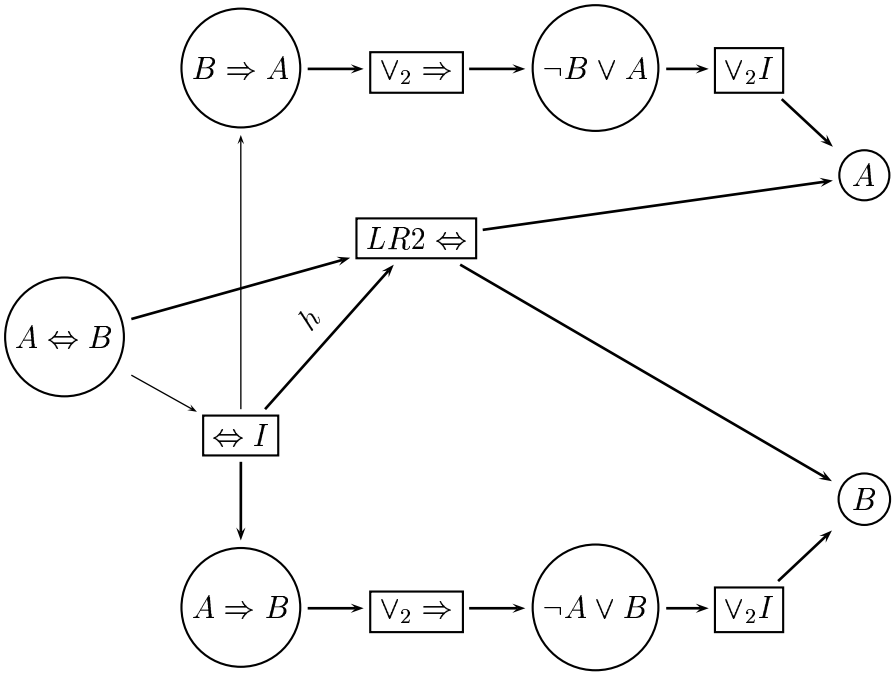


Figure 3: Old $\mathcal{PDS}$

the goal is to prove $A \Leftrightarrow B$. To show that, the tactic "$LR2 \Leftrightarrow$" can be applied, which states that A and B are equivalent if they are both true. As explained in Section 1, we need to expand the proof until we reach calculus level which we can proof check. This results in the insertion of $\Leftrightarrow I$ and two new open nodes containing $B \Rightarrow A$ and $A \Rightarrow B$. Both can be closed in a few steps. The

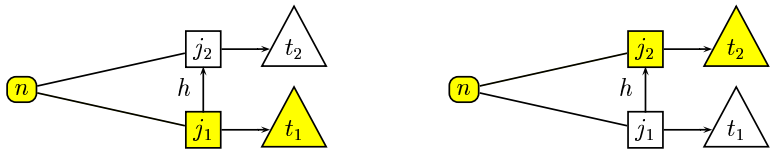


Figure 4: Possible views of proofs at different granularities inside a PDS

---

expanded part of "$LR2 \Leftrightarrow$" can be seen as subproof of the abstract tactic. The user can inspect the current proof-object at a specific granularity by selecting one specific justification for each node from the set of its outgoing justifications. This concept is defined in the $\mathcal{PDS}$-view:

Definition 2.2 (*Old* $\mathcal{PDS}$-view) Let $\langle \mathcal{N}, \mathcal{J}, \mathcal{H} \rangle$ be a $\mathcal{PDS}$. A $\mathcal{PDS}$-view $\mathcal{V} \subset \mathcal{J}$ selects for each $n \in \mathcal{N}$ exactly one outgoing justification (if one exists):

$$\forall n \in \mathcal{N}.O_n \neq \emptyset \Rightarrow \exists! v \in \mathcal{V}.v \in O_n$$

Figure 4 shows a node n which has two outgoing justifications j_1, j_2 that are connected by a hierarchical edge $h : j_1 \xrightarrow{h} j_2$. Hence j_1 corresponds to an expansion of j_2. If $\mathcal{V}$ is a $\mathcal{PDS}$-view corresponding to that $\mathcal{PDS}$, one of the justifications j_1, j_2 has to be in the set of selected justifications $\mathcal{V}$. If j_1 is selected, then the user sees the expanded, i.e. the more detailed version of the proof as shown on the left hand side, otherwise he sees the abstract version as shown on the right hand side. The selected justification is shadowed in the Figure.

2.2 The new $\mathcal{PDS}$

In the previous section we introduced the old $\mathcal{PDS}$. This structure is extended to

a) support alternative proof steps

b) parameterize the $\mathcal{PDS}$ over the actual content of nodes and edges

c) extend the tree structure to a multiple forest structure

First, the restriction to have specific content for nodes and justifications is removed. Furthermore, we allow there to be several outgoing justifications, which need not be connected by a hierarchical edge. Such justifications correspond to alternative ways to tackle the same problem stored in the justified node.

This extension requires an adaptation of the concept of $\mathcal{PDS}$-view, as we now have to select one abstraction level for each alternative. So each node maintains a set for selected alternatives.

Thereafter we extend the notion of a $\mathcal{PDS}$ to a $\mathcal{PDS}$-forest by putting several $\mathcal{PDS}$s together. We introduce a new kind of edge that allows to express dependencies between several $\mathcal{PDS}$s as well as the notion of a forest-view, which can be seen as a slice plane through a forest. Finally $\mathcal{PDS}$-forests can be put together to built multi-forests.

Definition 2.3 (PDS) A PDS is composed of nodes, justifications and hierarchical edges. Each such component x of a PDS is labeled with a pair $\mathsf{label}(x) = (c, t)$, where c maintains arbitrary content and $t \in \mathbb{N}$ is a timestamp. The time information enables us to define an order in which the objects have been created. The content of the labels can be freely instantiated, for instance, with proof statements in the case of proof nodes or with names of proof rules, tactics, and methods in the case of justifications. That is, our approach is parameterized over this sort of information that is typically very specific to different proof assistants.

Formally, a PDS is defined as a triple $P := \langle \mathcal{N}, \mathcal{J}, \mathcal{H} \rangle$ where

- $\mathcal{N}$ is a nonempty finite set of *nodes*. Each node $n \in \mathcal{N}$ has a label l, denoted as $\mathsf{label}(n)$.
- $\mathcal{J}$ is a finite set of *justifications*. Each justification $j \in \mathcal{J}$ is a triple (s, T, l). $s \in \mathcal{N}$, $T \subseteq \mathcal{N}$, and l specify the *source*, the *targets*, and the *label* of j. They are denoted as $\mathsf{source}(j)$, $\mathsf{targets}(j)$, and $\mathsf{label}(j)$, respectively. We will also denote justifications as $s \xrightarrow{l} T$. Generally, a justification $s \xrightarrow{l} T$ represents a *proof step* in which the proof node s is reduced to the nodes in T by application of the operator l. For each node $n \in \mathcal{N}$, we define the set of *incoming justifications* by $I_n := \{j \in \mathcal{J} | n \in \mathsf{targets}(j)\}$, and the set of *outgoing justifications* by $O_n := \{j \in \mathcal{J} | \mathsf{source}(j) = n\}$. The *graph* of $\mathcal{J}$ is $\{(\mathsf{source}(j), n) | j \in \mathcal{J} \wedge n \in \mathsf{targets}(j)\}$; we require it to be acyclic.
- We require that there exists exactly one node $n_r \in \mathcal{N}$ with $I_{n_r} = \emptyset$, called the *root node*.
- $\mathcal{H}$ is a finite set of *hierarchical edges* on $\mathcal{J}$. Each hierarchical edge $h \in \mathcal{H}$ is a triple (j_1, j_2, l). $j_1 \in \mathcal{J}$, $j_2 \in \mathcal{J}$, and l specify the *source*, the *target* and the *label* of h. They are denoted as $\mathsf{source}(h)$, $\mathsf{target}(h)$, and $\mathsf{label}(h)$, respectively. We will denote hierarchical edges also as $j_1 \xrightarrow{h}_l j_2$. The *graph* of $\mathcal{H}$ is defined as the set of pairs $\{(\mathsf{source}(h), \mathsf{target}(h)) | h \in \mathcal{H}\}$; we require it to be acyclic. For all hierarchical edges $j_1 \xrightarrow{h}_l j_2$ we require:
  - $\mathsf{source}(j_1) = \mathsf{source}(j_2)$ (i.e. hierarchical edges may only connect justifications sharing the same source node), and
  - for each $n_2 \in \mathsf{targets}(j_2)$ there exists an $n_1 \in \mathsf{targets}(j_1)$ such that (n_1, n_2) is in the reflexive and transitive closure of the graph of $\mathcal{J}$ (i.e. j_1 is the first proof step of a derivation that refines the proof step characterized by j_2).

Essentially, we have extended our graph to an AND-OR-graph, where the successor nodes of a justification are related by AND, that is have to be solved in order to solve the problem stored in the source, and justifications sharing the same source node but are not connected by hierarchical edges are OR-related, that is, only one of them has to be solved. Hence they represent alternative ways to tackle the same problem.

The time information enables us to define an order in which the objects have been created.

From the problem-solving point of view we need to know if within a horizontal layer a problem – including all its related subproblems – has already been solved or which (sub)problems still need to be solved. We introduce the following terminology in order to distinguish the different situations:

Definition 2.4 (Open/Closed Nodes) Let $P = \langle \mathcal{N}, \mathcal{J}, \mathcal{H} \rangle$ be a PDS and $n \in \mathcal{N}$ be a node of P. n is called *locally closed* if and only if there exists a $j \in J$ with $\mathsf{source}(j) = n$ and $\mathsf{target}(j) = \emptyset$; i.e. n is justified without reducing it to new subproblems. n is called *tree-wide closed* if it is locally closed or if there is a $j \in O_n$ such that all $m \in \mathsf{targets}(j)$ are tree-wide closed. The latter says that n is justified by a reduction to subproblems $m \in \mathsf{targets}(j)$ which are all already (recursively tree-wide) closed. A node is called *locally/tree-wide open* if it is not locally/tree-wide closed. □

2.3 $\mathcal{PDS}$-View

Hierarchical edges build up the vertical structure of a proof by connecting two justifications and indicating that the inference step can be expanded to the more detailed subproof by the source justification. To pack up justifications representing proof attempts at different granularities for the same problem, we introduce an ordering induced by the hierarchical edges among outgoing justifications of a node as follows:

Definition 2.5 (Strict partial ordering) A strict partial ordering on a set M is a binary relation on M that is irreflexive and transitive.

Definition 2.6 ($\mathcal{H}$-Induced Orderings $<$ and $\leq$)
Given a PDS $S = \langle \mathcal{N}, \mathcal{J}, \mathcal{H} \rangle$ we define $<$ to be the transitive closure of the graph of $\mathcal{H}$ and $\leq$ to be the reflexive closure of $<$.

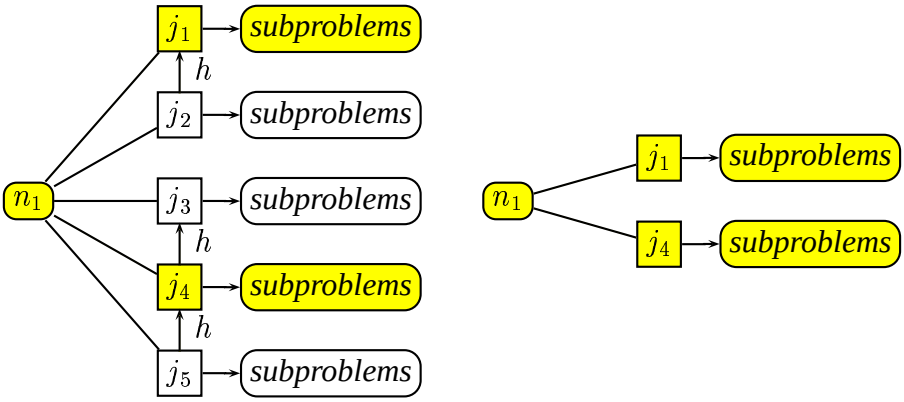
Note that $<$ and $>$ are well-founded orderings because the graph of $\mathcal{H}$ is acyclic and finite. In the remainder of this paper we consider a single ordering $<$ for each $\mathcal{PDS}$ instead of the family of induced orderings $(<_n)_n \in \mathcal{N}$.

So far a $\mathcal{PDS}$-node can have multiple outgoing justifications, representing alternative proof attempts or proofs at different levels of abstraction. During the proof construction or presentation, we need to restrict this set of justifications to get a complete set of alternatives at some specific granularity:

Definition 2.7 (Set of Alternatives) Let $\langle \mathcal{N}, \mathcal{J}, \mathcal{H} \rangle$ be a PDS, $n \in \mathcal{N}$, and $A \subseteq O_n$ a set of justifications for n.

- A is *adequate* if there are no $k, k' \in A$ such that $k{<}k'$.
- A is *complete* if for all $k \in O_n$ there is a $k' \in A$ such that $k{\leq}k'$ or $k'{\leq}k$.

A is a *set of alternatives for* n if it is adequate and complete. Given $j_1, j_2 \in O_n$, j_1 and j_2 are *comparable*, if $j_1{\leq}j_2$ or $j_2{\leq}j_1$; otherwise they are *not comparable*.



(a) PDS-node with all outgoing partially hierarchically ordered justifications, and j_1, j_4 in the set of alternatives. Justifications are depicted as boxes.

(b) PDS-node in the PDS-view obtained for the selected set of alternatives j_1, j_4.

Figure 5:

The adequacy property ensures that at most one descendant is selected for each alternative, whereas the completeness property says that there must be at least one. Hence for every alternative exactly one descendant is selected. Doing this selection for each node defines a $\mathcal{PDS}$-view:

Definition 2.8 ($\mathcal{PDS}$-view) A *$\mathcal{PDS}$-view* is a quadruple $\langle \mathcal{N}, \mathcal{J}, \mathcal{H}, \mathcal{L} \rangle$ where $\langle \mathcal{N}, \mathcal{J}, \mathcal{H} \rangle$ is a $\mathcal{PDS}$ and $\mathcal{L}$ is an $\mathcal{N}$-indexed family of alternatives.

Figure 5 shows a node n_1 with five outgoing justifications. These justifications can be divided into 2 groups, $G_1 = \{j_1, j_2\}, G_2 = \{j_3, j_4, j_5\}$, where the elements of a group represent the same proof attempt at different levels of abstraction. There are two possibilities to tackle the problem stored in node n_1, and for each we have to select a justification. So one member of the group G_1 and one of the group G_2 has to be selected.

Doing this selection for all nodes, we can view a complete $\mathcal{PDS}$ graph on a specific granularity:

Definition 2.9 (Current proof-level) Let $V = \langle \mathcal{N}, \mathcal{J}, \mathcal{H}, (L_n)_{n \in \mathcal{N}} \rangle$ be a $\mathcal{PDS}$-view. Define the *current proof-level of* V as the smallest $\mathcal{PDS}$ $\vec{C} = \langle \mathcal{N}', \mathcal{J}', \emptyset \rangle$ satisfying:

- $n_r \in \mathcal{N}'$,
- If $n \in \mathcal{N}'$ then all alternatives of n are in $\mathcal{J}'$, i.e. $\forall j \in L_n . j \in \mathcal{J}'$, and
- If $j \in \mathcal{J}'$ then all target nodes of j are in $\mathcal{N}'$, i.e. $\forall n' \in target(j) . n' \in \mathcal{N}'$.

Proofs with a coarse granularity correspond to proof plans or an outline of the proof which has to be expanded in order to get eventually a proof within a logical calculus. This calculus builds up the finest granularity, which is characterized as follows:

Definition 2.10 (Lowest $\mathcal{PDS}$-view) A $\mathcal{PDS}$-view $\vec{V} = \langle \mathcal{N}, \mathcal{J}, \mathcal{H}, (L_n)_{n \in \mathcal{N}} \rangle$ is called *lowest $\mathcal{PDS}$-view* if it satisfies:

$$\forall n \in \mathcal{N} \; \forall j \in (\mathcal{L})_n \; \neg \exists j' \in O_n . j' < j$$

Now we define the property of being an open or closed node with respect to a specific $\mathcal{PDS}$-view:

Definition 2.11 (tree-wide locally/global view-closed) Let S be a $\mathcal{PDS}$ and $\vec{V} = \langle \mathcal{N}, \mathcal{J}, \mathcal{H}, (L_n)_{n\in\mathcal{N}} \rangle$ a $\mathcal{PDS}$-view of S and let $n \in \mathcal{N}$. n is called *tree-wide locally view-closed* if it is tree-wide locally closed. It is called *tree-wide global view closed* if it is tree-wide locally view-closed or $\exists j \in L_n \forall n' \in O_j.n'$ is tree-wide global closed.

Definition 2.12 (tree-wide locally/global view-open) Let S be a $\mathcal{PDS}$ and $\vec{V} = \langle \mathcal{N}, \mathcal{J}, \mathcal{H}, (L_n)_{n\in\mathcal{N}} \rangle$ a $\mathcal{PDS}$-view of S and let $n \in N$. n is called *tree-wide locally view-open* with respect to $\vec{V}$ if and only if it is locally tree-wide open. It is called *tree-wide global view-open* if and only if it is not tree-wide global view-closed.

This concept can be extended to a whole $\mathcal{PDS}$-view:

Definition 2.13 (open/closed $\mathcal{PDS}$-view) Let S be a $\mathcal{PDS}$-graph and $\vec{V}$ be a $\mathcal{PDS}$-view of S. Then $\vec{V}$ is called *open $\mathcal{PDS}$-view* if and only if the root node n_r of S is global view-open. It is called *closed $\mathcal{PDS}$-view* if and only if the root node n_r of S is global view-closed.

In the case that we have a fully expanded proof, i.e. a proof at calculus level which does not contain open nodes, it can be proof checked. Such a $\mathcal{PDS}$ is called closed, otherwise it is called open:

Definition 2.14 (Basic open (closed) $\mathcal{PDS}$) Let S be a $\mathcal{PDS}$-graph and $\vec{L}$ be the lowest $\mathcal{PDS}$-view of S. Then S is called basic open (closed) if and only if the root node n_r of S is open (closed) with respect to $\vec{L}$.

2.4 Forests

We now extend the structure of a single $\mathcal{PDS}$-graph to a so-called $\mathcal{PDS}$-forest, i.e. a set of $\mathcal{PDS}$-graphs which can be interdependent, indicated by special links between those graphs. The idea behind this structure is that the proof of a theorem is a forest and consists of several trees. There is one tree representing the theorem to be proved; the others represent lemmas which can be used in other proof trees. The application of some lemma or theorem in some proof tree is represented by a forest link. To our knowledge, these proof forests occurred first in QUODLIBET system [6].

Definition 2.15 (Forest) Let $\mathcal{I}$ be an index set. A forest is a pair $\langle (PDS_i)_{i\in\mathcal{I}}, \mathcal{F} \rangle$ where

- $(PDS_i)_{i\in\mathcal{I}} = (\langle \mathcal{N}_i, \mathcal{J}_i, \mathcal{H}_i \rangle)_{i\in\mathcal{I}}$ is a family of disjoint PDSs.
- $\mathcal{F}$ is a finite set of *forest edges* between PDSs. Each forest edge $f \in \mathcal{F}$ is a pair (j, n_r) consisting of the source $\mathsf{source}(f) = j$, which is a justification from some $\mathcal{J}_i$, $i \in \mathcal{I}$, and the target $\mathsf{target}(f) = n_r$, which is the root node of some PDS $PDS_{i'}$, $n_r \in \mathcal{N}_{i'}$, $i' \in \mathcal{I}$. We denote a forest link (j, n_r) by $j \dashrightarrow n_r$.

Given a justification $j \in \mathcal{J}_i$ for some $i \in \mathcal{I}$, the set of outgoing forest links for that justification is denoted by $F_j := \{f \in \mathcal{F} | \mathsf{source}(f) = j\}$. □

Applying a new lemma on some justification j in a PDS p results in introducing a new PDS p' with root node n_r and a forest edge that connects j to n_r. Although we intuitively apply lemmas to a goal stored in a node, a forest edge starts at a justification of this node. This is necessary to determine in which alternative the lemma is to be applied. The node n is eventually *tree-wide closed* by the justification j, but n remains *forest-wide open* until the n_r in the PDS p' (just as the target nodes $\mathsf{targets}(j)$) are (forest-wide) closed.

Note that forest edges can produce cycles. This allows us to apply a lemma to itself, which is needed to represent induction in the form of *descente infinie* [26]. Moreover, due to our AND-OR proof trees these cycles may refer to different choices of OR-proofs.

2.5 Forest-Views

We now extend the notion of a PDS-view to the notion of a forest.

Definition 2.16 (Forest-View) Let $\langle \mathcal{PDS}, \mathcal{F} \rangle$ be a forest. A *forest-view with respect to some* $p \in \mathcal{PDS}$ is a forest $\langle \mathcal{PDS}', \mathcal{F}' \rangle$, such that $\mathcal{PDS}'$ and $\mathcal{F}'$ are the smallest sets with:

- The PDS-view for p is in $\mathcal{PDS}'$.
- For all justifications j in some PDS-view from $\mathcal{PDS}'$ and for all forest edges $j \dashrightarrow n_r \in \mathcal{F}$, $n_r \in p'$:
  $j \dashrightarrow n_r$ and the PDS-View for p' are contained in $\mathcal{F}'$ and $\mathcal{PDS}'$, respectively.

2.6 Multi-Forests

So far there is only the concept of a forest, containing any number of trees and links between these trees. For the purpose of the implementation, we introduce the notion of multiple forests, which consist of forests. For example, this can be useful when working on different problems in parallel.

Definition 2.17 (Multi-Forest, Multi-Forest-View) Let $\mathcal{T}$ be an index set. A *multi forest* is an $\mathcal{T}$-indexed set $\langle F_m \rangle_{m \in \mathcal{T}}$ where each F_m is a forest. A *multi-forest-view* is a $\mathcal{T}$-indexed set $\langle \vec{V} \rangle_{m \in \mathcal{T}}$ where each $\vec{V}_m$ is a forest view.

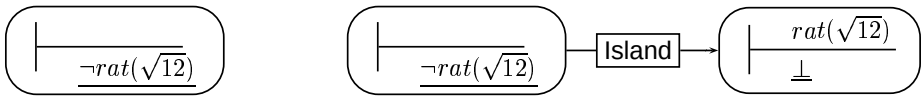


Figure 6: PDS trees respectively after step 0 and step 1

3 Sample Application

Our application of the proof data structure presented in this paper within the ΩMEGA project instantiates the framework with so-called *proof tasks*; i.e. they become the nodes of our proof data structure. Tasks were developed originally to represents proof situations in ΩMEGA's proof planner MULTI [17]. We extended tasks as a general technique for "natural" reasoning with abstract steps [14]. That is, the task framework allows for all kinds of steps with tasks ranging from formal steps like rewrite steps or definition expansion/contraction steps to abstract steps involving computations of external systems or merely sketching proof ideas and their flexible combination.

Proof tasks can be seen as sequents $\varphi_1, \ldots \varphi_n \vdash \psi_1, \ldots, \psi_m$ where there is always one formula—the so-called *focus*—annotated as the currently active one. The focus may be an antecedent or a succedent formula. For example, $\varphi_1, \ldots \varphi_n \vdash \underline{\psi_1}, \psi_2, \ldots, \psi_m$ describes a task where we have the context $\varphi_1, \ldots \varphi_n \vdash \psi_2, \ldots, \psi_m$ available for showing the focus $\vdash \psi_1$. In a user-interface tasks may be presented as

$$\frac{\varphi_1, \ldots \varphi_n}{\underline{\psi_1}, \ldots, \psi_m}$$

and use colors to further distinguish antecedent and succedent formulas, e.g. the negative formulas in red and the positive ones in black.

In the remainder of this section, we discuss the construction of a PDS with tasks for the example theorem "$\sqrt{12}$ is irrational". The general proof technique we shall apply to this problem works as follows: Given is the conjecture "$\sqrt[j]{l}$ is irrational". Assume that $\sqrt[j]{l}$ is rational. Then there are integers n, m, which have no common divisor and for which holds that $\sqrt[j]{l} = \frac{n}{m}$. Derive a contradiction to the assumption by showing that, indeed, n, m have a common divisor. Potential candidates for the common divisor are the prime factors of l.

3.1 Proof Construction in a $\mathcal{PDS}$

In a first step, we construct a PDS in the way a human mathematician would like to prove the given conjecture; see the proof sketch above. This is supported by so-called interactive island planning (see [23, 22] for details), a technique that expects an outline of the proof and has the user provide main subgoals, called *islands*. The details of the proof, eventually down to the logic level, are postponed. Hence, the user can write down *his* proof idea in a natural way with as many gaps as there are open at this first stage of the proof. Technically, in our framework the islands are tasks and all justifications between islands state `island`, i.e., they just indicate the intention that an island should follow from several other islands.

3.1.1 Step 0.

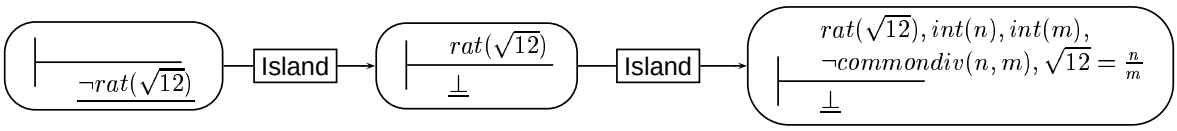
The proof starts with the initial task $\vdash \underline{\neg rat(\sqrt{12})}$ and the initial PDS show on the left of Figure 6

3.1.2 Step 1.

In the first step we introduce the indirect argument and reduce the initial task to $rat(\sqrt{12}) \vdash \underline{\bot}$ in which we assume that $rat(\sqrt{12})$ holds and the basic contradiction $\bot$ is to be proved. This action extends the PDS to one viewed on the right of Figure 6 where the justification Island states that the action introduces a new island node.

3.1.3 Step 2.

In the second step we derive from the assumption $rat(\sqrt{12})$ that there exist two integers n, m, which have no common divisors and for which $\sqrt{12} = \frac{n}{m}$ holds. This action further refines the PDS to



with the new task $rat(\sqrt{12}), int(n), int(m), \neg commondiv(n,m), \sqrt{12} = \frac{n}{m} \vdash \bot$

3.1.4 Step 3 + 4.

To complete the proof a common divisor is needed. Since 12 has the prime factors 2 and 3 there are two potential candidates. Moreover, for each candidate we have to show that both n and m are divided by it. This results in the PDS (shown below) with OR-branches (outgoing edges of PDS-nodes, e.g $\boxed{rat(\sqrt{12}), int(n), int(m), \neg commondiv(n,m), \sqrt{12} = \frac{n}{m} \vdash \underline{\bot}}$) and AND-branching (outgoing links of justification nodes, e.g. $\boxed{\text{Island}}$), where Σ abbreviates all so far available assumptions: $rat(\sqrt{12})$, $int(n)$, $int(m)$, $\neg commondiv(n,m)$, $\sqrt{12} = \frac{n}{m}$. To accomplish a proof we have now 2 possibilities: either we solve the two tasks $\Sigma \vdash div(n,3)$ and $\Sigma \vdash div(m,3)$, which demand to prove that both n and m have divisor 3, or we solve the two tasks $\Sigma \vdash div(n,2)$ and $\Sigma \vdash div(m,2)$, which demand to prove that both n and m have divisor 2.

3.1.5 Step 5 + . . .

We omit the further construction of the PDS in detail and just sketch the missing steps to derive a proof. We cannot show that both n and m have divisor 2 in the given context. Hence, the right branch of the PDS does not represent any progress. However, both n and m have divisor 3. From $\sqrt{12} = \frac{n}{m}$ follows that $m^2 * 12 = n^2$. Hence, n^2 has divisor 12 and thus also divisor 3. Then n also has divisor 3, since 3 is a prime number. This implies that $n = 3 * k$ for an integer k. Substituting n by $3 * k$ in the equation $m^2 * 12 = n^2$ results in $m^2 * 12 = 9 * k^2$. This equation can be simplified to $m^2 * 4 = 3 * k^2$. This implies that m^2 has divisor 3, from which follows that m

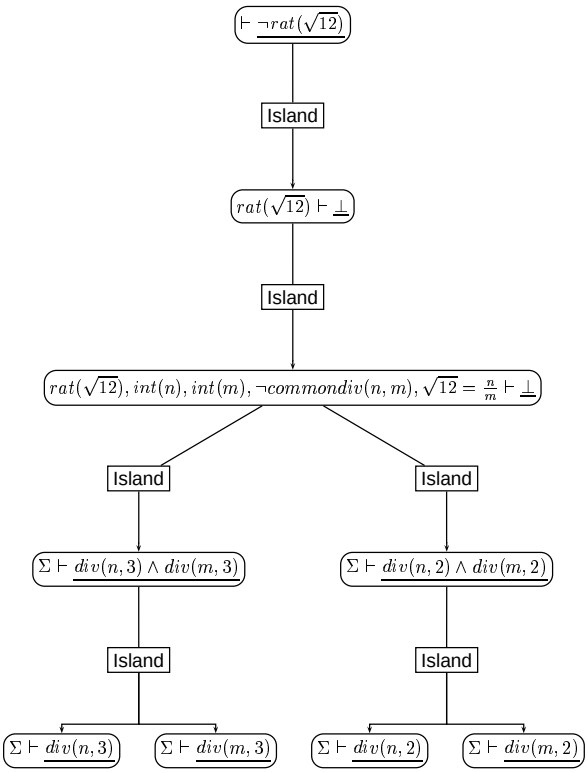


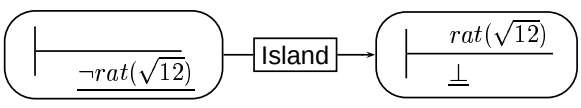
Figure 7: PDS after step $5 + \ldots$

has divisor 3 since 3 is a prime number. The introduction of all these steps closes the left branch, i.e. one alternative, of the last PDS and results in a closed PDS.

Note that a proof along this idea can also be automatically proof planned in ΩMEGA; for further details we refer to [22].

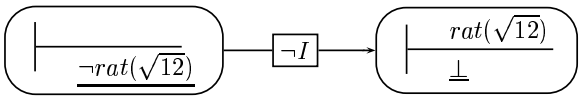
3.2 Proof Expansion in a PDS

So far, our proof has been developed and sketched only at an intuitive, abstract level and logical details have been neglected. Verification of this proof requires expanding it to a logic-calculus layer. How much "effort" this expansion causes and whether it succeeds depends on the island steps and the gaps they represent. In general, an island step can be arbitrarily difficult, so that each island step may again represent a proof problem in its own right. Nevertheless, the expansion can be supported by automated tools. For instance, automated theorem provers can try to solve subproblems, computer algebra systems can perform computations, and model generators can create counterexamples, which can point out missing facts in the proof. We omit a detailed discussion of automated expansion support here and refer the interested reader to [23] and [22]. Rather, we briefly discuss the expansion of two steps in our current example PDS and sketch the resulting extended and refined PDS.

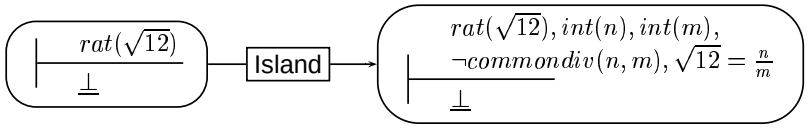


3.2.1 Expansion 1.

Consider the first step in the current PDS, which reduces the task $\vdash \underline{\neg rat(\sqrt{12})}$ to the task $rat(\sqrt{12}) \vdash \underline{\bot}$:



This step is already an instance of a proof step on calculus level. Indeed, it is a negation introduction step ($\neg I$). Hence, an expansion of this step simply results in a justification with $\neg I$ deriving $\vdash \underline{\neg rat(\sqrt{12})}$ from task $rat(\sqrt{12}) \vdash \bot$ by a calculus step. The resulting PDS fragment is shown above.



3.2.2 Expansion 2.

The second step in the proof reduced the task $rat(\sqrt{12}) \vdash \underline{\bot}$ to the task $rat(\sqrt{12}), int(n), int(m), \neg commondiv(n,m), \sqrt{12} = \frac{n}{m} \vdash \underline{\bot}$, which is represented by the PDS fragment above. This step implicitly encapsulates the application of the theorem that each rational number equals the fraction of two integers that have no common divisor. In the database of ΩMEGA this theorem is called *Rat-Criterion*:

$$Rat - Criterion ::= \forall x : Rat.\ \exists y, z : int.\ \left(\ x = \tfrac{y}{z} \wedge \neg commondiv(y,z)\ \right)$$

It says that for all rational x there exists integers y,z, which have no common divisor and furthermore $x = \frac{y}{z}$.

The expansion of the abstract step makes the application of the Rat-Criterion theorem explicit. This works as follows: The application of Rat-Criterion to the assumption $rat(\sqrt{12})$ in the task $rat(\sqrt{12}) \vdash \underline{\bot}$ derives the new assumption

$$\exists y, z : int.\ \left(\ \sqrt{12} = \tfrac{y}{z} \wedge \neg commondiv(y,z)\ \right)$$

which results in the corresponding new task

$$rat(\sqrt{12}),\ \exists y, z : int.\ \left(\ \sqrt{12} = \tfrac{y}{z} \wedge \neg commondiv(y,z)\ \right) \vdash \underline{\bot}$$

Decomposition of the composed new assumption then derives the task

$$rat(\sqrt{12}),\ int(n),\ int(m),\ \neg commondiv(n,m),\ \sqrt{12} = \frac{n}{m} \vdash \underline{\bot}$$

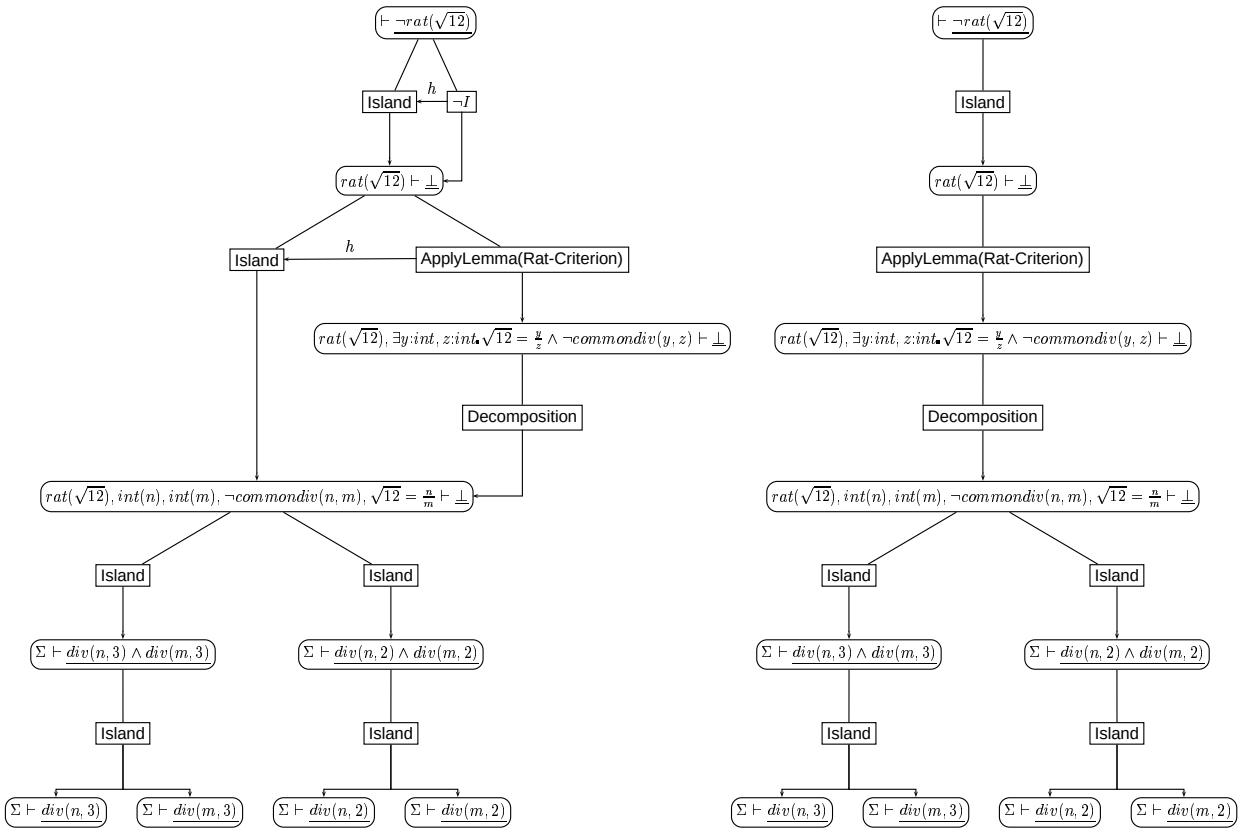
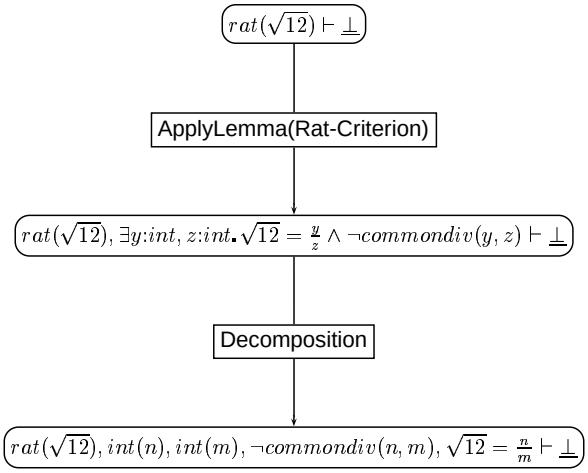


Figure 8: (Left) Complete PDS for the running example with alternative proof attempts and different layers of granularities. (Right) A possible PDS-View determined by selection of a set of alternatives for each PDS-node in the complete PDS.



Altogether the resulting expanded PDS fragment with a finer granularity has the form viewed on the right. Depending on the underlying basic calculus these steps either present already calculus steps or they can be further expanded. For instance, in the CORE system lemma application

is already a basic step.

As opposed thereto, in the old ΩMEGA system lemma applications have to be further expanded to derive Natural Deduction proofs.

The complete PDS maintaining simultaneously the initial abstract, less granular proof sketch and the lower, more granular verification of it is shown on the left-hand side in Figure 8. It also contains the hierarchical edges $.\overset{h}{\rightarrow}.$ which connect the different vertical layers. It supports four different PDS-views which result from alternative granularities of the outgoing justifications of the initial node $\vdash \neg rat(\sqrt{12})$ and of the outgoing justifications of the node $rat(\sqrt{12}) \vdash \bot$. Selecting the upper justification for the set of alternatives for the first node and the lower justification for the latter node results in the PDS-view shown on the right-hand side in Fig 8.

4 Implementation

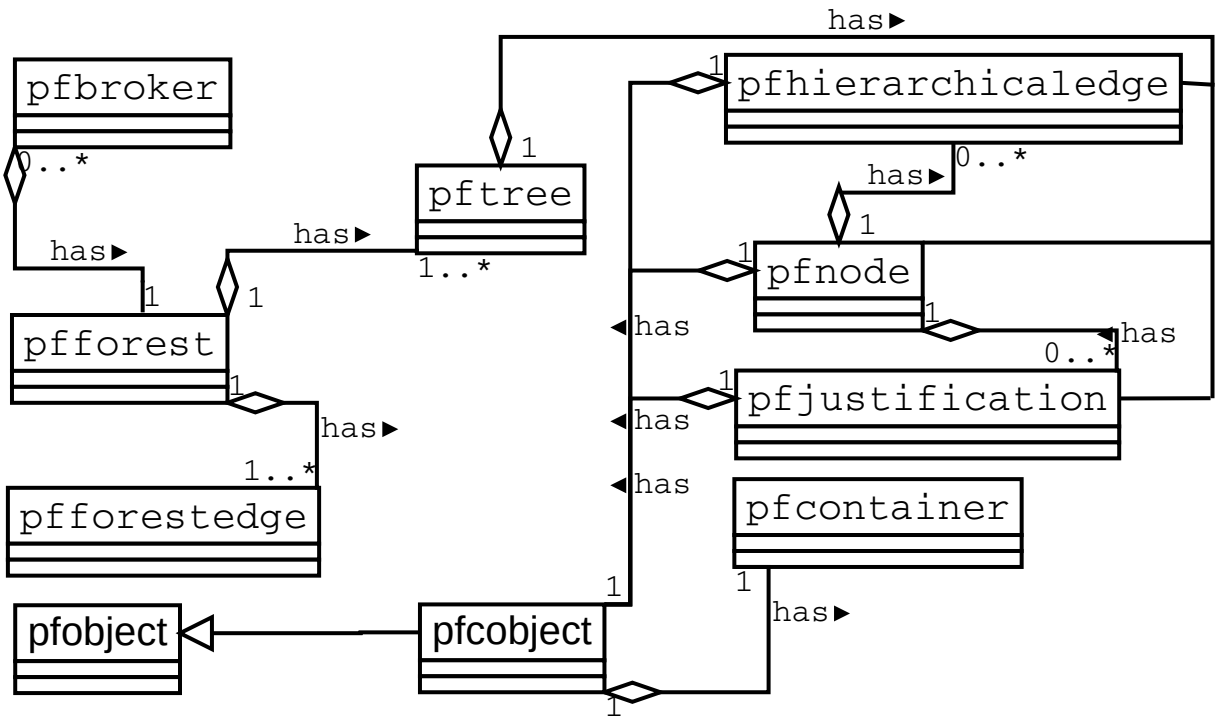


Figure 9: UML class diagram

This section describes the implementation of the theoretical model from Section 2, where all objects of the theoretical model such as nodes, justifications, hierarchical edges, trees, forests and multi forests, are implemented by corresponding classes. Most details are hidden from the user and we define an abstract interface, which comes close to the theoretical description. The user communicates with the $\mathcal{PDS}$-tree of forests via a simple object called `pfbroker`. The broker manages forests, which are based on $\mathcal{PDS}$-trees, justifications, hierarchical edges and nodes. Each $\mathcal{PDS}$-tree contains a $\mathcal{PDS}$-view. The communication between the user and the `pfbroker` is based on identifiers, which are assigned to all objects managed by the `pfbroker`. The overall class structure is shown in Figure 9.

We proceed as follows: In Section 4.1 we describe the top-level functions of the class `pf-broker`. Then we will introduce the identifier concept mentioned above, in Section 4.2. After defining an operational semantics for functions which insert objects in a $\mathcal{PDS}$-tree (Section 4.3), we will consider data import and export (Section 4.4) and how elements can be deleted from a $\mathcal{PDS}$-graph (Section 4.6).

4.1 The Interface of `pfbroker`

In this section we describe the top level functions of the interface, such as adding or deleting forests or trees, or inserting justifications. The UML diagram of the class is shown in Figure 10. The implementation language is Common Lisp [25].

4.1.1 `pf-broker_addforest`

The function `pf-broker_addforest` adds a new forest into the set of forests managed by the broker. This forest consists of a single tree with an open node with content $content$. `pf-broker_addforest` has the following arguments:

- the broker that manages all objects
- the content of the root node of the new inserted tree of the new forest

The function returns the identifier of the newly created tree.

Example:

```
;; generate broker
(setq broker (pf-broker_create))
;; add tree in nonexisting forest
(setq treeid (pf-broker_addforest broker
        (pf-containernode_create content)
        ))
```

4.1.2 `pf-broker_addpdstree`

The function `pf-broker_addpdstree` inserts a new $\mathcal{PDS}$-tree to the forest with the given *forest-id*. If the forest does not exist, a new forest with the next available forest-id is created. `pf-broker_addpdstree` has the following arguments:

- the broker that manages all objects
- the id of the forest, into which the tree will be inserted
- the content of the root-node of the tree that has to be inserted

Use `pf-forestid_generate` to generate any *forestid*.

Example:

pfbroker
-foreststore: forest list -timer: integer = 1 {timer >=0} +forestcounter: integer
+create() -newtime(broker:pfbroker): integer -newforestnumber(broker:pfbroker): pfforestid +clear(broker:pfbroker) +addforest(broker:pfbroker,content:pfcontainer): pfforestid +addpdstree(broker:pfbroker,fid:pfforestid,content:pfcontainer): pdstreeid +delete_forest(fid:pfforestid,broker:pfbroker): forestid +importforest(broker:pfbroker,stream): pfforestid +loadforest(broker:pfbroker,fid:pfforestid) +save_tree_dependent(broker:pfbroker,pdstree:pftreeid,outputstream): bool +selectjustification(broker:pfbroker,jd:pfjustificationid) +getopennodes(broker:pfbroker,treeid:pstreeid) +getrootnode(treeid:pftreeid): pfnode +getcontent(broker:pfbroker,object:pfcobject) +getlowerjustifications(broker:pfbroker,jid:pfjustificationid): pfjustification list +get_higherjustifications(broker:pfbroker,jid:pfjustificationid): pfjustification list +getselectedjustifications(broker:pfbroker,nid:pfnodeid): pfjustification list +isyounger(broker:pfbroker,o1:pfobject,o2:pfobject): {-1,1} +justify(b:pfbroker,c:con,parent:nid,ex_succ:[nid],nc:[con]): jid x [nid] +abstract_justification(b:pfbroker,jc:con,j:jid,ex_succ:[nid],nc:[con]): jid x [nid] +expand_justification(b:pfbroker,c:con,j:jid,ex_succ:[nid],nc:[con]): jid x [nid] -get_object(broker:pfbroker,jid:pfjustificationid): pfjustification -get_object(broker:pfbroker,hedgeid:hierarchicaledgeid): pfhierarchicaledge -get_object(broker:pfbroker,fid:pfforestid): pfforestid -get_object(broker:pfbroker,feid:pfforestedgeid): pfforestedge -get_object(broker:pfbroker,treeid:pftreeid): pftree -get_object(nodeid:pfnodeid): pfnode -getallforests(): forest list +prove_invariants(broker:pfbroker): bool +update_content(broker:pfbroker,oc:pfcobject) +addforestedge(b:pfbroker,parent:nid,ex_succid:[nid],nc:[con],c:con): pfforestedgeid +deletejustification(jid:pfjustificationid,broker:pfbroker) +deleteforestedge(feid:pfforestedgeid,broker:pfbroker) +exporttimexml(time:integer): xml*entity +print-object(broker:pfbroker,stream) +exportforestxml(broker:pfbroker,forestid:pfforestid): xml*entity

Figure 10: Pfbroker class

```
;; generate broker
(setq broker (pf-broker_create))
;; add tree in nonexisting forest
(setq treeid (pf-broker_addpdstree broker
                  (pf-forestid_generate 0)
                  (pf-containernode_create content)
                ))
```

4.1.3 pf-broker_justify

The function `pf-broker_justify` adds a justification into the tree that connects the source node with a set of successor nodes of that tree. The successor nodes may either already exist or may be generated. `pf-broker_justify` has the following arguments:

- the broker that manages objects
- the content for the justification
- the identifiers of already existing nodes the justification will be connected to
- the list of contents for successor nodes that will be created

Example:

```
(pf-broker_justify broker
  (pf-containerjustification_create 'Just1)
  parentid
  (list node1id, node2id)
  (list pf-containernode_create '1)
)
```

4.1.4 pf-broker_expand_justification

The function `pf-broker_abstract-justification` inserts a new justification into a tree and relates it to the specified justification by a hierarchical link. It has the following arguments:

- the broker that manages objects
- the content for the justification
- the identifier of the justification that will be abstracted
- the list of identifiers of already existing successors
- the list of contents for successor nodes that shall be created

Example:

```
(pf-broker_abstract-justification broker
  (pf-containerjustification_create '123)
  justificationid
  (list node1id node2id)
  (list (pf-containernode_create '1))
)
```

4.1.5 pf-broker_abstract_justification

The function `pf-broker_abstract-justification` inserts a new justification in a tree and relates it to the specified justification by a hierarchical link. Its arguments are

- the broker that manages objects
- the content for the new justification
- the identifier of the justification that shall be abstracted
- the list of identifiers of already existing successors
- the list of contents for successor nodes that shall be created

Example:

```
(pf-broker_expand-justification broker
  (pf-containerjustification_create '123)
  justificationid
  (list node1id node2id)
  (list (pf-containernode_create '1)))
```

4.1.6 pf-broker_delete_justification

The function `pf-broker_delete_justification` deletes a justification and recursively the subtree spanned by the justification, if it is independent to the rest of the tree (see Section 4.6 for details). It has the following arguments:

- the broker that manages objects
- the content for the new justification
- the identifier of the justification that shall be abstracted
- the list of identifiers of already existing successors
- the list of contents for successor nodes that shall be created

Example:

```
(pf-broker_delete_justification broker justificationid)
```

The deletion process is explained in detail in Section 4.6.

4.1.7 `pf-broker_selectjustification`

The function `pf-broker_selectjustification` selects the justification given by a justification-id, that is adds it to the set of alternatives of the current $\mathcal{PDS}$-view. This may result in removing other justifications from the set of alternatives in order to satisfy the adequacy and completeness property of the $\mathcal{PDS}$-view. The function has the following arguments:

- the broker that manages objects
- the identifier of the justification that shall be selected

Example:

```
(pf-broker_selectjustification broker justificationid)
```

4.2 Identifiers and the Projection Function

As mentioned above, an identifier assigns a name to each object. It is a design objective that communication with the broker only relies on identifiers. The identifiers have a hierarchical structure, with the following underlying idea:

Depending on the object type, identifiers are divided in 4 levels (compare Figure 11), where the lowest level (level 4) is given by *justification ids* and the highest level (level 1) by *forest ids* and lower levels inherit properties from the higher levels. For each level l, the identifier

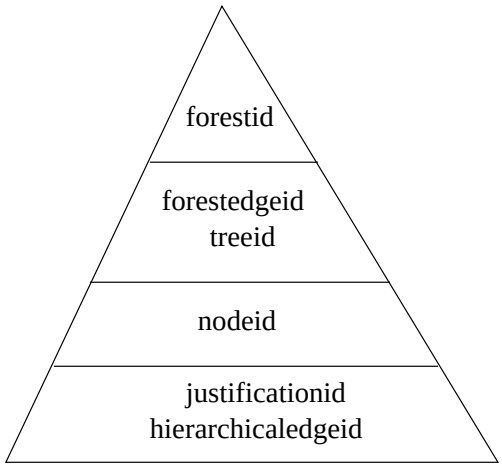


Figure 11: Hierarchical structure of identifiers

corresponding to is an element of $\mathbb{N}^{l-1} \times \mathbb{N}$ where $\mathbb{N}$ denotes the set of natural numbers. The $\mathbb{N}^{l-1}$-part is inherited by the level above. At the highest level, there is no level above, so forest ids consists of a single natural number.

One level below, we find tree ids. They consist of a tuple (m, n), where the m-part is determined by the level above in the following way: Suppose we have a tree t. This tree is in a forest f, and from this forest we get the first component, that is $m = id(f)$.

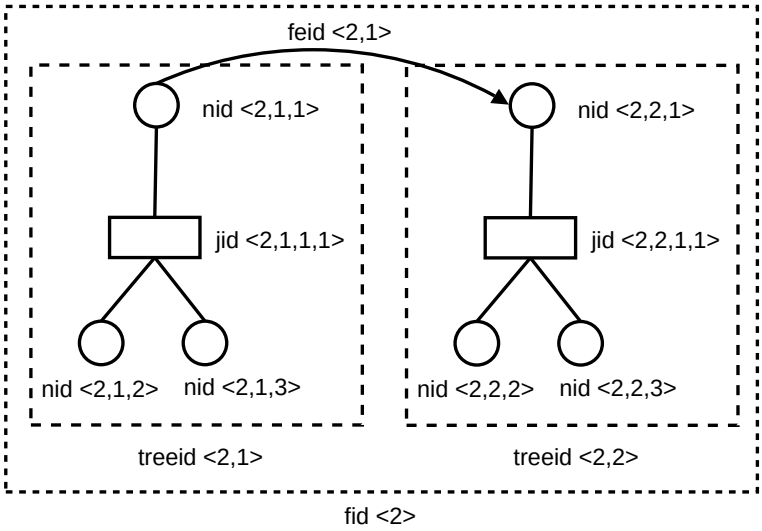


Figure 12: Forest, consisting of two trees

Consider the example in Figure 12. There is one forest, represented by the dotted box with id $\langle 2 \rangle$, consisting of two trees, represented by the dashed boxes inside the dotted box. As these trees are inside the forest, they extend the forest id to $\langle 2, 1 \rangle$ and $\langle 2, 2 \rangle$ respectively. This procedure goes on to the lower levels as shown in the figure.

Essentially, this concept has two benefits: On the one hand, it gives us a hashing function for memory access in a very natural way, on the other hand, given an id, the objects that are related to it at higher levels can easily be determined. The hierarchical identifiers are formalized in the following definition:

Definition 4.1 Let $\mathcal{F}$ denote the indexing set for multiple forests as introduced above, for each $f \in \mathcal{F}$, $\mathcal{T}_f$ represents the corresponding indexing set for the trees and $\mathcal{H}_f$ the corresponding indexing set of forest edges, $\mathcal{N}_{f,t}$ the set of nodes for a tree $t \in \mathcal{T}_f$, for each $n \in \mathcal{N}_{f,t}$ $J_{f,t}$ the indexing set for justifications and $\mathcal{H}_{f,t}$ the indexing set for hierarchical edges, then the identifier function, which assigns an identifier to an object is defined as follows:

- $id_{\mathcal{F}} : \mathcal{F} \to \mathbb{N}, \quad f \mapsto fnr$ where $fnr \in \mathbb{N}$ is a new forest number
- $id_{\mathcal{H}_f} : \mathcal{H}_f \to \mathbb{N}^2, \quad e \mapsto feid = (id(f), fenr)$ where $fenr \in \mathbb{N}$ is a new forest edge number
- $id_{\mathcal{T}} : \mathcal{T}_f \to \mathbb{N}^2, \quad t \mapsto tid = (id(f), tnr)$ where $tnr \in \mathbb{N}$ is a new tree number
- $id_{\mathcal{N}_{f,t}} : \mathcal{N}_{f,t} \to \mathbb{N}^3, \quad n \mapsto nid = (\pi_1(id(t)), \pi_2(id(t)), nnr)$ where $nnr \in \mathbb{N}$ such that nid is unique
- $id_{\mathcal{J}_{f,t}} : \mathcal{J}_{f,t} \to \mathbb{N}^4 \quad j \mapsto jid = (\pi_1(id(n)), \pi_2(id(n)), \pi_3(id(n)), jnr)$ where $n = source(j)$ and $jnr \in \mathbb{N}$ such that jid is unique
- $id_{f,t,\mathcal{H}} : \mathcal{H}_{f,t} \to \mathbb{N}^4 \quad h \mapsto hid = (\pi_1(id(n)), \pi_2(id(n)), \pi_3(id(n)), hnr)$ where $n = source(j)$ and $hnr \in \mathbb{N}$ such that hid is unique

Here π_1, π_2, π_3 denote the usual projection functions. As identifiers are unique, the inverse function exists. It is denoted by Θ

The UML-class diagram for the hierarchical identifier is shown in Figure 13. The definition above allows assigning an identifier to each object. For a forest, this identifier consists of a forest number, for a tree, of a tuple (forest number, tree number) indicating which number the tree has in the corresponding tree and so on. When it is clear from the context which function should be used we write just id.

Figure 13: UML-class diagram for ids

For notational convenience, we define the projection functions, which give us for each identifier the identifiers of the higher levels:

Definition 4.2 (Forest-projection) The forest-projection function π_f is defined as follows:

$$\begin{array}{rl} \pi_f : \mathbb{N} \cup \mathbb{N}^2 \cup \mathbb{N}^3 \cup \mathbb{N}^4 \rightarrow \mathbb{N} & \\ \pi_f(fnr) = fnr & fnr \in \mathbb{N} \\ \pi_f((fnr, tnr)) = fnr & (fnr, tnr) \in \mathbb{N}^2 \\ \pi_f((fnr, tnr, nnr)) = fnr & (fnr, tnr, nnr) \in \mathbb{N}^3 \\ \pi_f((fnr, tnr, nnr, jnr)) = fnr & (fnr, tnr, nnr, jnr) \in \mathbb{N}^4 \end{array}$$

Definition 4.3 (Tree-projection) The tree-projection function π_t is defined as follows:

$$\begin{array}{rl} \pi_t : \mathbb{N}^2 \cup \mathbb{N}^3 \cup \mathbb{N}^4 \rightarrow \mathbb{N} & \\ \pi_t((fnr, tnr)) = (fnr, tnr) & (fnr, tnr) \in \mathbb{N}^2 \\ \pi_t((fnr, tnr, nnr)) = (fnr, tnr) & (fnr, tnr, nnr) \in \mathbb{N}^3 \\ \pi_t((fnr, tnr, nnr, jnr)) = (fnr, tnr) & (fnr, tnr, nnr, jnr) \in \mathbb{N}^4 \end{array}$$

Definition 4.4 (Node-projection) The node-projection function π_n is defined as follows:

$$\begin{array}{rl} \pi_n : \mathbb{N}^3 \cup \mathbb{N}^4 \rightarrow \mathbb{N}^3 & \\ \pi_n(fnr, tnr, nnr)) = (fnr, tnr, nnr) & (fnr, tnr, nnr) \in \mathbb{N}^3 \\ \pi_n(fnr, tnr, nnr, jnr)) = (fnr, tnr, nnr, jnr) & (fnr, tnr, nnr, jnr) \in \mathbb{N}^4 \end{array}$$

Example: The justification in the left tree in Figure 12 has the id $\langle 2, 1, 1, 1 \rangle$. Using the projection functions defined above, we get $\pi_t \langle 2, 1, 1, 1 \rangle = \langle 2, 1 \rangle$ which is the id of the left tree.

For conversion of ids, the implementation provides the functions

- `pf-forestid_toforestid` to convert any id to a *forest id*
- `pf-treeid_totreeid` to convert any id except forest ids to a *tree id*
- `pf-nodeid_tonodeid` to convert a justification id or hierarchical edge id to a node id

Example:

```
(pf-forestid_toforestid nodeid) ;; get the corresponding forestid
```

4.3 Operational Semantics

In the following we define the operational semantics of the most important functions of the *pf-broker*. We will write

- $\mathcal{F}$ for the indexing set for the set of forests

- $\mathcal{H}_f$ for the indexing set for forest edges
- $\mathcal{T}$ for the indexing set of trees in one forest
- $\mathcal{N}$ for the indexing set of nodes of a given tree $\langle \mathcal{N}, \mathcal{J}, \mathcal{H} \rangle$
- $\mathcal{J}$ for the indexing set of justifications for a given tree $\langle \mathcal{N}, \mathcal{J}, \mathcal{H} \rangle$
- $\mathcal{H}$ for the indexing set of hierarchical edges for a given tree $\langle \mathcal{N}, \mathcal{J}, \mathcal{H} \rangle$
- c for a content
- b for a broker object
- n for a node
- $s_1, \ldots, s_l$ for successor nodes
- t for a timestamp
- id for an identifier
- j, j_i for justifications
- h for hierarchical edges

and distinguish between functions, the user is allowed to call (marked bold) and other functions that are hidden from the user. The former use identifiers rather than the concrete objects and modify multi-forest-views. In order to simplify the $\mathcal{PDS}$-view notion the set of alternatives $\mathcal{L}$ is kept implicit, that is the functions simply work on multi-forests $\mathcal{F}$. When the set of alternatives changes, the corresponding rule has an annotation.

Initially, there is no forest, so the indexing set $\mathcal{F}$ is empty. During the proof construction, forests can be generated by adding elements to $\mathcal{F}$ or deleted by deleting elements from $\mathcal{F}$. For a given forest $f \in \mathcal{F}$, trees can be manipulated by changing the corresponding indexing set for trees $\mathcal{T}_f$ or simply $\mathcal{T}$ if the forest is clear from the context. We denote the operation of adding an object o to a multi-forest, a forest or a tree with $\oplus$, which takes o as argument. The inverse operation is denoted by $\ominus$. We assume atomic functions `nodeid_create`, `hedgeid_create`, `justid_create`, `treeid_create`, `fedgeid_create`, `forestid_create` and `newtime` which we will use to create identifiers and timestamp for the objects. All of these functions are realized by simple counters.

4.3.1 Atomic Functions

We first define atomic expressions for the basic operations of creating a node, a justification, hierarchical edges, forests, and trees. They correspond to the creation of new objects of class *pfnode, pfhierarchicaledge* or *pfjustification, pfforest,* and *pftree,* respectively.

make_node: $make\_node(c, t, id)$

`make_node` creates a new node with content c, timestamp t and identifier id.

make_just: $make\_just(n, [s_1, \ldots, s_k], t, c, id)$

`make_just` creates a new justification with a content c, a timestamp t and an identifier id, which connects a node n with successor nodes $s_1, \ldots s_k$.

make_hedge: $make\_hedge(j_1, j_2, t, id)$

`make_hedge` creates a hierarchical edge that connects the justifications j_1 and j_2. It is assigned the identifier id and the timestamp t.

make_forestid: $make\_forest(id, t)$

`make_forest` creates a forest with identifier id and timestamp t.

make_tree: $make\_tree(id, t, n)$

`make_tree` creates a tree with identifier id, timestamp t and root node n.

make_fedge: $make\_fedge(n, [s_1, \ldots, s_k], c.t, id)$

`make_fedge` creates a new forest edge, which has one parent node n, a list of successors $s_1, \ldots, s_k$, content c, timestamp t and identifier id. The successors don't need to be in the same tree, but do need to be in the same forest.

4.3.2 Constructor Functions

For each class the implementation provides a constructor function, which essentially computes the arguments of the corresponding make function. This ensures that all parameters are properly specified and that all objects of the same kind are generated equally.

node_create: `node_create` takes a broker b (for the time management), a tree u (to generate the corresponding identifier) and the content c for the new node as arguments and produces a new node n.

$$\frac{newtime(b) \to t \quad nodeid\_create(u) \to nid \quad make\_node(c, t, nid) \to n}{node\_create(b, u, c) \to n}$$

just_create: `just_create` takes as arguments a broker b, a content c for the justification, the parent node of the justification and a list of child nodes. The result is a new justification j.

$$\frac{\begin{array}{l} newtime(b) \to t \\ justid\_create(n) \to id \quad make\_just(n, [s_1, \ldots, s_k], t, c, id) \to j \end{array}}{just\_create(b, c, n, [s_1, \ldots, s_k]) \to j}$$

The other constructor functions work analogously.

hedge_create:

$$\frac{\begin{array}{l} newtime(b) \to t \\ hedgeid\_create(source(j_1)) \to id \quad make\_hedge(j_1, j_2, t, id) \to h \end{array}}{hedge\_create(b, j_1, j_2) \to h}$$

fedge_create:

$$\frac{\begin{array}{c} newtime(b) \rightarrow t \\ fedgeid\_create(\Theta(\pi_f(idn))) \rightarrow id \\ make\_fedge(n, [s_1, \ldots, s_k], c, t, id) \rightarrow f \end{array}}{fedge\_create(b, n, [s_1, \ldots, s_k], c) \rightarrow f}$$

tree_create:

$$\frac{make\_tree(b, id, \emptyset) \rightarrow t \quad node\_create(b, t, c) \rightarrow n \quad setroot(t, n)}{tree\_create(b, f, c) \rightarrow t}$$

forest_create:

$$\frac{make\_forest(b) \rightarrow f \quad tree\_create(b, f, c) \rightarrow t}{forest\_create(b, c) \rightarrow (f \oplus t)}$$

4.3.3 Adding Forests

Having defined the basic operations, we are now able to define more complex operations that are more convenient for the user.

addforest:

$$\frac{forest\_create(b, c) \rightarrow f}{\mathbf{addforest}(\mathbf{b}, \mathbf{c}, \mathcal{F}) \rightarrow \mathcal{F} \oplus f}$$

`addforest` creates a new forest, which contains one tree, whose root node has content c.

addpdstree: Adding a new forest can also be done by the function *addpdstree*. It takes as arguments a broker b, a forest id fid, a content c and the current multi-forest $\mathcal{F}$ and returns a new multi forest $\mathcal{F}'$. If a forest with identifier fid exists, then a new tree is inserted into that forest, else a new forest (with possibly another identifier) is created.

$$\frac{addforest(b, c, \mathcal{F}) \rightarrow \mathcal{F}'}{\mathbf{addpdstree}(\mathbf{b}, \mathbf{fid}, \mathbf{c}, \mathcal{F}) \rightarrow \mathcal{F}'} \exists f \in \mathcal{F}.id(f) = fid$$

$$\frac{tree\_create(b, \Theta(fid), c) \Rightarrow f}{\mathbf{addpdstree}(\mathbf{b}, \mathbf{fid}, \mathbf{c}, \mathcal{F}) \rightarrow \mathcal{F} \oplus f} \forall f \in \mathcal{F} id(f) \neq fid$$

4.3.4 Inserting a Justification

insert_just: Aiming at the definition of *justify*, *expand_just*[3] and *abstract_just*[4], we first define the function *insert_just*. *insert_just* creates a justification and which connects a parent node n with successor nodes $s_1, \ldots, s_k$ and nodes $n_1, \ldots, n_l$. $s_1, \ldots, s_k$ must already exist, whereas $n_1, \ldots, n_l$ are newly created nodes with contents $c_1, \ldots, c_k$.

$$\frac{\begin{array}{rcl} node\_create(b, \Theta(id(n)), c_1) & \rightarrow & n_1 \\ & \vdots & \\ node\_create(b, \Theta(id(n)), c_l) & \rightarrow & n_l \\ just\_create(b, c, n, [s_1, \ldots, s_k, n_1, \ldots, n_l]) & \rightarrow & j \end{array}}{insert\_just(b, c, n, [s_1, \ldots, s_k], [c_1, \ldots, c_l]) \rightarrow \langle j, [n_1, \ldots, n_l] \rangle}$$

[3] shorthand for *expand_justification*

[4] short for *abstract_justification*

We now use the functions defined above to define the function *justify* which returns a tree:

justify (internal version):

$$\frac{\begin{array}{rcl}\Theta(\pi_t(id(n))) & = & \langle \mathcal{N}, \mathcal{J}, \mathcal{H} \rangle \\ insert\_just(b, c, n, [s_1, \ldots, s_k], [c_1, \ldots, c_k]) & \rightarrow & \langle j, [n_1, \ldots, n_l] \rangle \end{array}}{\begin{array}{c} justify(b, c, n, [s_1, \ldots, s_k], [c_1, \ldots, c_n]) \\ \rightarrow \langle \mathcal{N} \oplus \{n_1, \ldots, n_l\}, \mathcal{J} \oplus j, \mathcal{H} \rangle \end{array}} \; L_n := L_n \oplus j$$

The tree represents the updated version of the tree after the justify function. The nodes, which have been created by the *insert_justify* function, are added to the indexing set of nodes and the new justification is added. The projection function is used to determine the tree before the modification. The side condition ensures that the newly created justification is added to the list of alternatives.

We now define the interface function which acts on identifiers rather than on concrete objects:

justify (interface version):

$$\frac{\begin{array}{rcl}\Theta(\pi_f(nid)) & = & \mathcal{T} \\ \Theta(\pi_t(nid)) & = & t \\ \Theta(sid_1) & \rightarrow & s_1 \\ & \vdots & \\ \Theta(sid_n) & \rightarrow & s_n \\ \Theta(nid) & \rightarrow & n \\ justify(b, c, n, [s_1, \ldots, s_k], [c_1, \ldots, c_k]) & \rightarrow & t' \end{array}}{\mathbf{justify}(\mathbf{b}, \mathbf{c}, \mathbf{nid}, [\mathbf{sid_1}, \ldots, \mathbf{sid_m}], [\mathbf{c_1}, \ldots, \mathbf{c_k}], \mathcal{F}) \rightarrow \mathcal{F}'} \text{ where } \begin{array}{c} \mathcal{T}' = (\mathcal{T} \ominus t) \oplus t' \\ \mathcal{F}' = (\mathcal{F} \ominus \mathcal{T}) \oplus \mathcal{T}' \end{array}$$

It takes a broker b, the content of the new justification c, the id of the parent node of the justification, the ids of the successor nodes of the justification which already exist, the contents for the successor nodes of the justification which have to be created and a multi-forest $\mathcal{F}$ as argument and returns a multi-forest $\mathcal{F}'$, where the tree t in forest $\mathcal{T}$ is replaced by the tree t'.

4.3.5 Expanding and Abstracting a Justification

`expand_justification` essentially does the same as justify: it inserts a justification j_2 in a tree. But additionally it takes a justification j_1 as argument which is set to be more abstract as the justification j_2.In other words, a new hierarchical edge h is created with $j_1 \xrightarrow{h} j_2$.

There are two cases two consider, one where the new justification doesn't need to be added to the list of alternatives and one where it must (see Figure 14 for an example): The new justification j_1 must not be added to the set of alternatives if the justification j_2 or a more abstract realization j of j_2 is in the set of alternatives, as then $j_1 < j_2$ or $j_1 < j$ and one of j or j_2 is selected.

expand_just (case 1):

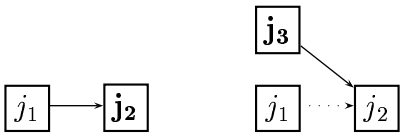


Figure 14: Case split for inserting justification j_1, selected justifications are marked bold

$$\frac{\begin{array}{rcl}\Theta(\pi_t(id(n))) & = & \langle \mathcal{N}, \mathcal{J}, \mathcal{H} \rangle \\ hedge\_create(b, j_1, j_2) & \rightarrow & h \\ insert\_just(b, c, n, [s_1, \ldots, s_k], [c_1, \ldots, c_k]) & \rightarrow & \langle j_2, [n_1, \ldots, n_l] \rangle \end{array}}{\begin{array}{c} expand\_just(b, c, j_1, [s_1, \ldots, s_n], [c_1, \ldots, c_n], t) \\ \rightarrow \langle \mathcal{N} \oplus \{n_1, \ldots, n_l\}, \mathcal{J} \oplus j, \mathcal{H} \oplus h \rangle \end{array}} \exists j \in L_n . j \sim j_2$$

Either j_2 or a more abstract realization of j_2 is in the set of alternatives, hence j_1 is not allowed to be in the set of alternatives.

expand_just (case 2):

$$\frac{\begin{array}{rcl}\Theta(\pi_t(id(n))) & = & \langle \mathcal{N}, \mathcal{J}, \mathcal{H} \rangle \\ hedge\_create(b, j_1, j_2) & \rightarrow & h \\ insert\_just(b, c, n, [s_1, \ldots, s_k], [c_1, \ldots, c_k]) & \rightarrow & \langle j_2, [n_1, \ldots, n_l] \rangle \end{array}}{\begin{array}{c} expand\_just(b, c, j_1, [s_1, \ldots, s_n], [c_1, \ldots, c_n], t) \\ \rightarrow \langle \mathcal{N} \oplus \{n_1, \ldots, n_l\}, \mathcal{J} \oplus j, \mathcal{H} \oplus h \rangle \end{array}} \forall j \in L_n . j \not\sim j_2$$

where $L_n := L_n \oplus j_2$.

As there is a justification reachable from j_1 via hierarchical edges, j_1 must be added to the set of alternatives. The interface version is defined as follows:

expand_just (interface version):

$$\frac{\begin{array}{rcl}\Theta(\pi_f(jid)) & \rightarrow & \mathcal{T} \\ \Theta(\pi_t(jid)) & \rightarrow & t \\ \Theta(sid_1) & \rightarrow & s_1 \\ & \vdots & \\ \Theta(sid_k) & \rightarrow & s_k \\ \Theta(jid) & \rightarrow & j \\ expand\_just(c, j, [s_1, \ldots, s_k], [c_1, \ldots, c_l]) & \rightarrow & t' \end{array}}{\mathbf{expand\_just}(\mathbf{c}, \mathbf{jid}, [\mathbf{sid_1}, \ldots, \mathbf{sid_m}], [\mathbf{c_1}, \ldots, \mathbf{c_n}], \mathcal{F}) \rightarrow \mathcal{F}'}$$

where $\pi_t(sid_1) = \ldots = \pi_t(sid_k)$
$\pi_t(sid_1) = \pi_t(jid)$
$\mathcal{T}' = \mathcal{T} \ominus t \oplus t'$
$\mathcal{F}' = \mathcal{F} \ominus \mathcal{T} \oplus \mathcal{T}'$

The side-condition ensures that the justification j_1 and all successor nodes $s_1, \ldots, s_k$ of the new justification, which already exists, are all contained in one tree. Furthermore the old tree t has to be replaced by the new tree t' in the multi-forest.

`abstract_just` works exactly as `expand_just` except that the inserted hierarchical edge h is of opposite direction: $j_2 \xrightarrow{h} j_1$. The corresponding rules are shown below:

abstract_just (case 1):

$$\frac{\begin{array}{rcl}\Theta(\pi_t(id(n))) & = & \langle \mathcal{N}, \mathcal{J}, \mathcal{H} \rangle \\ hedge\_create(b, j_2, j_1) & \rightarrow & h \\ insert\_just(b, c, n, [s_1, \ldots, s_k], [c_1, \ldots, c_k]) & \rightarrow & \langle j_2, [n_1, \ldots, n_l] \rangle \end{array}}{\begin{array}{c} abstract\_just(b, c, j_1, [s_1, \ldots, s_n], [c_1, \ldots, c_n], t) \\ \rightarrow \langle \mathcal{N} \oplus \{n_1, \ldots, n_l\}, \mathcal{J} \oplus j, \mathcal{H} \oplus h \rangle \end{array}} \; \exists j \in L_n.j \sim j_2$$

abstract_just (case 2):

$$\frac{\begin{array}{rcl}\Theta(\pi_t(id(n))) & = & \langle \mathcal{N}, \mathcal{J}, \mathcal{H} \rangle \\ hedge\_create(b, j_2, j_1) & \rightarrow & h \\ insert\_just(b, c, n, [s_1, \ldots, s_k], [c_1, \ldots, c_k]) & \rightarrow & \langle j_2, [n_1, \ldots, n_l] \rangle \end{array}}{\begin{array}{c} abstract\_just(b, c, j_1, [s_1, \ldots, s_n], [c_1, \ldots, c_n], t) \\ \rightarrow \langle \mathcal{N} \oplus \{n_1, \ldots, n_l\}, \mathcal{J} \oplus j, \mathcal{H} \oplus h \rangle \end{array}} \; \forall j \in L_n.j \not\sim j_2$$

where $L_n := L_n \oplus j_2$.

abstract_just (interface version):

$$\frac{\begin{array}{rcl}\Theta(\pi_f(jid)) & \rightarrow & \mathcal{T} \\ \Theta(\pi_t(jid)) & \rightarrow & t \\ \Theta(sid_1) & \rightarrow & s_1 \\ & \vdots & \\ \Theta(sid_k) & \rightarrow & s_k \\ \Theta(jid) & \rightarrow & j \\ abstract\_just(c, j, [s_1, \ldots, s_k], [c_1, \ldots, c_l]) & \rightarrow & t' \end{array}}{\mathbf{abstract\_just}(\mathbf{c}, \mathbf{jid}, [\mathbf{sid_1}, \ldots, \mathbf{sid_m}], [\mathbf{c_1}, \ldots, \mathbf{c_n}], \mathcal{F}) \rightarrow \mathcal{F}'}$$

where $\pi_t(sid_1) = \ldots = \pi_t(sid_k)$
$\pi_t(sid_1) = \pi_t(jid)$
$\mathcal{T}' = \mathcal{T} \ominus t \oplus t'$
$\mathcal{F}' = \mathcal{F} \ominus \mathcal{T} \oplus \mathcal{T}'$

4.4 Import/Export of Data

This section describes the $\mathcal{PDS}$-data format that enables to import or to export data in/from a $\mathcal{PDS}$-forest. Given a forest, the user wants to

(i) export a single tree t

(ii) export a single tree t and all trees that are connected to t via hierarchical edges

(iii) import a set of trees into a new forest

(iv) import a set of trees into an existing forest

To this end we provide an XML-data format (see Appendix B, Figure 22), as it is system independent and verifiable by external tools. This format has to be able to store all information, which means the $\mathcal{PDS}$-view and the structure of the trees.

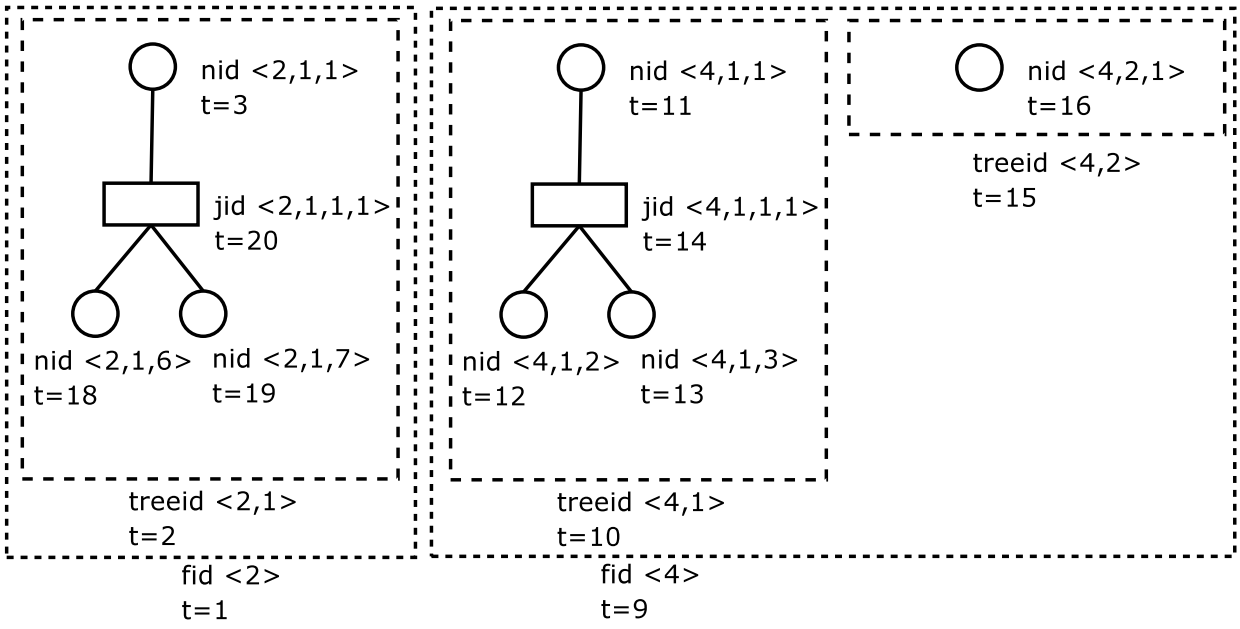


Figure 15: Tree to be imported and target forest before import

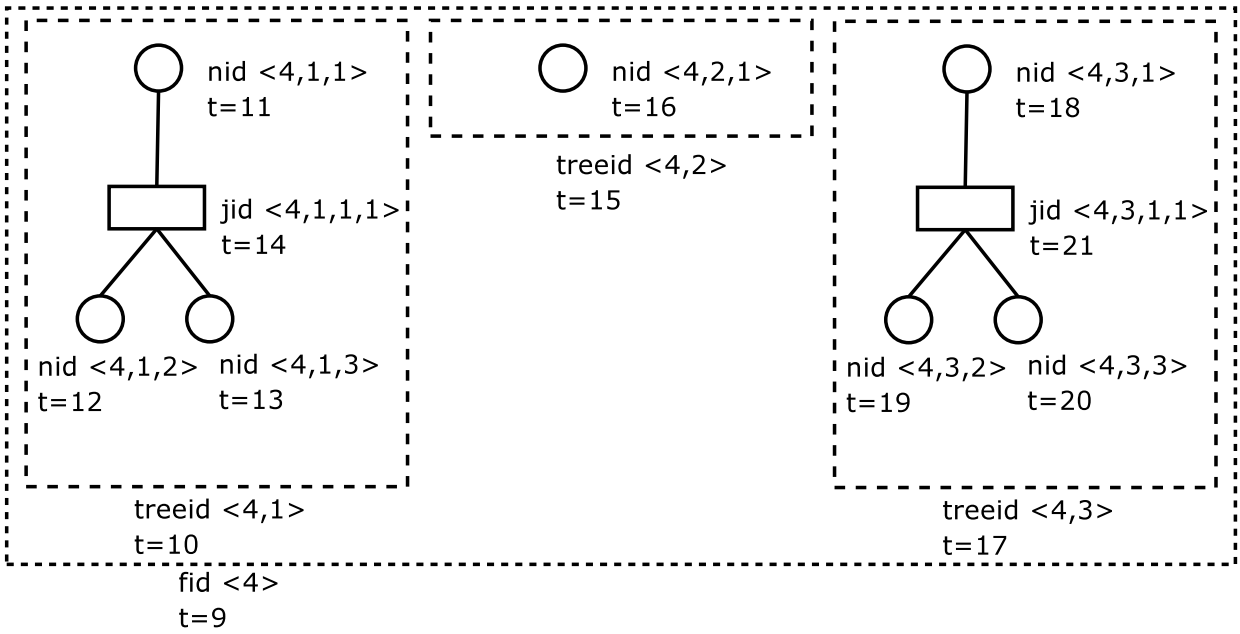


Figure 16: Forest after importing tree

The $\mathcal{PDS}$-view is completely determined by the selected justifications and hence simple to save. The data-format provides an attribute *selected* for each justification, which is set to 1 if the given justification corresponds to the set of alternatives and otherwise 0.

A set of trees is completely determined by the objects they contain, which are nodes, justifications, hierarchical edges, and connections between them, namely forest edges. All these objects are annotated with a time information, have an identifier and some specific content. As the time information and the identifier depend on all other objects in the $\mathcal{PDS}$ and possibly other forests or trees, it is not possible to store the identifiers and time information, as they may become invalid when they are imported.

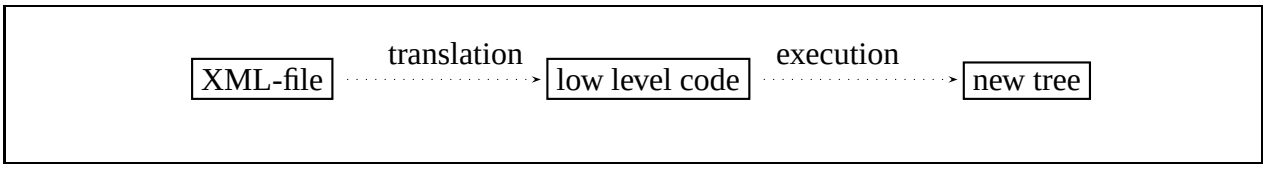


Figure 17: Importing a tree

In order to get consistent timestamps and identifiers, they are regenerated during the import process, such that they respect the time ordering of the forest they were exported from and fit in the new situation. That means if two objects o_1 and o_2 are exported and $time(o_1) < time(o_2)$, then this property should be preserved without sticking to the exact values of their timestamps. Furthermore there should be no object o that has been created before the import process with $time(o) > time(o_1)$.

So we store the time information in the XML-file, but we use it later only to get the order in which the objects have been created. When loading the objects again, this order is used to recreate the objects by the constructor functions. They calculate the new identifier and timestamp and take into consideration all other objects which already exist. In this way, the imported tree gets exactly the identifiers it would get if created from scratch and there is no difference between loading a tree and stepwise creating the same tree by the user.

To export the unknown content, the user has to provide specific import and export functions, that build a concrete realization of the abstract class *pfcontainer*.

Example: Figure 15 shows on the left hand side a forest containing a tree which shall be imported into the forest on the right hand side. Due to some some previous delete operations, there is no object with timestamp 4 to 17 in the left tree.

Figure 16 shows the result of the import process: The tree gets new identifiers and new timestamps. The gap described above between timestamps is closed.

4.5 $\mathcal{PDS}$-Visualization

To visualize a $\mathcal{PDS}$, we support the export of data to Wilma[5], a graph visualization tool (see Figure 18), which provides an intuitive presentation of the proof and how a proof is related to subproofs.

[5] `http://www.wilmascope.org/`

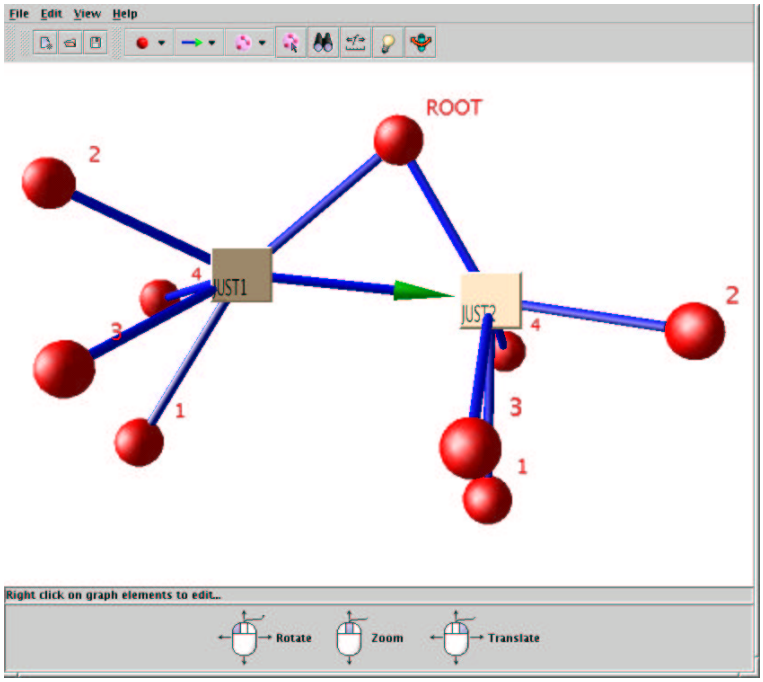


Figure 18: Wilma representation of a pds

4.6 Deleting Justifications

This section describes how justifications can be deleted from a $\mathcal{PDS}$. If a justification j is deleted, we also have to delete justifications that represent direct abstractions or expansions of the given justification, i.e. that are connected to the given justification by a hierarchical edge, if they were introduced after j.

Definition 4.5 (h-connected justification) Let $\langle \mathcal{N}, \mathcal{J}, \mathcal{H} \rangle$ be a $\mathcal{PDS}$ and $j_1, j_2 \in \mathcal{J}$. Then j_1, j_2 are *h-connected*, written $j_1 \overset{h}{\leftrightarrow} j_2$, iff

$$j_1 \overset{h}{\leftrightarrow} j_2 :\Leftrightarrow \exists h \in \mathcal{H}.j_1 \overset{h}{\rightarrow} j_2 \vee j_2 \overset{h}{\rightarrow} j_1$$

Definition 4.6 Let $S = \langle \mathcal{N}, \mathcal{J}, \mathcal{H} \rangle$ be an $\mathcal{PDS}$-graph, $j_1, j_2 \in \mathcal{J}$ be justifications. j_1 is called *dependent* from j_2 iff

$$\exists h \in \mathcal{H} \text{ with } j_m \overset{h}{\leftrightarrow} j_2 \text{ and } time(j_2) < time(j_1)$$

Deleting a justification j_1 is now formulated as follows:

1. Delete the marked justification j_1
2. For all sub-nodes n: if $|I_n| = 1$ then delete the node n, else do nothing
3. For all $j \in \mathcal{J}$ if j depends from j_m then delete j
4. For all hierarchical edges h: if $source(h) = j_m$ or $target(h) = j_m$ then delete h

where I_n denotes the set of incoming links of node n. Note that deleting nodes consists also of deleting all outgoing justifications. Normally, the user deletes a justification j when it is clear that the proof step represented by j doesn't lead to a solution. In order to analyze the failure, he may want to store the deleted proof steps. To enable this, `delete_justification` returns a XML-description of the objects that have been deleted, which are nodes, justifications and hierarchical edges. Generally, these elements don't build a $\mathcal{PDS}$, not even a subgraph of the tree where the deletion has taken place. This is because justifications or hierarchical edges may be deleted, but the objects they connect cannot.

This loss can in some way be repaired by adding exactly these objects to a list, but as they are not deleted, they have to be specially marked. In our implementation, they are collected in the entity *assume*.

We give an illustrating example for a complete deletion process.

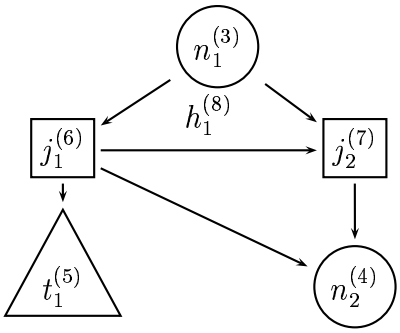


Figure 19: Example $\mathcal{PDS}$ before deleting operation

Example: In Figure 19 we see a $\mathcal{PDS}$-tree that consists of nodes n_1, n_2, two justifications j_1 and j_2, a hierarchical edge h_1 and a subtree t_1, whose exact structure doesn't matter for our purposes. Each of these elements is annotated with the time, at which they have been created, in square brackets. This allows to reconstruct the order in which the objects have been created: First n_1, then t_1, j_1, n_2, j_2, h_1. Note that the justification is older than the elements it connects, as the function `pf-broker_justify` first generates the objects it shall connect and then the connection. The same holds for the hierarchical edges.

In this example, there are only two candidates for justifications, namely j_1 and j_2 which can be deleted. The hierarchical edge h_1 connects them, so they are h-connected, i.e. $j_1 \overset{h}{\leftrightarrow} j_2$. As $time(j_2) > time(j_1)$ j_2 is dependent on j_1, but j_1 is not dependent onj_2. So, deleting j_1 results in deleting j_2 as well, but j_2 can be deleted independently from j_1.

The results of deleting j_1 or j_2 are shown in Figure 4.6 and 4.6 respectively. The result of

Figure 20: Example $\mathcal{PDS}$ after deleting j_1

deleting j_2 is shown below (where the content of the objects where simple numbers):

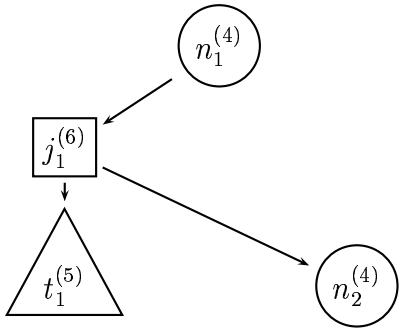


Figure 21: Example $\mathcal{PDS}$ after deleting j_2

5 Conclusion

This paper describes the PDS, a new generic proof data structure, which originates from and extends the successful datastructures of systems so different as the old ΩMEGA (deductive higher-order theorem prover with strong automatization on planning level) and QUODLIBET (inductive first-order clausal theorem prover with powerful automatization on calculus level). Among its key features are:

- the representation of alternative proof steps for *both* the reduction of a goal as well as for the expansion of a complex proof step to lower granularity
- the structuring of proof parts (i.e. lemmatization) into separate but connected parts of the data structure
- the generic representation of proof statements and justifications, biased neither to any specific calculus nor to any specific formalism for representing abstract proof plans

The explicit introduction of hierarchical levels within one data structure supports the bridging between intuitive, abstract level proof development, proof explanation and proof verification. Whereas proofs are typically developed and presented at an abstract and intuitive level, proof verification typically requires some underlying calculus at a very fine granularity. The PDS provides, for instance, the flexibility to perform alternative expansions of some abstract proof steps to represent the same proof idea in different underlying calculi. Maintaining simultaneously the proofs with different granularities accommodates, for instance, proof explanation systems, which can start with a presentation of the high-level proof, and on-demand generate presentations for expansions of *some* chosen proof steps [11]. Furthermore, the hierarchies represent the parts of the search space taken by automatic proof techniques, like for instance proof planning methods, tactics, and methodicals. Representing the search space as well as explored alternatives to represent the branches of the search space is well suited for debugging new proof techniques [10].

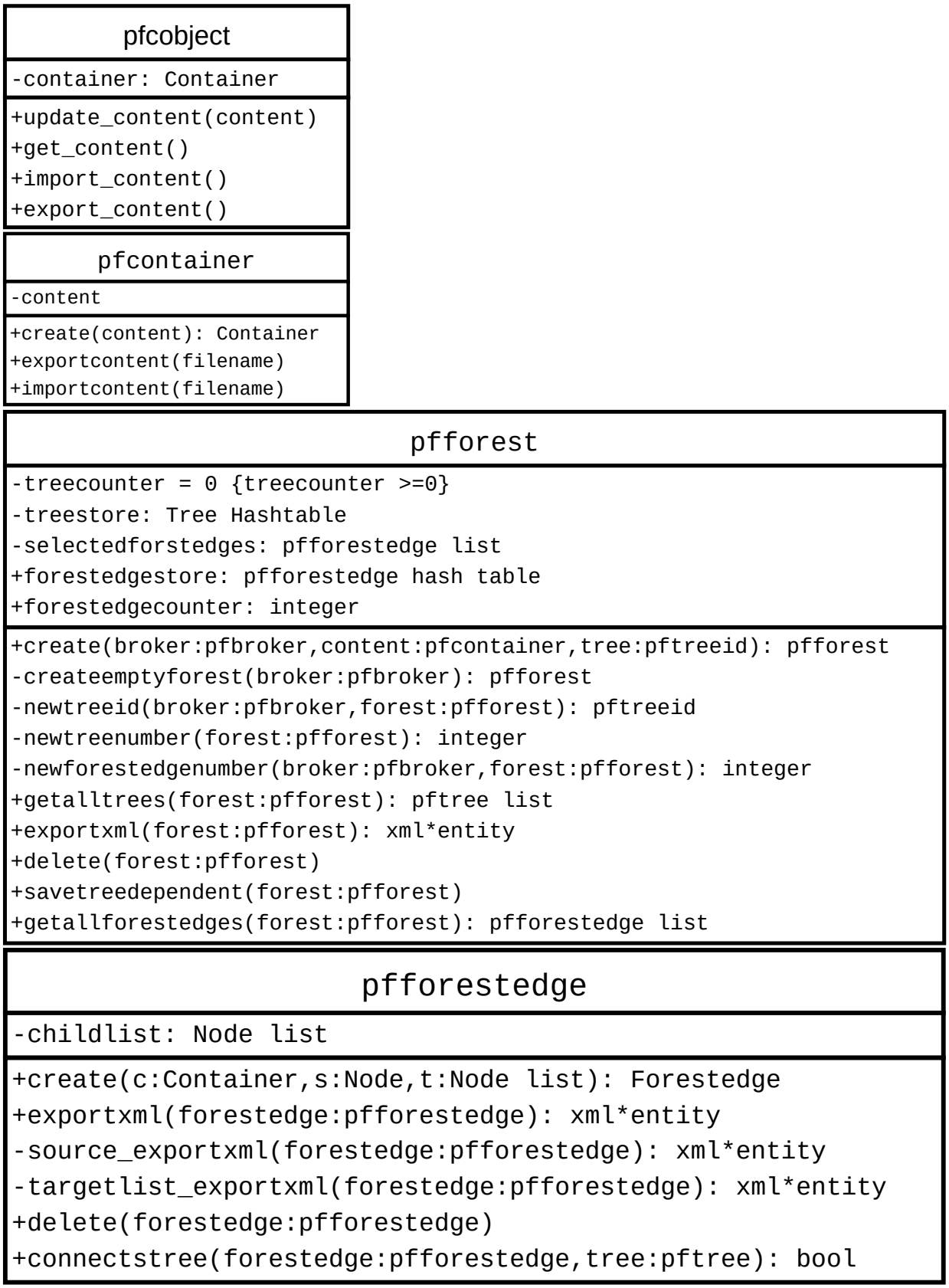
The PDS provides a flexible and general framework for storing and representing proofs under construction. However, the proof manipulations and refinements manipulating the PDS have to be determined and controlled by the proof system making use of the PDS. This proof system has to handle and control operations such as backtracking, instantiation of variables, collection of constraints etc. Moreover, it has to decide about whether to allow and how to realize features such as local definitions and cyclic structures in the PDS.[6]

[6]Technically, cyclic structures can be realized by cyclic lemmas in PDS forests, [26].

Many proof assistants actually provide proof data structures, e.g. COQ [8], INKA [15], ISABELLE [19], NUPRL [16], TPS [1], and VSE [5] to mentioned only a few. However, to the best of our knowledge none of them has been designed to support both a horizontal and vertical representation mechanism for proofs as presented in this paper.

We implemented the generic PDS described in this paper in Common Lisp and defined a content independent XML format for exporting and importing forests, trees, or parts of them. Furthermore, we are able to visualize our three-dimensional graphs. For the interaction with the user, however, PDS-views are essential.

A UML-Diagrams




pfhierarchicaledge
-source: Node
-target
+container: Container
+create(source:Node,target:Node list)
+deleteinhedge(broker:pfbroker,hedge:pfhierarchicaledge)
+deleteouthedge(broker:pfbroker,hedge:pfhierarchicaledge)
+exportxml(hedge:pfhierarchicaledge): xml*entity
-source_exportxml(hedge:pfhierarchicaledge): xml*entity
-target_exportxml(hedge:pfhierarchicaledge): xml*entity
pfjustification
-childlist: Node list
-outhedges: hierarchicaledge list
+create(broker:pfbroker,content:pfcontainer,parentnode:pfnode,children:node list): Link
+isexpansion(j1:pfjustification,j2:pfjustification): bool
+isabstraction(j1:pfjustification,j2:pfjustification): bool
+exportxml(justification:pfjustification): xml*entity
-source_exportxml(justification:pfjustification): xml*entity*
+targetlist_exportxml(justification:pfjustification): xml*entity
+delete(justification:pfjustification)
+print-object(justification:pfjustification,stream)
+select(justification:pfjustification)
pfnode
-container: Container
-alternativelist: Justification list
-hedgecounter
-justificationcounter
-hedges: pfhierarchicaledge hashtable
-outedges: pfjustification hashtable
+create(content): Node
+proveinvariants(node:pfnode): bool
+newjustificationnumber(node:pfnode): integer
+newhierarchicaledgenumber(node:pfnode): integer
+delete(node:pfnode)
+remove(node:pfnode)
+getallhierarchicaledges(node:pfnode): pfhierarchicaledge list
+getalljustifications(node:pfnode): pfjustification list
-adequacytest(alternativelist:pfnode list): bool
-completenesstest(outedgelist:pfnode list,alternativelist:pfnode list): bool
+insertjustification(b:pfbroker,c:pfcontainer,p:pfnode,ex_succ:[pfnode],nc:[pfcontainer]): pfjust
+getlowerjustifications(justification :pfjustification): pfjustification list
+gethigherjustifications(justification:pfjustification): pfjustification list
+exportxml(node:pfnode): xml*entity
+repairinvariant(node:pfnode)
pfobject
-timestamp
-id
-hidden
+exportxml: xml*entity
+exportcontent
+hide()
+unhide()

pftree
-nodecounter -nodestore: Node Hashtable -opennodes: Node list -rootnode: Node
-createemptytree(broker:pfbroker,forest:pfforest): pftree +create(broker:pfbroker,forest:pfforest,content:pfcontainer): pftree +clear() +getallnodes(): pfnode list +hedgecompare(h1:hierarchicaledge,h2:hierarchicaledge) -newnodenumber(tree:pftree): integer +exportxml(tree:pftree): xml*entity +delete(tree:pftree)

B DTD for XML-export

```
<!ELEMENT forest (time,treelist,fedgelist)>
<!ELEMENT treelist (tree*)>
<!ELEMENT tree (time, assume?, (node|hedge|justification)*)>
<!ELEMENT assume (node|hedge|justification)*)>
<!ELEMENT content (#PCDATA)>
<!ELEMENT node (symid,time,content)>
<!ELEMENT justification (symid,time,content,source,targetlist)>
<!ELEMENT hedge (symid,time,content,source,target)>
<!ELEMENT symid (#PCDATA)>
<!ELEMENT time (#PCDATA)>
<!ELEMENT source (symid)>
<!ELEMENT target (symid)>
<!ELEMENT targetlist (symid*)>
<!ELEMENT fedgelist (fedge)>
<!ELEMENT fedge (time, content, source, targetlist)>

<!ATTLIST justification
  selected (0|1) "0">
```

Figure 22: pds.dtd

C Test-environments

All tests are written using the CLOS-unit testing framework. For a complete description of this framework see [20].

They run automatically and hence can be repeated at any time.

C.1 Addpdstree-method-test

C.1.1 Scenario 1

1. Add a tree in forest with id 0 and content 123 of type pfcontainernode

Tested criteria: new forest created, new tree in forest, new node in tree with correct content, root-node=new node, timestamp of forest, tree, node, forest-counter increased, timer increased by 3, tree-counter of new forest ok, node counter of new tree correct, forestid correct, invariant hold (in the sequel, this test is called addpdstreeinnewforest)

C.1.2 Scenario 2

1. Add a tree in forest with id 0 and content 2 to get forest1
2. Add a tree in forest with id (id forest1) and content 3
3. Add a tree in forest with id 0 and content 5 to get forest2
4. Add a tree in forest with id 0 and content 7 to get forest3
5. Add a tree in forest with id (forest2) with content 11
6. Add a tree in forest with id (forest3) with content 13
7. Add a tree in forest with id (forest3) with content 17
8. Add a tree in forest with id (forest3) with content 19
9. Add a tree in forest with id (forest2) with content 23

Tested criteria: forest with that id exists, new tree in that forest, new node in the new tree, root node set to new node, timestamp forest didn't change, timestamp tree correct, timestamp node correct, forest counter not changed, timer incremented by 2, tree counter incremented by 1, forestid not changed, treeid correct, nodeid correct, content of root node correct (in the sequel this test is called addpdstreeinexisting forest)

C.2 Justify-method-test

Scenario

1. Create new tree with content 123
2. Create new justification with content 1234 that connects the root node with 4 new nodes with content 1, 2, 3, 4 respectively

3. Create new justification with content 56 that connects the root node with the first to nodes created above and 2 new nodes with content 5,6

Tested criteria: insertpdstreeinnewforest, after each justify: type of result, timestamp content of new created nodes, source of justification, jid entry in justificationstore, justification-id, out-edges of parent node, justification counter, timestamp of justification, target list of justification

C.3 Expand-method-test

Scenario

1. Create new tree with content 123
2. Create new justification with content 1234, connecting the root node of the new tree with 4 new generated nodes with content 1, 2, 3, 4 respectively
3. Expand the new justification with an justification with content 56, connecting the root node with the first and second of the already created nodes as well as 2 new created nodes with content 5 and 6

Tested criteria: same as for justify, and in addition: hedge counter has been increased, hedge-id is correct, hedge store has new entry, hedge is in out-edges of justification, hedge is in in-edges of target justification, hedge is right direction and source and target are correct

C.4 Abstract-method-test

Scenario
same as above, but abstract instead of expand

Tested criteria: same as for expand-method-test

C.5 Hlinks-test

Scenario

1. Create new tree with root node content j123
2. Create new justification j1 with content j1234,k connecting root node with 4 new created nodes with content 1, 2, 3, 4 (n1, n2, n3, n4)
3. Abstract j1 with a justification j5 with content j5, that connects the root node with a new created node n5 with content 5

4. Abstract j1 with a justification j6 with content j6, that connects the root node with a new created node n6 with content 6

5. Abstract j5 with a justification j7 with content j7, that connects the root node with no other nodes

6. Abstract j5 with a justification j8 with content j8, that connects the root node with a new node with content 8

7. Expand j8 with a justification j9 with content j9, that connects the root node with a new node with content 9

8. Call gethigherjustifications j1

9. Call getlowerjutsifications j7

Tested criteria: correct elements in the sets

C.6 Invariant-test

C.6.1 Scenario 1:

Create a node with 1 outgoing justification, expand it 4 times. Select different justifications

Tested criteria: failure in adequacy recognized by invariant-test function

C.6.2 Scenario 2:

Create a node with 1 outgoing justification, abstract it 4 times. Select different justifications

Tested criteria: failure in adequacy recognized by invariant-test function

C.6.3 Scenario 3:

Create a node with 5 outgoing justifications, separated into 2 groups. One of the groups has no representative in the set of alternatives

Tested criteria: completeness failure recognized by invariant-test function

C.7 Justificationselection-test

Scenario: Create a tree with a root node and 50 outgoing justifications, each of which has only one successor. During the insertion it is randomly changed between insert-justification, expand and abstract. When the tree has been created.

Tested criterias: does the invariant hold after each reselection

C.8 Addforestedge_test

Scenario

1. Add a new tree with root node content 1, call id tree1
2. Add a new tree with root node content 2, call it tree2
3. Add a new tree with root node content 3, call it tree3
4. Add a forested GE from root node of tree1 to root node of tree2 and root node of tree3

Tested criteria: return type of addforestedge, time changed properly, forest edge added in forest edge store, length of forest edge store incremented by 1, time of forest edge, id of forest edge, length target list, content of target list, content of new created trees, source of forest-edge

C.9 Delete test

Scenario:

1. Add a new tree with root node content R1
2. Add a new justification that connects the root node with new created nodes with content 1, 2, 3, 4 respectively
3. Expand the justification with new successors node1id and a new created node with content 7
4. Delete justification2

Tested criteria: XML-result, correct entities deleted

References

[1] Peter B. Andrews, Mat Bishop, and Chad E. Brown. System description: TPS: A theorem proving system for type theory. In *Proc. of the 17th Int. Conf. on Automated Deduction (CADE-17)*, LNCS, pages 164–169. Springer, 2000.

[2] Serge Autexier. *Hierachical Context Reasoning*. PhD thesis, Universität des Saarlandes, 2003.

[3] Serge Autexier, Christoph Benzmüller, Dominik Dietrich, Andreas Meier, and Claus-Peter Wirth. A generic modular data structure for proof attempts alternating on ideas and granularity. In Michael Kohlhase, editor, *Proceedings of MKM'05*, volume 3863 of *LNAI*, IUB Bremen, Germany, june 2005. Springer.

[4] Serge Autexier, Christoph Benzmüller, Malte Hübner, Andreas Meier, Martin Pollet, and Claus-Peter Wirth. Proposing a task interface for proof assistants. 2003.

[5] Serge Autexier, Dieter Hutter, Bruno Langenstein, Heiko Mantel, Georg Rock, Axel Schairer, Werner Stephan, Roland Vogt, and Andreas Wolpers. VSE: Formal methods meet industrial needs. *International Journal on Software Tools for Technology Transfer, Special issue on Mechanized Theorem Proving for Technology, Springer*, september 1998.

[6] Jürgen Avenhaus, Ullrich. Kühler, Tobias Schmidt-Samoa, and Claus-Peter Wirth. How to prove inductive theorems? QUODLIBET! In *Proc. of the 19th Int. Conf. on Automated Deduction (CADE-19)*, number 2741 in LNAI, pages 328–333. Springer, 2003.

[7] Christoph Benzmüller, Andreas Meier, Erica Melis, Martin Pollet, and Volker Sorge. Proof planning: A fresh start? In *Proceedings of the IJCAR 2001 Workshop: Future Directions in Automated Reasoning*, pages 25–37, Siena, Italy, 2001.

[8] Yves Bertot and Paul Castéran. *Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science, An EATCS Series. Springer, 2004.

[9] Lassaad Cheikhrouhou and Volker Sorge. $\mathcal{PDS}$— a three-dimensional data structure for proof plans. In *Proceedings of the International Conference on Artificial and Computational Intelligence for Decision, Control and Automation in Engineering and Industrial Applications (ACIDCA'2000)*, Monastir, Tunisia, 22–24 March 2000.

[10] Lucas Dixon. Interactive and hierarchical tracing of techniques in IsaPlanner. In *Proc. of UITP'05*, 2005.

[11] Armin Fiedler. Dialog-driven adaptation of explanations of proofs. In *Proc. of the 17th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1295–1300, Seattle, WA, 2001. Morgan Kaufmann.

[12] Gerhard Gentzen. *The Collected Papers of Gerhard Gentzen (1934-1938)*. Edited by Szabo, M. E., North Holland, Amsterdam, 1969.

[13] Mike J. C. Gordon and Tom F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[14] Malter Hübner, Serge Autexier, Christoph Benzmüller, and Andreas Meier. Interactive theorem proving with tasks. *Electronic Notes in Theoretical Computer Science*, 103(C):161–181, November 2004.

[15] Dieter Hutter and Claus Sengler. INKA - The Next Generation. In *Proc. of the 13th International Conference on Automated Deduction (CADE–13)*, LNAI. Springer, 1996.

[16] Christoph Kreitz, Lori. Lorigo, R. Eaton, Robert L. Constable, and Steward .F Allen. The nuprl open logical environment, 2000.

[17] Andreas Meier. *Proof Planning with Multiple Strategies*. PhD thesis, Universit"at des Saarlandes, 2004.

[18] Robin Milner. Logic for computable functions: description of a machine implementation. Memo AIM-169, Stanford University, 1972.

[19] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer Verlag, 1994.

[20] Sandro Pedrazzini. A CLOS implementation of the JUnit Testing Framework Architecture: A case study.

[21] Jörg Siekmann, Christoph Benzmüller, Vladimir Brezhnev, Lassaad Cheikhrouhou, Armin Fiedler, Andreas Franke, Helmut Horacek, Michael Kohlhase, Andreas Meier, Erica Melis, Markus Moschner, Immanuel Normann, Martin Pollet, Volker Sorge, Carsten Ullrich, Claus-Peter Wirth, and Jürgen Zimmer. Proof development with OMEGA. In Andrei Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction (CADE-18)*, number 2392 in LNAI, pages 144–149, Copenhagen, Denmark, 2002. Springer.

[22] Jörg Siekmann, Christoph Benzmüller, Armin Fiedler, Andreas Meier, Immanuel Normann, and Martin Pollet. *Proof Development in OMEGA: The Irrationality of Square Root of 2*, pages 271–314. Kluwer Academic Publishers, 2003.

[23] Jörg Siekmann, Christoph Benzmüller, Armin Fiedler, Andreas Meier, and Martin Pollet. Proof development with OMEGA: Sqrt(2) is irrational. In *Logic for Programming, Artificial Intelligence, and Reasoning, 9th Int. Conf., LPAR 2002*, number 2514 in LNAI, pages 367–387. Springer, 2002.

[24] Jörg Siekmann, Frank Pfenning, and Xiarong Huang, editors. *Proceedings of the First International Workshop on Proof Transformation and Presentation*, Schloss Dagstuhl, 1997.

[25] Guy L. Steele jr. *Common Lisp*. Digital Press, 1990.

[26] Claus-Peter Wirth. Descente infinie + Deduction. *Logic J. of the IGPL*, 12(1):1–96, 2004.