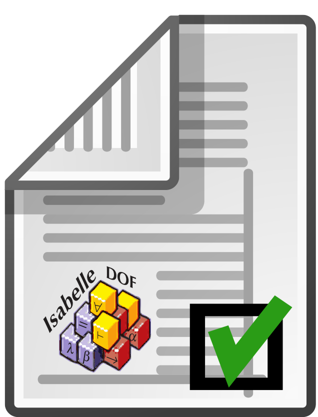
# Isabelle/DOF

## User and Implementation Manual

Achim D. Brucker Burkhart Wolff

August 18, 2019



Department of Computer Science
University of Exeter
Exeter, EX4 4QF
UK

Laboratoire en Recherche en Informatique (LRI)
Université Paris-Saclay
91405 Orsay Cedex
France

Copyright © 2019 University of Exeter, UK
2018–2019 Université Paris-Saclay, France
2018–2019 The University of Sheffield, UK

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

**SPDX-License-Identifier:** BSD-2-Clause

This manual describes Isabelle/DOF version 1.0.0/Isabelle2019. The latest official release is 1.0.0/Isabelle2019 (doi:10.5281/zenodo.3370483). The latest development version as well as official releases are available at https://git.logicalhacking.com/Isabelle_DOF/Isabelle_DOF.

**Contributors.** We would like to thank the following contributors to Isabelle/DOF (in alphabetical order): Idir Ait-Sadoune, Paolo Crisafulli, and Chantal Keller.

**Acknowledgments.** This work has been partially supported by IRT SystemX, Paris-Saclay, France, and therefore granted with public funds of the Program "Investissements d'Avenir."

# Contents


1 Introduction 7

2 Background 11
- 2.1 The Isabelle System Architecture . . . 11
- 2.2 The Document Model Required by DOF . . . 11
- 2.3 Implementability of the Required Document Model. . . . 13

3 Isabelle/DOF: A Guided Tour 15
- 3.1 Getting Started . . . 15
  - 3.1.1 Installation . . . 15
  - 3.1.2 Creating an Isabelle/DOF Project . . . 18
- 3.2 Writing Academic Publications (scholarly_paper) . . . 19
  - 3.2.1 The Scholarly Paper Example . . . 19
  - 3.2.2 Modeling Academic Publications . . . 20
  - 3.2.3 Editing Support for Academic Papers . . . 22
- 3.3 Writing Certification Documents (CENELEC_50128) . . . 23
  - 3.3.1 The CENELEC 50128 Example . . . 23
  - 3.3.2 Modeling CENELEC 50128 . . . 24
  - 3.3.3 Editing Support for CENELEC 50128 . . . 25
- 3.4 Writing Exams (math_exam) . . . 26
  - 3.4.1 The Math Exam Example . . . 26
  - 3.4.2 Modeling Exams . . . 27
- 3.5 Style Guide . . . 29

4 Developing Ontologies 31
- 4.1 Overview and Technical Infrastructure . . . 31
  - 4.1.1 Ontologies . . . 31
  - 4.1.2 Document Templates . . . 33
- 4.2 The Ontology Definition Language (ODL) . . . 33
  - 4.2.1 Some Isabelle/HOL Specification Constructs Revisited . . . 35
  - 4.2.2 Defining Document Classes . . . 37
  - 4.2.3 Common Ontology Library (COL) . . . 40
  - 4.2.4 Annotatable Top-level Text-Elements . . . 43
  - 4.2.5 Status and Inspection Commands . . . 46
  - 4.2.6 Advanced ODL Concepts . . . 46
- 4.3 Defining Document Templates . . . 48
  - 4.3.1 The Core Template . . . 48

4.3.2 Tips, Tricks, and Known Limitations . . . . . 49

**5 Extending Isabelle/DOF** **53**
5.1 Isabelle/DOF: A User-Defined Plugin in Isabelle/Isar . . . . . 53
5.2 Programming Antiquotations . . . . . 55
5.3 Implementing Second-level Type-Checking . . . . . 56
5.4 Programming Class Invariants . . . . . 56
5.5 Implementing Monitors . . . . . 57
5.6 The LaTeX-Core of Isabelle/DOF . . . . . 57

**Abstract**

Isabelle/DOF provides an implementation of DOF on top of Isabelle/HOL. DOF itself is a novel framework for *defining* ontologies and *enforcing* them during document development and document evolution. Isabelle/DOF targets use-cases such as mathematical texts referring to a theory development or technical reports requiring a particular structure. A major application of DOF is the integrated development of formal certification documents (e.g., for Common Criteria or CENELEC 50128) that require consistency across both formal and informal arguments.

Isabelle/DOF is integrated into Isabelle's IDE, which allows for smooth ontology development as well as immediate ontological feedback during the editing of a document. Its checking facilities leverage the collaborative development of documents required to be consistent with an underlying ontological structure.

In this user-manual, we give an in-depth presentation of the design concepts of DOF's Ontology Definition Language (ODL) and describe comprehensively its major commands. Many examples show typical best-practice applications of the system. Isabelle/DOF is the first ontology language supporting machine-checked links between the formal and informal parts in an LCF-style interactive theorem proving environment.

**Keywords:** Ontology, Ontological Modeling, Document Management, Formal Document Development, Document Authoring, Isabelle/DOF

# 1 Introduction

The linking of the *formal* to the *informal* is perhaps the most pervasive challenge in the digitization of knowledge and its propagation. This challenge incites numerous research efforts summarized under the labels "semantic web," "data mining," or any form of advanced "semantic" text processing. A key role in structuring this linking play *document ontologies* (also called *vocabulary* in the semantic web community [19]), i.e., a machine-readable form of the structure of documents as well as the document discourse.

Such ontologies can be used for the scientific discourse within scholarly articles, mathematical libraries, and in the engineering discourse of standardized software certification documents [3, 7]: certification documents have to follow a structure. In practice, large groups of developers have to produce a substantial set of documents where the consistency is notoriously difficult to maintain. In particular, certifications are centered around the *traceability* of requirements throughout the entire set of documents. While technical solutions for the traceability problem exists (most notably: DOORS [10]), they are weak in the treatment of formal entities (such as formulas and their logical contexts).

Further applications are the domain-specific discourse in juridical texts or medical reports. In general, an ontology is a formal explicit description of *concepts* in a domain of discourse (called *classes*), properties of each concept describing *attributes* of the concept, as well as *links* between them. A particular link between concepts is the *is-a* relation declaring the instances of a subclass to be instances of the super-class.

To adress this challenge, we present the Document Ontology Framework (DOF) and an implementation of DOF called Isabelle/DOF. DOF is designed for building scalable and user-friendly tools on top of interactive theorem provers. Isabelle/DOF is a novel framework, implemented as extension of Isabelle/HOL, to *model* typed ontologies and to *enforce* them during document evolution. Based on Isabelle's infrastructures, ontologies may refer to types, terms, proven theorems, code, or established assertions. Based on a novel adaption of the Isabelle IDE, a document is checked to be *conform* to a particular ontology—Isabelle/DOF is designed to give fast user-feedback *during the capture of content*. This is particularly valuable in case of document evolution, where the *coherence* between the formal and the informal parts of the content can be mechanically checked.

To avoid any misunderstanding: Isabelle/DOF is *not a theory in HOL* on ontologies and operations to track and trace links in texts, it is an *environment to write structured text* which *may contain* Isabelle/HOL definitions and proofs like mathematical articles, tech-reports and scientific papers—as the present one, which is written in Isabelle/DOF itself. Isabelle/DOF is a plugin into the Isabelle/Isar framework in the style of [24].

## How to Read This Manual

This manual can be read in different ways, depending on what you want to accomplish. We see three different main user groups:

1. *Isabelle/DOF users*, i.e., users that just want to edit a core document, be it for a paper or a technical report, using a given ontology. These users should focus on Chapter 3 and, depending on their knowledge of Isabelle/HOL, also Chapter 2.

2. *Ontology developers*, i.e., users that want to develop new ontologies or modify existing document ontologies. These users should, after having gained acquaintance as a user, focus on Chapter 4.

3. *Isabelle/DOF developers*, i.e., users that want to extend or modify Isabelle/DOF, e.g., by adding new text-elements. These users should read Chapter 5

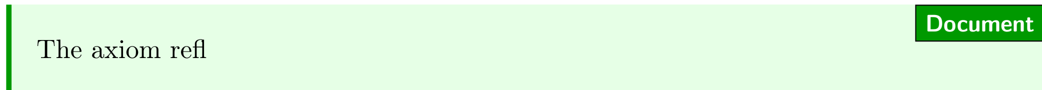
## Typographical Conventions

We acknowledge that understanding Isabelle/DOF and its implementation in all details requires separating multiple technological layers or languages. To help the reader with this, we will type-set the different languages in different styles. In particular, we will use

- a light-blue background for input written in Isabelle's Isar language, e.g.:

Isar
```
lemma refl: x = x
by simp
```

- a green background for examples of generated document fragments (i.e., PDF output):




- a red background for For (S)ML-code:

SML
```
fun id x = x
```

- a yellow background for LaTeX-code:

LaTeX
```
\newcommand{\refl}{$x = x$}
```

- a grey background for shell scripts and interactive shell sessions:

```
Bash
achim@logicalhacking:~$ ls
CHANGELOG.md CITATION examples install LICENSE README.md ROOTS src
```

## How to Cite Isabelle/DOF

If you use or extend Isabelle/DOF in your publications, please use

- for the Isabelle/DOF system [5]:

  A. D. Brucker, I. Ait-Sadoune, P. Crisafulli, and B. Wolff. Using the Isabelle ontology framework: Linking the formal with the informal. In *Conference on Intelligent Computer Mathematics (CICM)*, number 11006 in Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, 2018. 10.1007/978-3-319-96812-4_3.

  A BibTeX-entry is available at: https://www.brucker.ch/bibliography/abstract/brucker.ea-isabelle-ontologies-2018.

- for the implementation of Isabelle/DOF [4]:

  A. D. Brucker and B. Wolff. Isabelle/DOF: Design and implementation. In P.C. Ölveczky and G. Salaün, editors, *Software Engineering and Formal Methods (SEFM)*, number 11724 in Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, 2019. 10.1007/978-3-030-30446-1_15.

  A BibTeX-entry is available at: https://www.brucker.ch/bibliography/abstract/brucker.ea-isabelledof-2019.

## Availability

The implementation of the framework is available at https://git.logicalhacking.com/Isabelle_DOF/Isabelle_DOF. The website also provides links to the latest releases. Isabelle/DOF is licensed under a 2-clause BSD license (SPDX-License-Identifier: BSD-2-Clause).

# 2 Background

## 2.1 The Isabelle System Architecture

While Isabelle [18] is widely perceived as an interactive theorem prover for HOL (Higher-order Logic) [18], we would like to emphasize the view that Isabelle is far more than that: it is the *Eclipse of Formal Methods Tools*. This refers to the "*generic system framework of Isabelle/Isar underlying recent versions of Isabelle. Among other things, Isar provides an infrastructure for Isabelle plug-ins, comprising extensible state components and extensible syntax that can be bound to ML programs. Thus, the Isabelle/Isar architecture may be understood as an extension and refinement of the traditional 'LCF approach', with explicit infrastructure for building derivative systems.*" [24]

The current system framework offers moreover the following features:

- a build management grouping components into to pre-compiled sessions,
- a prover IDE (PIDE) framework [20] with various front-ends
- documentation-generation,
- code generators for various target languages,
- an extensible front-end language Isabelle/Isar, and,
- last but not least, an LCF style, generic theorem prover kernel as the most prominent and deeply integrated system component.

  The Isabelle system architecture shown in Figure 2.1 comes with many layers, with Standard ML (SML) at the bottom layer as implementation language. The architecture actually foresees a *Nano-Kernel* (our terminology) which resides in the SML structure `Context`. This structure provides a kind of container called *context* providing an identity, an ancestor-list as well as typed, user-defined state for components (plugins) such as Isabelle/DOF. On top of the latter, the LCF-Kernel, tactics, automated proof procedures as well as specific support for higher specification constructs were built.

## 2.2 The Document Model Required by DOF

In this section, we explain the assumed document model underlying our Document Ontology Framework (DOF) in general. In particular we discuss the concepts *integrated document*, *sub-document*, *text-element* and *semantic macros* occurring inside text-elements. Furthermore, we assume two different levels of parsers (for *outer* and *inner syntax*) where the inner-syntax is basically a typed $\lambda$-calculus and some Higher-order Logic (HOL).

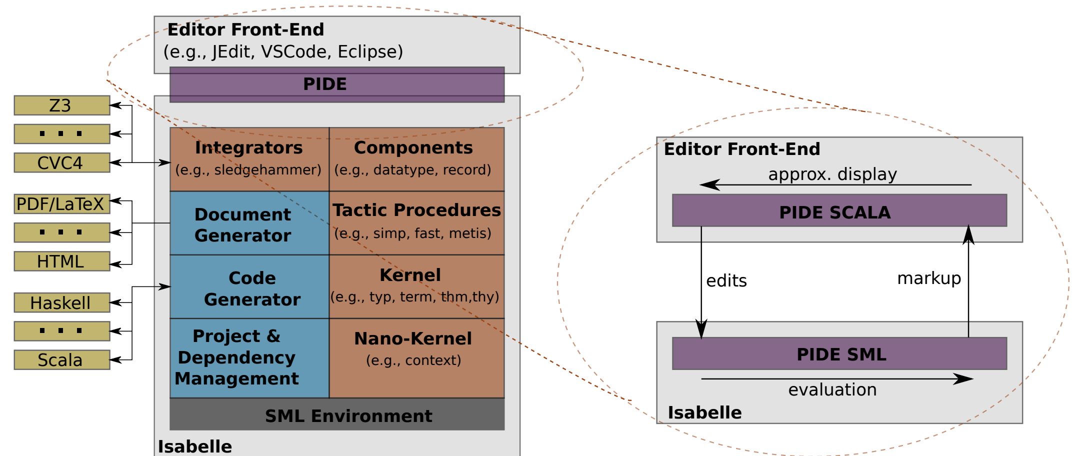



Figure 2.1: The system architecture of Isabelle (left-hand side) and the asynchronous communication between the Isabelle system and the IDE (right-hand side).

We assume a hierarchical document model, i.e., an *integrated* document consist of a hierarchy *sub-documents* (files) that can depend acyclically on each other. Sub-documents can have different document types in order to capture documentations consisting of documentation, models, proofs, code of various forms and other technical artifacts. We call the main sub-document type, for historical reasons, *theory*-files. A theory file consists of a *header*, a *context definition*, and a body consisting of a sequence of *commands* (see Figure 2.2). Even the header consists of a sequence of commands used for introductory text elements not depending on any context. The context-definition contains an `import` and a keyword section, for example:

Isar

```
theory Example              (* Name of the 'theory'              *)
  imports                   (* Declaration of 'theory' dependencies *)
    Main                    (* Imports a library called 'Main'   *)
  keywords                  (* Registration of keywords defined locally *)
    requirement             (* A command for describing requirements    *)
```

where `Example` is the abstract name of the text-file, `Main` refers to an imported theory (recall that the import relation must be acyclic) and **keywords** are used to separate commands from each other.

We distinguish fundamentally two different syntactic levels:

- the *outer-syntax* (i.e., the syntax for commands) is processed by a lexer-library and parser combinators built on top, and
- the *inner-syntax* (i.e., the syntax for $\lambda$-terms in HOL) with its own parametric polymorphism type checking.

On the semantic level, we assume a validation process for an integrated document, where the semantics of a command is a transformation $\theta \rightarrow \theta$ for some system state $\theta$. This document model can be instantiated with outer-syntax commands for common text elements,

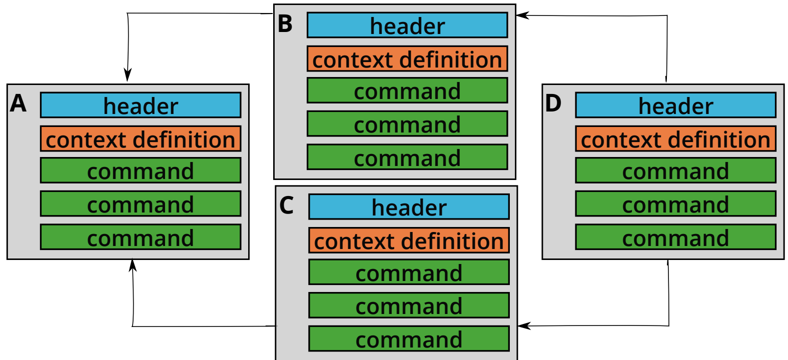



Figure 2.2: A Theory-Graph in the Document Model.

e.g., `section‹ . . . ›` or `text‹ . . . ›`. Thus, users can add informal text to a sub-document using a text command:

```
text‹This is a description.›
```
Isar

This will type-set the corresponding text in, for example, a PDF document. However, this translation is not necessarily one-to-one: text elements can be enriched by formal, i.e., machine-checked content via *semantic macros*, called antiquotations:

```
text‹According to the reflexivity axiom @{thm refl}, we obtain in Γ
for @{term fac 5} the result @{value fac 5}.›
```
Isar

which is represented in the final document (e.g., a PDF) by:

According to the reflexivity axiom $x = x$, we obtain in $\Gamma$ for $\mathrm{fac}\,5$ the result $120$.

Document

Semantic macros are partial functions of type $\theta \longrightarrow$ `text`; since they can use the system state, they can perform all sorts of specific checks or evaluations (type-checks, executions of code-elements, references to text-elements or proven theorems such as `refl`, which is the reference to the axiom of reflexivity).

Semantic macros establish *formal content* inside informal content; they can be type-checked before being displayed and can be used for calculations before being typeset. They represent the device for linking the formal with the informal.

## 2.3 Implementability of the Required Document Model.

Batch-mode checkers for DOF can be implemented in all systems of the LCF-style prover family, i.e., systems with a type-checked `term`, and abstract `thm`-type for theorems (protected by a kernel). This includes, e.g., ProofPower, HOL4, HOL-light, Isabelle, or Coq and

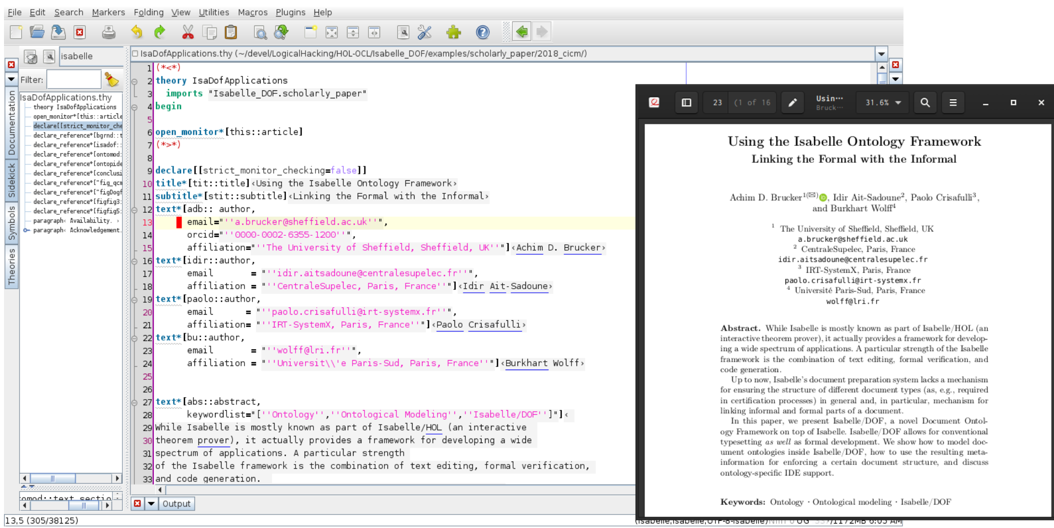



Figure 2.3: The Isabelle/DOF IDE (left) and the corresponding PDF (right), showing the first page of [5].

its derivatives. DOF is, however, designed for fast interaction in an IDE. If a user wants to benefit from this experience, only Isabelle and Coq have the necessary infrastructure of asynchronous proof-processing and support by a PIDE [1, 9, 20, 21] which in many features over-accomplishes the required features of DOF. For example, current Isabelle versions offer cascade-syntaxes (different syntaxes and even parser-technologies which can be nested along the ‹ . . . › barriers, while DOF actually only requires a two-level syntax model.

We call the present implementation of DOF on the Isabelle platform Isabelle/DOF. Figure 2.3 shows a screen-shot of an introductory paper on Isabelle/DOF [5]: the Isabelle/DOF PIDE can be seen on the left, while the generated presentation in PDF is shown on the right.

Isabelle provides, beyond the features required for DOF, a lot of additional benefits. For example, it also allows the asynchronous evaluation and checking of the document content [1, 20, 21] and is dynamically extensible. Its PIDE provides a *continuous build, continuous check* functionality, syntax highlighting, and auto-completion. It also provides infrastructure for displaying meta-information (e.g., binding and type annotation) as pop-ups, while hovering over sub-expressions. A fine-grained dependency analysis allows the processing of individual parts of theory files asynchronously, allowing Isabelle to interactively process large (hundreds of theory files) documents. Isabelle can group sub-documents into sessions, i.e., sub-graphs of the document-structure that can be "pre-compiled" and loaded instantaneously, i.e., without re-processing.

# 3 Isabelle/DOF: A Guided Tour

In this chapter, we will give a introduction into using Isabelle/DOF for users that want to create and maintain documents following an existing document ontology.

## 3.1 Getting Started

As an alternative to installing Isabelle/DOF locally, the latest official release Isabelle/DOF is also available on Docker Hub. Thus, if you have Docker installed and your installation of Docker supports X11 application, you can start Isabelle/DOF as follows:

```bash
achim@logicalhacking:~$ docker run -ti --rm -e DISPLAY=$DISPLAY \
   -v /tmp/.X11-unix:/tmp/.X11-unix \
   logicalhacking/isabelle_dof-1.0.0_isabelle2019 \
   isabelle jedit
```

### 3.1.1 Installation

In this section, we will show how to install Isabelle/DOF and its pre-requisites: Isabelle and LaTeX. We assume a basic familiarity with a Linux/Unix-like command line (i.e., a shell).

#### Pre-requisites

Isabelle/DOF has to major pre-requisites:

- **Isabelle** (Isabelle2019: June 2019). Isabelle/DOF uses a two-part version system (e.g., 1.0.0/2019), where the first part is the version of Isabelle/DOF (using semantic versioning) and the second part is the supported version of Isabelle. Thus, the same version of Isabelle/DOF might be available for different versions of Isabelle.

- **TeXLive 2019** or any other modern LaTeX-distribution where pdfTeX supports `\expanded` (https://www.texdev.net/2018/12/06/a-new-primitive-expanded).

**Installing Isabelle** Please download and install the Isabelle Isabelle2019 distribution for your operating system from the Isabelle website (https://isabelle.in.tum.de/website-Isabelle2019/). After the successful installation of Isabelle, you should be able to call the `isabelle` tool on the command line:

```bash
achim@logicalhacking:~$ isabelle version
Isabelle2019: June 2019
```

Depending on your operating system and depending if you put Isabelle's `bin` directory in your `PATH`, you will need to invoke `isabelle` using its full qualified path, e.g.:

```bash
achim@logicalhacking:~$ /usr/local/IsabelleIsabelle2019/bin/isabelle version
Isabelle2019: June 2019
```

**Installing TEXLive** Modern Linux distribution will allow you to install TEXLive using their respective package managers. On a modern Debian system or a Debian derivative (e.g., Ubuntu), the following command should install all required LATEX packages:

```bash
achim@logicalhacking:~$ sudo aptitude install texlive-latex-extra  \
    texlive-fonts-extra
```

Please check that this, indeed, installs a version of pdfTEX that supports the `\expanded`-primitive. To check your pdfTEX-binary, execute

```bash
achim@logicalhacking:~$ pdftex \\expanded{Success}\\end
This is pdfTeX, Version 3.14159265-2.6-1.40.20 (TeX Live 2019/Debian).
Output written on texput.pdf (1 page, 8650 bytes).
Transcript written on texput.log.
```

If this generates successfully a file `texput.pdf`, your pdfTEX-binary supports the `\expanded`-primitive. If your Linux distribution does not (yet) ship TEXLive 2019 or your are running Windows or OS X, please follow the installation instructions from https://www.tug.org/texlive/acquire-netinstall.html.

### Installing Isabelle/DOF

In the following, we assume that you already downloaded the Isabelle/DOF distribution (Isabelle_DOF-1.0.0_Isabelle2019.tar.xz) from the Isabelle/DOF web site. The main steps for installing are extracting the Isabelle/DOF distribution and calling its `install` script. We start by extracting the Isabelle/DOF archive:

```bash
achim@logicalhacking:~$ tar xf Isabelle_DOF-1.0.0_Isabelle2019.tar.xz
```

This will create a directory `Isabelle_DOF-1.0.0_Isabelle2019` containing Isabelle/DOF distribution. Next, we need to invoke the `install` script. If necessary, the installations automatically downloads additional dependencies from the AFP (https://www.isa-afp.org), namely the AFP entries "Functional Automata" [16] and "Regular Sets and Expressions" [14]. This might take a few minutes to complete. Moreover, the installation script applies a patch to the Isabelle system, which requires *write permissions for the Isabelle system directory* and registers Isabelle/DOF as Isabelle component.

If the `isabelle` tool is not in your PATH, you need to call the `install` script with the `--isabelle` option, passing the full-qualified path of the `isabelle` tool (`install --help`

gives you an overview of all available configuration options):

```
Bash
achim@logicalhacking:~$ cd Isabelle_DOF-1.0.0_Isabelle2019
achim@logicalhacking:~/Isabelle_DOF-1.0.0_Isabelle2019$ ./install --isabelle  \
    /usr/local/IsabelleIsabelle2019/bin/isabelle

Isabelle/DOF Installer
======================
* Checking Isabelle version:
  Success: found supported Isabelle version (Isabelle2019: June 2019)
* Checking (La)TeX installation:
  Success: pdftex supports \expanded{} primitive.
* Check availability of Isabelle/DOF patch:
  Warning: Isabelle/DOF patch is not available or outdated.
           Trying to patch system ....
       Applied patch successfully, Isabelle/HOL will be rebuilt during
       the next start of Isabelle.
* Checking availability of AFP entries:
  Warning: could not find AFP entry Regular-Sets.
  Warning: could not find AFP entry Functional-Automata.
           Trying to install AFP (this might take a few *minutes*) ....
           Registering Regular-Sets in
                 /home/achim/.isabelle/IsabelleIsabelle2019/ROOTS
           Registering Functional-Automata in
                 /home/achim/.isabelle/IsabelleIsabelle2019/ROOTS
           AFP installation successful.
* Searching for existing installation:
  No old installation found.
* Installing Isabelle/DOF
  - Installing Tools in
       /home/achim/.isabelle/IsabelleIsabelle2019/DOF/Tools
  - Installing document templates in
       /home/achim/.isabelle/IsabelleIsabelle2019/DOF/document-template
  - Installing LaTeX styles in
      /home/achim/.isabelle/IsabelleIsabelle2019/DOF/latex
  - Registering Isabelle/DOF
    * Registering tools in
      /home/achim/.isabelle/IsabelleIsabelle2019/etc/settings
* Installation successful. Enjoy Isabelle/DOF, you can build the session
  Isabelle/DOF and all example documents by executing:
  /usr/local/IsabelleIsabelle2019/bin/isabelle build -D .
```

After the successful installation, you can now explore the examples (in the sub-directory examples or create your own project. On the first start, the session Isabelle_DOF will be built automatically. If you want to pre-build this session and all example documents, execute:

```
Bash
achim@logicalhacking:~/Isabelle_DOF-1.0.0_Isabelle2019$ isabelle build -D .
```

### 3.1.2 Creating an Isabelle/DOF Project

Isabelle/DOF provides its own variant of Isabelle's `mkroot` tool, called `mkroot_DOF`:

Bash
```
achim@logicalhacking:~$ isabelle mkroot_DOF -h

Usage: isabelle mkroot_DOF [OPTIONS] [DIR]

Options are:
-h          print this help text and exit
-n NAME     alternative session name (default: DIR base name)
-o ONTOLOGY (default: scholarly_paper)
Available ontologies:
* CENELEC_50128
* math_exam
* scholarly_paper
* technical_report
-t TEMPLATE  (default: scrartcl)
Available document templates:
* lncs
* scrartcl
* scrreprt-modern
* scrreprt

Prepare session root DIR (default: current directory).
```

Creating a new document setup requires two decisions:

- which ontologies (e.g., scholarly_paper) are required and
- which document template (layout) should be used (e.g., scrartcl). Some templates (e.g., lncs) require that the users manually obtains and adds the necessary LATEXclass file (e.g., `llncs.cls`. This is mostly due to licensing restrictions.

If you are happy with the defaults, i.e., using the ontology for writing academic papers (scholarly_paper) using a report layout based on the article class (`scrartcl`) of the KOMA-Script bundle [12], you can create your first project `myproject` as follows:

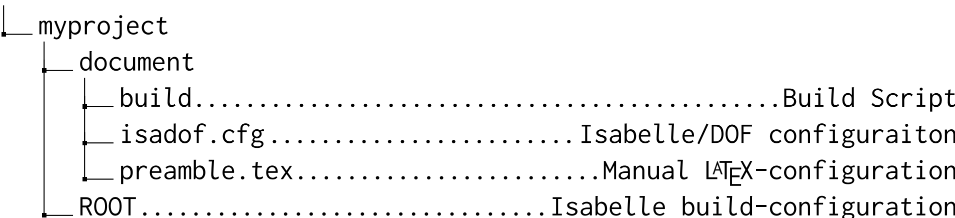
Bash
```
achim@logicalhacking:~$ isabelle mkroot_DOF myproject

Preparing session "myproject" in "myproject"
creating "myproject/ROOT"
creating "myproject/document/root.tex"

Now use the following command line to build the session:
isabelle build -D myproject
```

This creates a directory `myproject` containing the Isabelle/DOF-setup for your new document. To check the document formally, including the generation of the document in PDF,

you only need to execute

```bash
achim@logicalhacking:~$  isabelle build -d . myproject
```

This will create the directory myproject:




The Isabelle/DOF configuration (isadof.cfg) specifies the required ontologies and the document template using a YAML syntax.[1] The main two configuration files for users are:

- The file ROOT, which defines the Isabelle session. New theory files as well as new files required by the document generation (e.g., images, bibliography database using BibTEX, local LATEX-styles) need to be registered in this file. For details of Isabelle's build system, please consult the Isabelle System Manual [23].
- The file praemble.tex, which allows users to add additional LATEX-packages or to add/modify LATEX-commands.

## 3.2 Writing Academic Publications (scholarly_paper)

### 3.2.1 The Scholarly Paper Example

The ontology "scholarly_paper" is a small ontology modeling academic/scientific papers. In this Isabelle/DOF application scenario, we deliberately refrain from integrating references to (Isabelle) formal content in order demonstrate that Isabelle/DOF is not a framework from Isabelle users to Isabelle users only. Of course, such references can be added easily and represent a particular strength of Isabelle/DOF.

The Isabelle/DOF distribution contains an example (actually, our CICM 2018 paper [5]) using the ontology "scholarly_paper" in the directory examples/scholarly_paper/2018-cicm-isabelle_dof-applications/. You can inspect/edit the example in Isabelle's IDE, by either

- starting Isabelle/jedit using your graphical user interface (e.g., by clicking on the Isabelle-Icon provided by the Isabelle installation) and loading the file examples/scholarly_paper/2018-cicm-isabelle_dof-applications/IsaDofApplications.thy.

[1] Isabelle power users will recognize that Isabelle/DOF's document setup does not make use of a file root.tex: this file is replaced by built-in document templates.

- starting Isabelle/jedit from the command line by calling:

```bash
achim@logicalhacking:~/Isabelle_DOF-1.0.0_Isabelle2019$
isabelle jedit  \
    examples/scholarly_paper/2018-cicm-isabelle_dof-applications/\
IsaDofApplications.thy
```

You can build the PDF-document by calling:

```bash
achim@logicalhacking:~$ isabelle build \
2018-cicm-isabelle_dof-applications
```

### 3.2.2 Modeling Academic Publications

We start by modeling the usual text-elements of an academic paper: the title and author information, abstract, and text section:

```isar
doc_class title =
   short_title :: string option  <=  None

doc_class subtitle =
   abbrev :: string option       <=  None

doc_class author =
   affiliation :: string

doc_class abstract =
   keyword_list :: string list  <= None

doc_class text_section =
   main_author :: author option  <=  None
   todo_list   :: string list    <=  []
```

The attributes `short_title`, `abbrev` etc are introduced with their types as well as their default values. Our model prescribes an optional `main_author` and a todo-list attached to an arbitrary text section; since instances of this class are mutable (meta)-objects of text-elements, they can be modified arbitrarily through subsequent text and of course globally during text evolution. Since author is a HOL-type internally generated by Isabelle/DOF framework and can therefore appear in the `main_author` attribute of the `text_section` class; semantic links between concepts can be modeled this way.

Figure 3.1 shows the corresponding view in the Isabelle/jedit of the start of an academic paper. The text uses Isabelle/DOF's own text-commands containing the meta-information

```
IsaDofApplications.thy (~/codebox/publications/working/2018-cicm-isa_dof-applications/IsaDofApplications/)
1  (*<*)
2  theory IsaDofApplications
3    imports "Isabelle_DOF/ontologies/scholarly_paper"
4  begin
5  (*>*)
6
7  title*[tit::title]‹Using The Isabelle Ontology Framework›
8  subtitle*[stit::subtitle]‹Linking the Formal with the Informal›
9
10 text*[adb::author,   email="''a.brucker@sheffield.ac.uk''",  orcid="''0000-0002-6355-1200''",
11                      affiliation="''University of Sheffield, Sheffield, UK''"]  ‹Achim D. Brucker›
12 text*[auth2::author, email="''idir.aitsadoune@centralesupelec.fr''",
13                      affiliation = "''CentraleSupelec, Paris, France''"]        ‹Idir Ait-Sadoune›
14 text*[auth3::author, email="''paolo.crisafulli@irt-systemx.fr''",
15                      affiliation = "''IRT-SystemX, Paris, France''"]            ‹Paolo Crisafulli›
16 text*[bu::author,    email="''wolff@lri.fr''",
17                      affiliation="''Universit\\'e Paris-Sud, Paris, France''"]  ‹Burkhart Wolff›
18
19
20 text*[abs::abstract, keywordlist="[''Isabelle/Isar'', ''HOL'', ''Ontologies'']"]{*
```

Figure 3.1: Ouroboros I: This paper from inside ...

provided by the underlying ontology. We proceed by a definition of `introduction`'s, which we define as the extension of `text_section` which is intended to capture common infrastructure:

Isar
```
doc_class introduction = text_section +
   comment :: string
```

As a consequence of the definition as extension, the `introduction` class inherits the attributes `main_author` and `todo_list` together with the corresponding default values.

We proceed more or less conventionally by the subsequent sections:

Isar
```
doc_class technical = text_section +
   definition_list :: string list <=  []

doc_class example   = text_section +
   comment :: string

doc_class conclusion = text_section +
   main_author :: author option  <=  None

doc_class related_work = conclusion +
   main_author :: author option  <=  None
```

Moreover, we model a document class for including figures (actually, this document class is already defined in the core ontology of Isabelle/DOF):

Figure 3.2: Ouroboros II: figures . . .

(a) Exploring a reference of a text-element.

(b) Exploring the class of a text element.

Figure 3.3: Exploring text elements.

```
Isar
datatype placement = h | t | b | ht | hb
doc_class figure   = text_section +
   relative_width  :: int (* percent of textwidth *)
   src             :: string
   placement       :: placement
   spawn_columns   :: bool <= True
```

The document class figure (supported by the Isabelle/DOF command figure*) makes it possible to express the pictures and diagrams such as Figure 3.2.

Finally, we define a monitor class definition that enforces a textual ordering in the document core by a regular expression:

```
Isar
doc_class article =
   style_id :: string                <= ''LNCS''
   version  :: (int × int × int)  <= (0,0,0)
   where (title ~~ [[subtitle]] ~~ {author}+ ~~ abstract ~~
          introduction ~~ {technical || example}+ ~~ conclusion ~~
          bibliography)
```

### 3.2.3 Editing Support for Academic Papers

From these class definitions, Isabelle/DOF also automatically generated editing support for Isabelle/jedit. In Figure 3.3a and Figure 3.3b we show how hovering over links permits to explore its meta-information. Clicking on a document class identifier permits to hyperlink into the corresponding class definition (Figure 3.4a); hovering over an attribute-definition (which is qualified in order to disambiguate; Figure 3.4b).

An ontological reference application in Figure 3.5: the ontology-dependant antiquotation @{example . . . } refers to the corresponding text-elements. Hovering allows for inspection,

(a) Hyperlink to class-definition.

(b) Exploring an attribute.

Figure 3.4: Hyperlinks.

Figure 3.5: Exploring an attribute (hyperlinked to the class).

clicking for jumping to the definition. If the link does not exist or has a non-compatible type, the text is not validated.

## 3.3 Writing Certification Documents (CENELEC_50128)

### 3.3.1 The CENELEC 50128 Example

The ontology "CENELEC_50128" is a small ontology modeling documents for a certification following CENELEC 50128 [3]. The Isabelle/DOF distribution contains a small example using the ontology "CENELEC_50128" in the directory examples/CENELEC_50128/mini_odo/. You can inspect/edit the example in Isabelle's IDE, by either

- starting Isabelle/jedit using your graphical user interface (e.g., by clicking on the Isabelle-Icon provided by the Isabelle installation) and loading the file examples/CENELEC_50128/mini_odo/mini_odo.thy.
- starting Isabelle/jedit from the command line by calling:

```bash
achim@logicalhacking:~/Isabelle_DOF-1.0.0_Isabelle2019$
isabelle jedit examples/CENELEC_50128/mini_odo/mini_odo.thy
```

You can build the PDF-document by calling:

```bash
achim@logicalhacking:~$ isabelle build mini_odo
```

### 3.3.2 Modeling CENELEC 50128

Documents to be provided in formal certifications (such as CENELEC 50128 [3] or Common Criteria [7]) can much profit from the control of ontological consistency: a lot of an evaluators work consists in tracing down the links from requirements over assumptions down to elements of evidence, be it in the models, the code, or the tests. In a certification process, traceability becomes a major concern; and providing mechanisms to ensure complete traceability already at the development of the global document will clearly increase speed and reduce risk and cost of a certification process. Making the link-structure machine-checkable, be it between requirements, assumptions, their implementation and their discharge by evidence (be it tests, proofs, or authoritative arguments), is therefore natural and has the potential to decrease the cost of developments targeting certifications. Continuously checking the links between the formal and the semi-formal parts of such documents is particularly valuable during the (usually collaborative) development effort.

As in many other cases, formal certification documents come with an own terminology and pragmatics of what has to be demonstrated and where, and how the trace-ability of requirements through design-models over code to system environment assumptions has to be assured.

In the sequel, we present a simplified version of an ontological model used in a case-study [2]. We start with an introduction of the concept of requirement:

Isar

```
doc_class requirement = long_name :: string option

doc_class requirement_analysis = no :: nat
   where requirement_item +

doc_class hypothesis = requirement +
      hyp_type :: hyp_type <= physical  (* default *)

datatype ass_kind = informal | semiformal | formal

doc_class assumption = requirement +
     assumption_kind :: ass_kind <= informal
```

Such ontologies can be enriched by larger explanations and examples, which may help the team of engineers substantially when developing the central document for a certification, like an explication what is precisely the difference between an *hypothesis* and an *assumption* in the context of the evaluation standard. Since the PIDE makes for each document class its definition available by a simple mouse-click, this kind on meta-knowledge can be made far more accessible during the document evolution.

For example, the term of category *assumption* is used for domain-specific assumptions. It has formal, semi-formal and informal sub-categories. They have to be tracked and discharged by appropriate validation procedures within a certification process, by it by test or proof. It is different from a hypothesis, which is globally assumed and accepted.

In the sequel, the category *exported constraint* (or *ec* for short) is used for formal assump-

```
1035 text{*
1036 The resolution of time, distance, speed and acceleration data, in International System Unit,
1037 shall be:
1038 ▪ @{term  Time}: 10$^{-2}$s
1039   the resolution needed for calculation.
1040 ▪ @{term  Distance}: 10$^{-3}$m (i.e. 1mm)
1041 ▪ @{const Speed}: 1.3 x 10$^{-3}$m/s (i.e. 0.005 km/h)
1042 ▪ @{const Acceleration}: 0.005m/s$^2$
1043 ▪ @{const Jerk
1044
1045 The precision                            culations and propagated to the external
1046 interface data.
                constant "Odo_ReqAna.Acceleration"
```

Figure 3.6: Standard antiquotations referring to theory elements.

tions, that arise during the analysis, design or implementation and have to be tracked till the final evaluation target, and discharged by appropriate validation procedures within the certification process, by it by test or proof. A particular class of interest is the category *safety related application condition* (or *srac* for short) which is used for *ec*'s that establish safety properties of the evaluation target. Their track-ability throughout the certification is therefore particularly critical. This is naturally modeled as follows:

```
Isar
doc_class ec = assumption  +
     assumption_kind :: ass_kind <= (*default *) formal

doc_class srac = ec  +
     assumption_kind :: ass_kind <= (*default *) formal
```

We now can, e.g., write

```
Isar
text*[ass123::SRAC]‹
  The overall sampling frequence of the odometer subsystem is therefore
  14 khz, which includes sampling, computing and result communication
  times \ldots
›
```

This will be shown in the PDF as follows:

**SRAC 1.** *The overall sampling frequence of the odometer subsystem is therefore 14 khz, which includes sampling, computing and result communication times . . .*

### 3.3.3 Editing Support for CENELEC 50128

The corresponding view in Figure 3.6 shows core part of a document conformimg to the CENELEC 50128 ontology. The first sample shows standard Isabelle antiquotations [22] into formal entities of a theory. This way, the informal parts of a document get "formal content" and become more robust under change.

```
814 text*[enough_samples::srac]{* Note that the theorem above establishes a constraint between
815 @{consts w_circ}, @{consts tpw} , @{consts Speed_Max} and  sample_frequency; since this
816 exported constraint is fundamental for the safe functioning of  odometer and therefore
817 a safety-related exported application constraint. It is formally expressed as follows:
818 *}
819
```

Figure 3.7: Defining a SRAC reference ...

```
822
823 text{* Summing up, the property that the odometer provides sufficient sampling
824 precision --- meaning no wheel encodings were ``lost'' compared to any sampling done with
825 a higher sampling rate --- can be established under the set of general hypothesis captured
826 in @{docref ‹general_hyps›} (formally expressed in @{thm normally_behaved_distance_function_def})
827 and the SRAC @{ec ‹enough_samples›} formally expressed by @{thm srac_1_def}. *}
828
```

Figure 3.8: Using a SRAC as EC document reference.

The subsequent sample in Figure 3.7 shows the definition of an *safety-related application condition*, a side-condition of a theorem which has the consequence that a certain calculation must be executed sufficiently fast on an embedded device. This condition can not be established inside the formal theory but has to be checked by system integration tests. Now we reference in Figure 3.8 this safety-related condition; however, this happens in a context where general *exported constraints* are listed. Isabelle/DOF's checks establish that this is legal in the given ontology.

## 3.4 Writing Exams (math_exam)

### 3.4.1 The Math Exam Example

The ontology "math_exam" is an experimental ontology modeling the process of writing exams at higher education institution in the United Kingdom, where exams undergo both an internal and external review process. The Isabelle/DOF distribution contains a tiny example using the ontology "math_exam" in the directory examples/math_exam/MathExam/. You can inspect/edit the example in Isabelle's IDE, by either

- starting Isabelle/jedit using your graphical user interface (e.g., by clicking on the Isabelle-Icon provided by the Isabelle installation) and loading the file examples/math_exam/MathExam/MathExam.thy.
- starting Isabelle/jedit from the command line by calling:

```bash
achim@logicalhacking:~/Isabelle_DOF-1.0.0_Isabelle2019$
isabelle jedit examples/math_exam/MathExam/MathExam.thy
```

You can build the PDF-document by calling:

Bash
```
achim@logicalhacking:~$ isabelle build MathExam
```

### 3.4.2 Modeling Exams

The math-exam scenario is an application with mixed formal and semi-formal content. It addresses applications where the author of the exam is not present during the exam and the preparation requires a very rigorous process.

We assume that the content has four different types of addressees, which have a different *view* on the integrated document:

- the *setter*, i.e., the author of the exam,
- the *checker*, i.e., an internal person that checks the exam for feasibility and non-ambiguity,
- the *external*, i.e., an external person that checks the exam for feasibility and non-ambiguity, and
- the *student*, i.e., the addressee of the exam.

The latter quality assurance mechanism is used in many universities, where for organizational reasons the execution of an exam takes place in facilities where the author of the exam is not expected to be physically present. Furthermore, we assume a simple grade system (thus, some calculation is required). We can model this as follows:

Isar
```
doc_class Author = . . .
datatype Subject =  algebra | geometry | statistical
datatype Grade =  A1 | A2 | A3
doc_class Header =  examTitle   :: string
                    examSubject :: Subject
                    date        :: string
                    timeAllowed :: int --  minutes
datatype ContentClass =  setter
                       | checker
                       | external_examiner
                       | student
doc_class Exam_item =  concerns :: ContentClass set
doc_class Exam_item =  concerns :: ContentClass set

type_synonym SubQuestion = string
```

The heart of this ontology is an alternation of questions and answers, where the answers can consist of simple yes-no answers or lists of formulas. Since we do not assume familiarity of the students with Isabelle (`term` would assume that this is a parse-able and type-checkable entity), we basically model a derivation as a sequence of strings:

```
Isar
doc_class Answer_Formal_Step =  Exam_item +
  justification :: string
  term          :: string

doc_class Answer_YesNo =  Exam_item +
  step_label :: string
  yes_no     :: bool  -- \isa{for\ checkboxes}

datatype Question_Type =
  formal | informal | mixed

doc_class Task = Exam_item +
  level    :: Level
  type     :: Question_Type
  subitems :: (SubQuestion *
                 (Answer_Formal_Step list + Answer_YesNo) list) list
  concerns :: ContentClass set <= UNIV
  mark     :: int
doc_class Exercise = Exam_item +
  type     :: Question_Type
  content  :: (Task) list
  concerns :: ContentClass set <= UNIV
  mark     :: int
```

In many institutions, having a rigorous process of validation for exam subjects makes sense: is the initial question correct? Is a proof in the sense of the question possible? We model the possibility that the *examiner* validates a question by a sample proof validated by Isabelle:

```
Isar
doc_class Validation =
   tests  :: term list  <=[]
   proofs :: thm list   <=[]

doc_class Solution = Exam_item +
  content  :: Exercise list
  valids   :: Validation list
  concerns :: ContentClass set <= {setter,checker,external_examiner}

doc_class MathExam=
  content :: (Header + Author + Exercise) list
  global_grade :: Grade
  where ⦃Author⦄+      Header      ⦃Exercise    Solution⦄+
```

In our scenario this sample proofs are completely *intern*, i.e., not exposed to the students but just additional material for the internal review process of the exam.

## 3.5 Style Guide

The document generation process of Isabelle/DOF is based on Isabelle's document generation framework, using LaTeX as the underlying back-end. As Isabelle's document generation framework, it is possible to embed (nearly) arbitrary LaTeX-commands in text-commands, e.g.:

```
Isar
text‹ This is \emph{emphasized} and this is a
      citation \cite{brucker.ea:isabelle-ontologies:2018}›
```

In general, we advise against this practice and, whenever positive, use the Isabelle/DOF (respetively Isabelle) provided alternatives:

```
Isar
text‹ This is *‹emphasized› and this is a
       citation @{cite brucker.ea:isabelle-ontologies:2018}.›
```

Clearly, this is not always possible and, in fact, often Isabelle/DOF documents will contain LaTeX-commands, this should be restricted to layout improvements that otherwise are (currently) not possible. As far as possible, the use of LaTeX-commands should be restricted to the definition of ontologies and document templates (see Chapter 4).

Restricting the use of LaTeXhas two advantages: first, LaTeXcommands can circumvent the consistency checks of Isabelle/DOF and, hence, only if no LaTeXcommands are used, Isabelle/DOF can ensure that a document that does not generate any error messages in Isabelle/jedit also generated a PDF document. Second, future version of Isabelle/DOF might support different targets for the document generation (e.g., HTML) which, naturally, are only available to documents not using native LaTeX-commands.

Similarly, (unchecked) forward references should, if possible, be avoided, as they also might create dangeling references during the document generation that break the document generation.

Finally, we recommend to use the `check_doc_global` command at the end of your document to check the global reference structure.

# 4 Developing Ontologies

In this chapter, we explain the concepts for modeling new ontologies, developing a document representation for them, as well as developing new document templates.

## 4.1 Overview and Technical Infrastructure

Isabelle/DOF is embedded in the underlying generic document model of Isabelle as described in Section 2.2. Recall that the document language can be extended dynamically, i.e., new *user−defined* can be introduced at run-time. This is similar to the definition of new functions in an interpreter. Isabelle/DOF as a system plugin is is a number of new command definitions in Isabelle's document model.

Isabelle/DOF consists consists basically of four components:

- an own *family of text-elements* such as **title***, **chapter*** **text***, etc., which can be annotated with meta-information defined in the underlying ontology definition and allow to build a *core* document,
- the *ontology definition language* (called ODL) which allow for the definitions of document-classes and necessary auxiliary datatypes,
- an infrastructure for ontology-specific *layout definitions*, exploiting this meta-information, and
- an infrastructure for generic *layout definitions* for documents following, e.g., the format guidelines of publishers or standardization bodies.

The list of fully supported (i.e., supporting both interactive ontological modeling and document generation) ontologies and the list of supported document templates can be obtained by calling `isabelle mkroot_DOF -h` (see Section 3.1.2). Note that the postfix `-UNSUPPORTED` denotes experimental ontologies or templates for which further manual setup steps might be required or that are not fully tested. Also note that the LaTeX-class files required by the templates need to be already installed on your system. This is mostly a problem for publisher specific templates (e.g., Springer's llncs.cls), which cannot be re-distributed due to copyright restrictions.

### 4.1.1 Ontologies

The document core *may*, but *must* not use Isabelle definitions or proofs for checking the formal content—this manual is actually an example of a document not containing any proof. Consequently, the document editing and checking facility provided by Isabelle/DOF addresses

the needs of common users for an advanced text-editing environment, neither modeling nor proof knowledge is inherently required.

We expect authors of ontologies to have experience in the use of Isabelle/DOF, basic modeling (and, potentially, some basic SML programming) experience, basic LATEX knowledge, and, last but not least, domain knowledge of the ontology to be modeled. Users with experience in UML-like meta-modeling will feel familiar with most concepts; however, we expect no need for insight in the Isabelle proof language, for example, or other more advanced concepts.

Technically, ontologies are stored in a directory `src/ontologies` and consist of a Isabelle theory file and a LATEX-style file:

```
└─ src
   └─ ontologies ........................................ Ontologies
      ├─ ontologies.thy ......................... Ontology Registration
      ├─ CENELEC_50128 ................................ CENELEC_50128
      │  ├─ CENELEC_50128.thy
      │  └─ DOF-CENELEC_50128.sty
      ├─ scholarly_paper ............................. scholarly_paper
      │  ├─ scholarly_paper.thy
      │  └─ DOF-scholarly_paper.sty
      └─ . . .
```

Developing a new ontology "`foo`" requires, from a technical perspective, the following steps:

- create a new sub-directory `foo` in the directory `src/ontologies`
- definition of the ontological concepts, using Isabelle/DOF's Ontology Definition Language (ODL), in a new theory file src/ontologies/foo/foo.thy.
- definition of the document representation for the ontological concepts in a LATEX-style file src/ontologies/foo/DOF-foo.sty
- registration (as import) of the new ontology in the file. src/ontologies/ontologies.thy.
- activation of the new document setup by executing the install script. You can skip the lengthy checks for the AFP entries and the installation of the Isabelle patch by using the `--skip-patch-and-afp` option:

  **Bash**
  ```
  achim@logicalhacking:~/Isabelle_DOF-1.0.0_Isabelle2019$ ./install  \
      --skip-patch-and-afp
  ```

### 4.1.2 Document Templates

Document-templates define the overall layout (page size, margins, fonts, etc.) of the generated documents and are the the main technical means for implementing layout requirements that are, e.g., required by publishers or standardization bodies. Document-templates are stored in a directory src/document-templates:

```
└─src
  └─document-templates ......................... Document templates
    ├─root-lncs.tex
    ├─root-scrartcl.tex
    ├─root-scrreprt-modern.tex
    └─root-scrreprt.tex
```

Developing a new document template "bar" requires the following steps:

- develop a new LaTeX-template `src/document-templates/root-bar.tex`
- activation of the new document template by executing the install script. You can skip the lengthy checks for the AFP entries and the installation of the Isabelle patch by using the `--skip-patch-and-afp` option:

```bash
achim@logicalhacking:~/Isabelle_DOF-1.0.0_Isabelle2019$ ./install  \
    --skip-patch-and-afp
```

As the document generation of Isabelle/DOF is based on LaTeX, the Isabelle/DOF document templates can (and should) make use of any LaTeX-classes provided by publishers or standardization bodies.

## 4.2 The Ontology Definition Language (ODL)

ODL shares some similarities with meta-modeling languages such as UML class models: It builds upon concepts like class, inheritance, class-instances, attributes, references to instances, and class-invariants. Some concepts like advanced type-checking, referencing to formal entities of Isabelle, and monitors are due to its specific application in the Isabelle context. Conceptually, ontologies specified in ODL consist of:

- *document classes* (**doc_class**) that describe concepts;
- an optional document base class expressing single inheritance class extensions;
- *attributes* specific to document classes, where

  - attributes are HOL-typed;
  - attributes of instances of document elements are mutable;
  - attributes can refer to other document classes, thus, document classes must also be HOL-types (such attributes are called *links*);
  - attribute values were denoted by HOL-terms;
- a special link, the reference to a super-class, establishes an *is-a* relation between classes;
- classes may refer to other classes via a regular expression in a *where* clause;
- attributes may have default values in order to facilitate notation.

The Isabelle/DOF ontology specification language consists basically on a notation for document classes, where the attributes were typed with HOL-types and can be instantiated by terms HOL-terms, i.e., the actual parsers and type-checkers of the Isabelle system were reused. This has the particular advantage that Isabelle/DOF commands can be arbitrarily mixed with Isabelle/HOL commands providing the machinery for type declarations and term specifications such as enumerations. In particular, document class definitions provide:

- a HOL-type for each document class as well as inheritance,
- support for attributes with HOL-types and optional default values,
- support for overriding of attribute defaults but not overloading, and
- text-elements annotated with document classes; they are mutable instances of document classes.

Attributes referring to other ontological concepts are called *links*. The HOL-types inside the document specification language support built-in types for Isabelle/HOL `typ`'s, `term`'s, and `thm`'s reflecting internal Isabelle's internal types for these entities; when denoted in HOL-terms to instantiate an attribute, for example, there is a specific syntax (called *inner syntax antiquotations*) that is checked by Isabelle/DOF for consistency.

Document classes support **where**-clauses containing a regular expression over class names. Classes with a **where** were called *monitor classes*. While document classes and their inheritance relation structure meta-data of text-elements in an object-oriented manner, monitor classes enforce structural organization of documents via the language specified by the regular expression enforcing a sequence of text-elements.

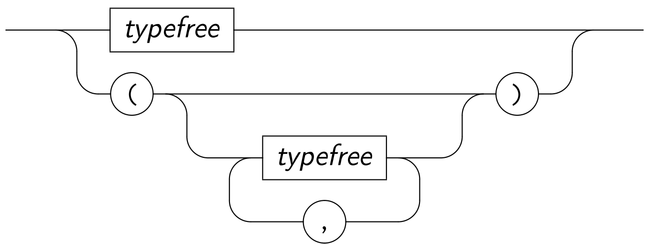
A major design decision of ODL is to denote attribute values by HOL-terms and HOL-types. Consequently, ODL can refer to any predefined type defined in the HOL library, e.g., `string` or `int` as well as parameterized types, e.g., `_ option`, `_ list`, `_ set`, or products `_ × _`. As a consequence of the document model, ODL definitions may be arbitrarily intertwined with standard HOL type definitions. Finally, document class definitions result in themselves in a HOL-types in order to allow *links* to and between ontological concepts.

### 4.2.1 Some Isabelle/HOL Specification Constructs Revisited

As ODL is an extension of Isabelle/HOL, document class definitions can therefore be arbitrarily mixed with standard HOL specification constructs. To make this manual self-contained, we present syntax and semantics of the specification constructs that are most likely relevant for the developer of ontologies (for more details, see [22]. Our presentation is a simplification of the original sources following the needs of ontology developers in Isabelle/DOF:

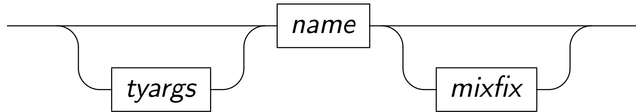
- *name*: with the syntactic category of *name*'s we refer to alpha-numerical identifiers (called *short_id*'s in [22]) and identifiers in . . . which might contain certain "quasi-letters" such as _, -, . (see [22] for details).

- *tyargs*:



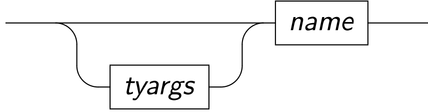

  *typefree* denotes fixed type variable(*'a*, *'b*, ...) (see [22])

- *dt_name*:



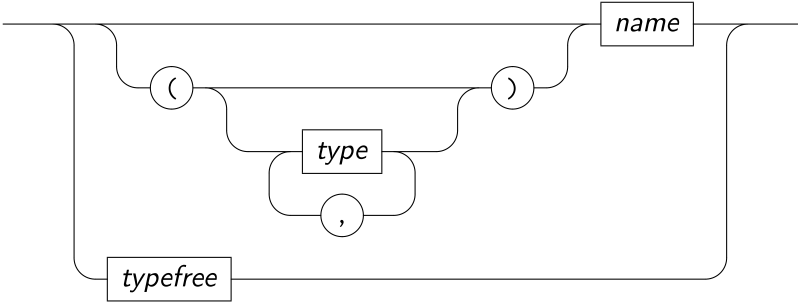

  The syntactic entity *name* denotes an identifier, *mixfix* denotes the usual parenthesized mixfix notation (see [22]). The *name*'s referred here are type names such as `int`, `string`, `list`, `set`, etc.

- *type_spec*:

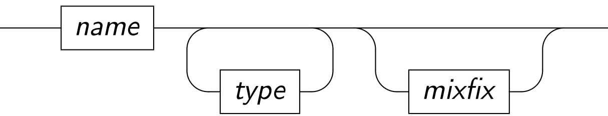



  The *name*'s referred here are type names such as `int`, `string`, `list`, `set`, etc.

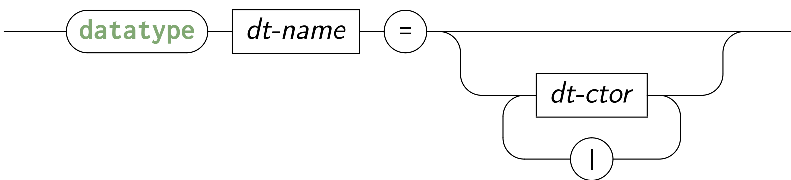
- *type*:


name
(
)
type
,
typefree

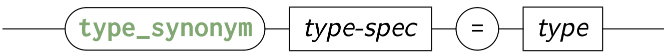
- *dt_ctor*:



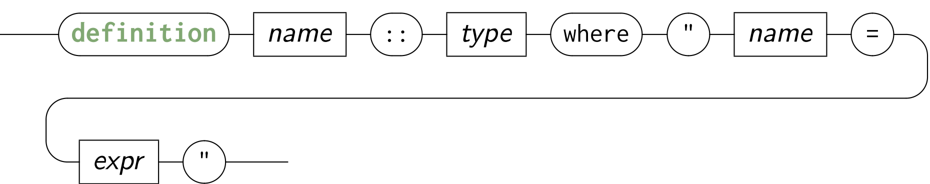

- *datatype_specification*:




- *type_synonym_specification*:




- *constant_definition* :




- *expr*: the syntactic category *expr* here denotes the very rich "inner-syntax" language of mathematical notations for $\lambda$-terms in Isabelle/HOL. Example expressions are: `1+2` (arithmetics), `[1,2,3]` (lists), `''ab c''` (strings), `{1,2,3}` (sets), `(1,2,3)` (tuples), `∀ x. P(x) ∧ Q x = C` (formulas). For details, see [17].

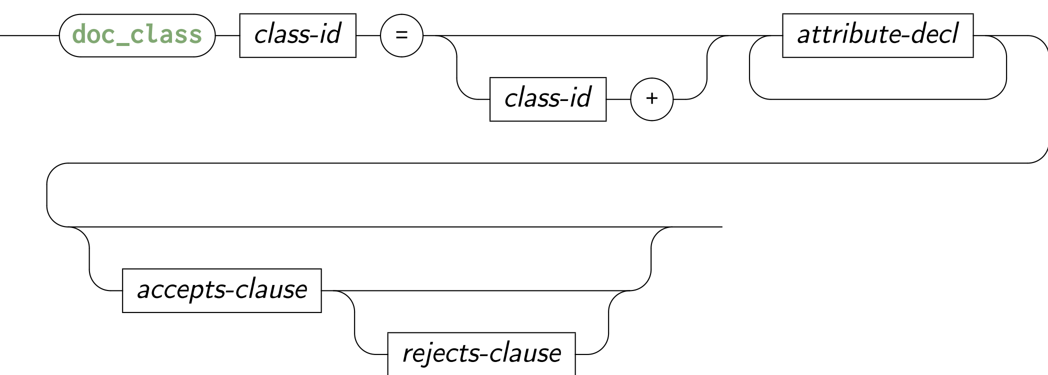
Advanced ontologies can, e.g., use recursive function definitions with pattern-matching [13], extensible record pecifications [22], and abstract type declarations.

Note that Isabelle/DOF works internally with fully qualified names in order to avoid confusions occurring otherwise, for example, in disjoint class hierarchies. This also extends to names for **doc_class**es, which must be representable as type-names as well since they can be used in attribute types. Since theory names are lexically very liberal (`0.thy` is a legal theory name), this can lead to subtle problems when constructing a class: `foo` can be a legal name for a type definition, the corresponding type-name `0.foo` is not. For this reason, additional checks at the definition of a **doc_class** reject problematic lexical overlaps.

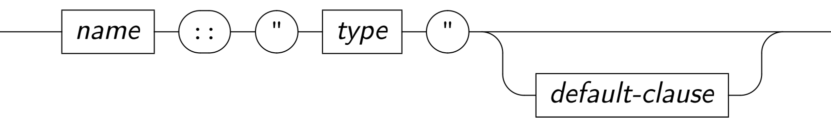
### 4.2.2 Defining Document Classes

A document class can be defined using the **doc_class** keyword:

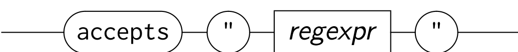
- *class_id*: a type-*name* that has been introduced via a *doc_class_specification*.
- *doc_class_specification*: We call document classes with an *accepts_clause* *monitor classes* or *monitors* for short.



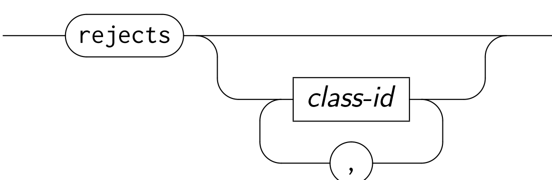

- *attribute_decl*:



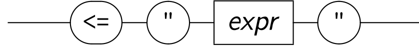

- *accepts_clause*:



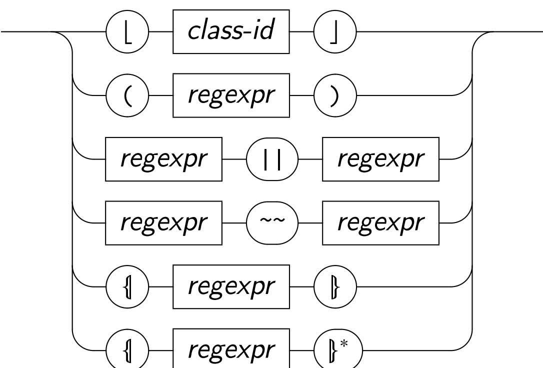

- *rejects_clause*:




- *default_clause*:



- *regexpr*:




Regular expressions describe sequences of *class_ids* (and indirect sequences of document items corresponding to the *class_ids*). The constructors for alternative, sequence, repetitions and non-empty sequence follow in the top-down order of the above diagram.

Isabelle/DOF provides a default document representation (i.e., content and layout of the generated PDF) that only prints the main text, omitting all attributes. Isabelle/DOF provides the `\newisadof[]{}` command for defining a dedicated layout for a document class in LaTeX. Such a document class-specific LaTeX-definition can not only provide a specific layout (e.g., a specific highlighting, printing of certain attributes), it can also generate entries in in the table of contents or an index. Overall, the `\newisadof[]{}` command follows the structure of the `doc_class`-command:

```
\newisadof{class_id}[label=,type=, attribute_decl][1]{%
% LaTeX-definition of the document class representation
\begin{isamarkuptext}%
#1%
\end{isamarkuptext}%
}
```

The *class_id* is the full-qualified name of the document class and the list of *attribute_decl* needs to declare all attributes of the document class. Within the LaTeX-definition of the document class representation, the identifier `#1` refers to the content of the main text of the document class (written in ‹ ... ›) and the attributes can be referenced by their name using the `\commandkey{...}`-command (see the documentation of the LaTeX-package "keycommand" [6] for details). Usually, the representations definition needs to be wrapped in a `\begin{isarmarkup}...\end{isamarkup}`-environment, to ensure the correct context within Isabelle's LaTeX-setup.

Moreover, Isabelle/DOF also provides the following two variants of `\newisadof{}[]{}`:

- `\renewisadof{}[]{}` for re-defining (over-writing) an already defined command, and
- `\provideisadof{}[]{}` for providing a definition if it is not yet defined.

While arbitrary LATEX-commands can be used within these commands, special care is required for arguments containing special characters (e.g., the underscore "_") that do have a special meaning in LATEX. Moreover, as usual, special care has to be taken for commands that write into aux-files that are included in a following LATEX-run. For such complex examples, we refer the interested reader, in general, to the style files provided in the Isabelle/DOF distribution. In particular the definitions of the concepts **title*** and author* in the file ontologies/scholarly_paper/DOF-scholarly_paper.sty show examples of protecting special characters in definitions that need to make use of a entries in an aux-file.

### 4.2.3 Common Ontology Library (COL)

Isabelle/DOF uses the concept of implicit abstract classes (or: *shadow classes*). These refer to the set of possible **doc_class** declarations that posses a number of attributes with their types in common. Shadow classes represent an implicit requirement (or pre-condition) on a given class to posses these attributes in order to work properly for certain Isabelle/DOF commands.

shadow classes will find concrete instances in COL, but Isabelle/DOF text elements do not *depend* on our COL definitions: Ontology developers are free to build own class instances for these shadow classes, with own attributes and, last not least, own definitions of invariants independent from ours.

In particular, these shadow classes are used at present in Isabelle/DOF:

Isar

```
DOCUMENT_ALIKES =
   level         :: int  option    <=  None

ASSERTION_ALIKES =
    properties :: term list

FORMAL_STATEMENT_ALIKE =
    properties :: thm list
```

These shadow-classes correspond to semantic macros `ODL_Command_Parser.enriched_document_command`, `ODL_Command_Parser.assertion_cmd'`, and `ODL_Command_Parser.enriched_formal_statement_command`.

Isabelle/DOF provides a Common Ontology Library (COL) that introduces ontology concepts that are either sample instances for shadow classes as we use them in our own document generation processes or, in some cases, are so generic that they we expect them to be useful for all types of documents (figures, for example).

In particular it defines the super-class `text_element`: the root of all text-elements,

```
Isar
doc_class text_element =
   level         :: int  option    <=  None
   referentiable :: bool <= False
   variants      :: String.literal set <= {STR ''outline'', STR ''document''}
```

Here, `level` defines the section-level (e.g., using a LaTeX-inspired hierarchy: from `Some -1` (corresponding to `\part`) to `Some 0` (corresponding to `\chapter`, respectively, `chapter*`) to `Some 3` (corresponding to `\subsubsection`, respectively, `subsubsection*`). Using an invariant, a derived ontology could, e.g., require that any sequence of technical-elements must be introduced by a text-element with a higher level (this would require that technical text section are introduce by a section element).

Similarly, we provide "minimal" instances of the `ASSERTION_ALIKES` and `FORMAL_STATEMENT_ALIKE` shadow classes:

```
Isar
doc_class assertions =
    properties :: term list

doc_class thms =
    properties :: thm list
```

### Example: Text Elemens with Levels

The category "exported constraint (EC)" is, in the file ontologies/CENELEC_50128/CENELEC_50128.thy defined as follows:

```
Isar
doc_class requirement = text_element +
   long_name     :: string option
   is_concerned :: role set
doc_class AC = requirement +
   is_concerned :: role set <= UNIV
doc_class EC = AC  +
     assumption_kind :: ass_kind <= (*default *) formal
```

We now define the document representations, in the file ontologies/CENELEC_50128/DOF-CENELEC_50128.sty. Let us assume that we want to register the definition of ECs in a dedicated table of contents (`tos`) and use an earlier defined environment `\begin{EC}...\end{EC}` for their graphical representation. Note that the `\newisadof{}[]{}`-command requires the full-qualified names, e.g., `text.CENELEC_50128.EC` for the document class and `CENELEC_50128.requirement.long_name` for the attribute `long_name`, inherited from the document class `requirement`. The representation of ECs can now be defined as follows:

```
LATEX
\newisadof{text.CENELEC_50128.EC}%
[label=,type=%
,Isa_COL.text_element.level=%
,Isa_COL.text_element.referentiable=%
,Isa_COL.text_element.variants=%
,CENELEC_50128.requirement.is_concerned=%
,CENELEC_50128.requirement.long_name=%
,CENELEC_50128.EC.assumption_kind=][1]{%
\begin{isamarkuptext}%
   \ifthenelse{\equal{\commandkey{CENELEC_50128.requirement.long_name}}{}}{%
      % If long_name is not defined, we only create an entry in the table tos
      % using the auto-generated number of the EC
      \begin{EC}%
          \addxcontentsline{tos}{chapter}[]{\autoref{\commandkey{label}}}%
    }{%
      % If long_name is defined, we use the long_name as title in the
      % layout of the EC, in the table "tos" and as index entry. .
      \begin{EC}[\commandkey{CENELEC_50128.requirement.long_name}]%
        \addxcontentsline{toe}{chapter}[]{\autoref{\commandkey{label}}: %
              \commandkey{CENELEC_50128.requirement.long_name}}%
        \DOFindex{EC}{\commandkey{CENELEC_50128.requirement.long_name}}%
    }%
    \label{\commandkey{label}}% we use the label attribute as anchor
    #1% The main text of the EC
  \end{EC}
\end{isamarkuptext}%
}
```

### Example: Assertions

Assertions are a common feature to validate properties of models, presented as a collection of Isabelle/HOL definitions. They are particularly relevant for highlighting corner cases of a formal model. For example, assume a definition:

**definition** *last* :: *'a list* ⇒ *'a* **where** *last S* = *hd*(*rev S*)

We want to check the consequences of this definition and can add the following statements:

```
Isar
text*[claim::assertions]‹For non-empty lists, our definition yields indeed
                              the last element of a list.›
assert*[claim::assertions] last[4::int] = 4
assert*[claim::assertions] last[1,2,3,4::int] = 4
```

As an `ASSERTION_ALIKES`, the `assertions` class possesses a `properties` attribute. The `assert*` command evaluates its argument; in case it evaluates to true the property is added

to the property list of the `claim` - text-element. Commands like `Definitions*` or `Theorem*` work analogously.

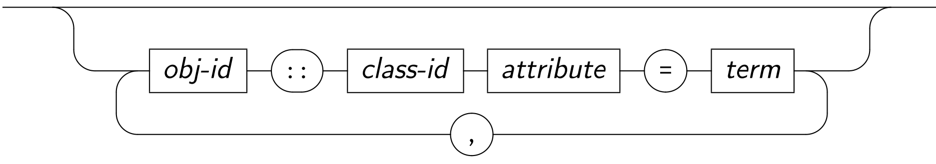
### 4.2.4 Annotatable Top-level Text-Elements

While the default user interface for class definitions via the **text***‹ . . . ›-command allow to access all features of the document class, Isabelle/DOF provides short-hands for certain, widely-used, concepts such as **title***‹ . . . › or **section***‹ . . . ›, e.g.:

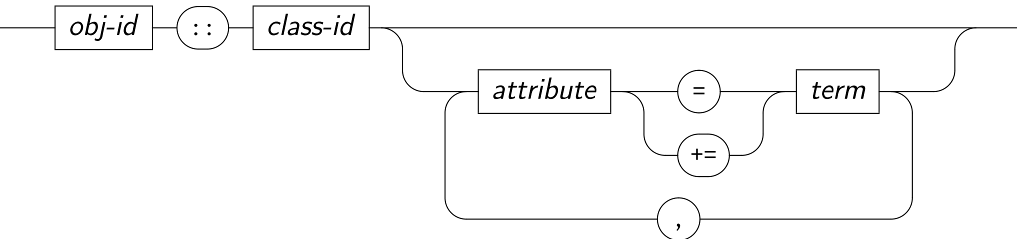
```
Isar
title*[title::title]‹Isabelle/DOF›
subtitle*[subtitle::subtitle]‹User and Implementation Manual›
text*[adb:: author, email=‹a.brucker@exeter.ac.uk›,
orcid=‹0000-0002-6355-1200›, http_site=‹https://brucker.ch/›,
affiliation=‹University of Exeter, Exeter, UK›] ‹Achim D. Brucker›
text*[bu::author, email = ‹wolff@lri.fr›,
affiliation = ‹Université Paris-Saclay, LRI, Paris, France›]‹Burkhart Wolff›
```

In general, all standard text-elements from the Isabelle document model such as **chapter**, **section**, **text**, have in the Isabelle/DOF implementation their counterparts in the family of text-elements that are ontology-aware, i.e., they dispose on a meta-argument list that allows to define that a test-element that has an identity as a text-object labelled as *obj_id*, belongs to a document class *class_id* that has been defined earlier, and has its class-attributes set with particular values (which are denotable in Isabelle/HOL mathematical term syntax).

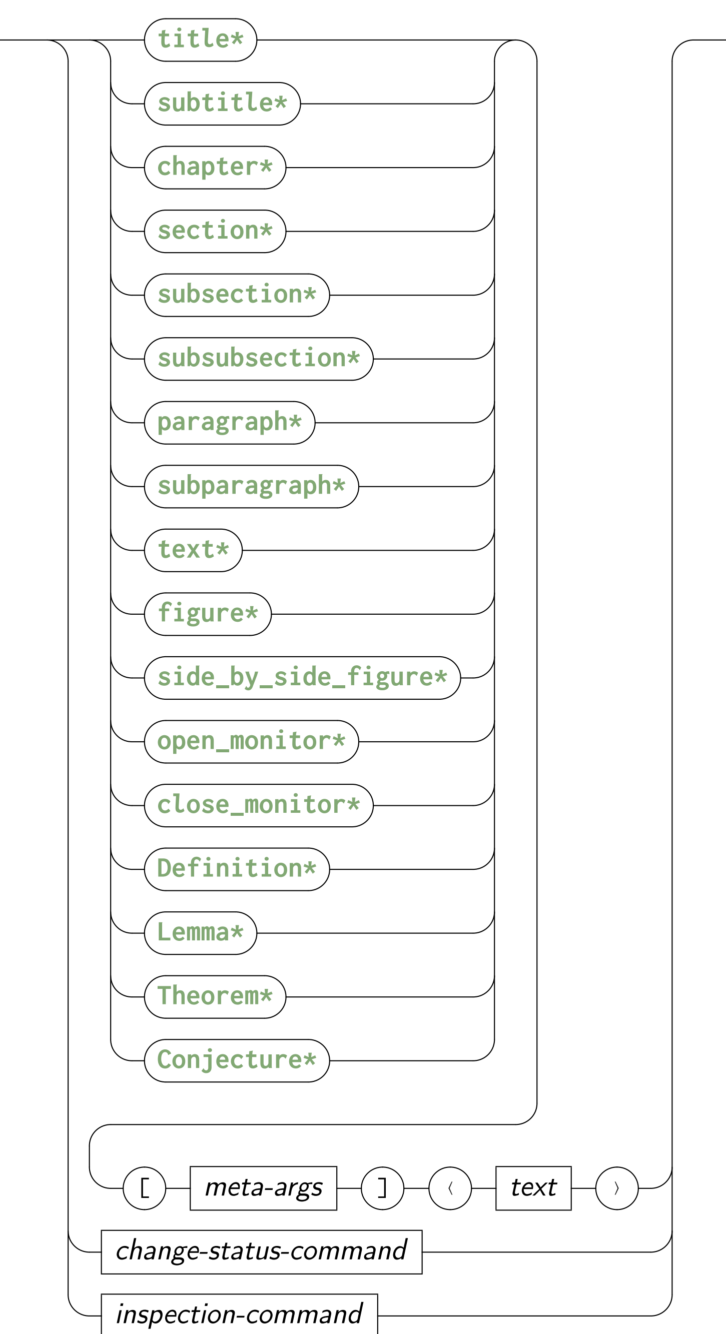
- *meta_args* :




- *rich_meta_args* :



- *annotated_text_element* :




### Experts: Defining New Top-Level Commands

Defining such new top-level commands requires some Isabelle knowledge as well as extending the dispatcher of the LaTeX-backend. For the details of defining top-level commands, we refer

the reader to the Isar manual [22]. Here, we only give a brief example how the **section***-command is defined; we refer the reader to the source code of Isabelle/DOF for details.

First, new top-level keywords need to be declared in the **keywords**-section of the theory header defining new keywords:

```
Isar
theory
    ...
  imports
    ...
  keywords
    section*
begin
...
end
```

Second, given an implementation of the functionality of the new keyword (implemented in SML), the new keyword needs to be registered, together with its parser, as outer syntax:

```
SML
val _ =
  Outer_Syntax.command ("section*", @{here}) "section_heading"
    (attributes -- Parse.opt_target -- Parse.document_source --| semi
      >> (Toplevel.theory o (enriched_document_command (SOME(SOME 1))
           {markdown = false} )));
```

Finally, for the document generation, a new dispatcher has to be defined in LaTeX—this is mandatory, otherwise the document generation will break. These dispatcher always follow the same schemata:

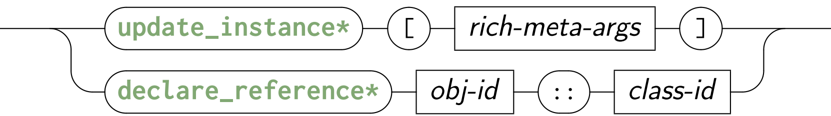
```
LaTeX
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% begin: section*-dispatcher
\NewEnviron{isamarkupsection*}[1][]{\isaDof[env={section},#1]{\BODY}}
% end: section*-dispatcher
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

After the definition of the dispatcher, one can, optionally, define a custom representation using the **newisadof**-command, as introduced in the previous section:

```
LaTeX
\newisadof{section}[label=,type=][1]{%
  \isamarkupfalse%
    \isamarkupsection{#1}\label{\commandkey{label}}%
  \isamarkuptrue%
}
```

### 4.2.5 Status and Inspection Commands

- Isabelle/DOF *change_status_command* :




- Isabelle/DOF *inspection_command* :

print_doc_classes
print_doc_items
check_doc_global

### 4.2.6 Advanced ODL Concepts

#### Meta-types as Types

To express the dependencies between text elements to the formal entities, e.g., `term` ($\lambda$-term), `typ`, or `thm`, we represent the types of the implementation language *inside* the HOL type system. We do, however, not reflect the data of these types. They are just declared abstract types, "inhabited" by special constant symbols carrying strings, for example of the format `@{thm <string>}`. When HOL expressions were used to denote values of `doc_class` instance attributes, this requires additional checks after conventional type-checking that this string represents actually a defined entity in the context of the system state $\theta$. For example, the `establish` attribute in the previous section is the power of the ODL: here, we model a relation between *claims* and *results* which may be a formal, machine-check theorem of type `thm` denoted by, for example: `property = [@{thm ''system_is_safe''}]` in a system context $\theta$ where this theorem is established. Similarly, attribute values like `property = @{term ‹A ↔ B›}` require that the HOL-string `A ↔ B` is again type-checked and represents indeed a formula in $\theta$. Another instance of this process, which we call *second−level type−checking*, are term-constants generated from the ontology such as `@{definition <string>}`.

#### ODL Monitors

We call a document class with an accept-clause a *monitor*. Syntactically, an accept-clause contains a regular expression over class identifiers. For example:

```
Isar
doc_class article = style_id :: string   <= ''CENELEC_50128''
accepts (title    ⦃author⦄\<^sup>+    abstract    ⦃introduction⦄\<^sup>+
⦃technical || example⦄\<^sup>+    ⦃conclusion⦄\<^sup>+)
```

Semantically, monitors introduce a behavioral element into ODL:

```
Isar
open_monitor*[this::article]  (* begin of scope of monitor this *)
 . . .
close_monitor*[this]          (* end of scope of monitor this   *)
```

Inside the scope of a monitor, all instances of classes mentioned in its accept-clause (the *accept-set*) have to appear in the order specified by the regular expression; instances not covered by an accept-set may freely occur. Monitors may additionally contain a reject-clause with a list of class-ids (the reject-list). This allows specifying ranges of admissible instances along the class hierarchy:

- a superclass in the reject-list and a subclass in the accept-expression forbids instances superior to the subclass, and
- a subclass $S$ in the reject-list and a superclass $T$ in the accept-list allows instances of superclasses of $T$ to occur freely, instances of $T$ to occur in the specified order and forbids instances of $S$.

Monitored document sections can be nested and overlap; thus, it is possible to combine the effect of different monitors. For example, it would be possible to refine the `example` section by its own monitor and enforce a particular structure in the presentation of examples.

Monitors manage an implicit attribute `trace` containing the list of "observed" text element instances belonging to the accept-set. Together with the concept of ODL class invariants, it is possible to specify properties of a sequence of instances occurring in the document section. For example, it is possible to express that in the sub-list of `introduction`-elements, the first has an `introduction` element with a `level` strictly smaller than the others. Thus, an introduction is forced to have a header delimiting the borders of its representation. Class invariants on monitors allow for specifying structural properties on document sections.

### ODL Class Invariants

Ontological classes as described so far are too liberal in many situations. For example, one would like to express that any instance of a `result` class finally has a non-empty property list, if its `kind` is `proof`, or that the `establish` relation between `claim` and `result` is surjective.

In a high-level syntax, this type of constraints could be expressed, e.g., by:

```
Isar
(* 1 *) ∀ x ∈ result. x@kind = proof ↔ x@kind ≠ []
(* 2 *) ∀ x ∈ conclusion. ∀ y ∈ Domain(x@establish)
                    → ∃ y ∈ Range(x@establish). (y,z) ∈ x@establish
(* 3 *) ∀ x ∈ introduction. finite(x@authored_by)
```

where `result`, `conclusion`, and `introduction` are the set of all possible instances of these document classes. All specified constraints are already checked in the IDE of DOF

while editing; it is however possible to delay a final error message till the closing of a monitor (see next section). The third constraint enforces that the user sets the authored_by set, otherwise an error will be reported.

For the moment, there is no high-level syntax for the definition of class invariants. A formulation, in SML, of the first class-invariant in Section 4.2.3 is straight-forward:

```
SML
fun check_result_inv oid {is_monitor:bool} ctxt =
  let val kind = compute_attr_access ctxt "kind" oid @{here} @{here}
      val prop = compute_attr_access ctxt "property" oid @{here} @{here}
      val tS = HOLogic.dest_list prop
  in  case kind_term of
       @{term "proof"} => if not(null tS) then true
                          else error("class_result_invariant_violation")
      | _ => false
  end
 val _ = Theory.setup (DOF_core.update_class_invariant
                           "tiny_cert.result" check_result_inv)
```

The setup-command (last line) registers the check_result_inv function into the Isabelle/DOF kernel, which activates any creation or modification of an instance of result. We cannot replace compute_attr_access by the corresponding antiquotation @{docitem_value kind::oid}, since oid is bound to a variable here and can therefore not be statically expanded.

## 4.3 Defining Document Templates

### 4.3.1 The Core Template

Document-templates define the overall layout (page size, margins, fonts, etc.) of the generated documents and are the the main technical means for implementing layout requirements that are, e.g., required by publishers or standardization bodies. If a new layout is already supported by a LaTeX-class, then developing basic support for it is straight forwards: after reading the authors guidelines of the new template, Developing basic support for a new document template is straight forwards In most cases, it is sufficient to replace the document class in Line 1 of the template and add the LaTeX-packages that are (strictly) required by the used LaTeX-setup. In general, we recommend to only add LaTeX-packages that are always necessary fro this particular template, as loading packages in the templates minimizes the freedom users have by adapting the preample.tex. Moreover, you might want to add/modify the template specific configuration (Line 22-24). The new template should be stored in src/document-templates and its file name should start with the prefix root-. After adding a new template, call the install script (see Section 4.1 The common structure of an Isabelle/DOF document template looks as follows:

LATEX
```
\documentclass{article}     % The LaTeX-class of your template
%% The following part is (mostly) required by Isabelle/DOF, do not modify
\usepackage[T1]{fontenc}    % Font encoding
\usepackage[utf8]{inputenc} % UTF8 support
\usepackage{xcolor}
\usepackage{isabelle,isabellesym,amssymb} % Required (by Isabelle)
\usepackage{amsmath}        % Used by some ontologies
\bibliographystyle{abbrv}
\IfFileExists{DOF-core.sty}{}{ % Required by Isabelle/DOF
  \PackageError{DOF-core}{The document preparation
   requires the Isabelle/DOF framework.}{For further help, see
   https://git.logicalhacking.com/Isabelle_DOF/Isabelle_DOF
}
\input{ontologies}          % This will include the document specific
                            % ontologies from isadof.cfg
\IfFileExists{preamble.tex}{\input{preamble.tex}}{}
\usepackage{graphicx}       % Required for images.
\usepackage[caption]{subfig}
\usepackage[size=footnotesize]{caption}
\usepackage{hyperref}       % Required by Isabelle/DOF

%% Begin of template specific configuration
\urlstyle{rm}
\isabellestyle{it}

%% Main document, do not modify
\begin{document}
\maketitle\input{session}
\IfFileExists{root.bib}{\bibliography{root}}{}
\end{document}
```

### 4.3.2 Tips, Tricks, and Known Limitations

In this sectin, we sill discuss several tips and tricks for developing new or adapting existing document templates or LATEX-represenations of ontologies.

#### Getting Started

In general, we recommend to create a test project (e.g., using `isabelle mkroot_DOF`) to develop new document templates or ontology representations. The default setup of the Isabelle/DOF build system generated a output/document directory with a self-contained LATEX-setup. In this directory, you can directly use LATEX on the main file, called root.tex:

Bash
```
achim@logicalhacking:~/MyProject/output/document$ pdflatex root.tex
```

This allows you to develop and check your LATEX-setup without the overhead of running `isabelle build` after each change of your template (or ontology-style). Note that the content of the output directory is overwritten by executing `isabelle build`.

### Truncated Warning and Error Messages

By default, LATEX cuts of many warning or error messages after 79 characters. Due to the use of full-qualified names in Isabelle/DOF, this can often result in important information being cut off. Thus, it can be very helpful to configure LATEX in such a way that it prints long error or warning messages. This can easily be done on the command line for individual LATEX invocations:

```bash
achim@logicalhacking:~/MyProject/output/document$ max_print_line=200 \
    error_line=200 half_error_line=100 pdflatex root.tex
```

### Deferred Declaration of Information

During document generation, sometimes, information needs to be printed prior to its declaration in a Isabelle/DOF theory. This violation the declaration-before-use-principle requires that information is written into an auxiliary file during the first run of LATEX so that the information is available at further runs of LATEX. While, on the one hand, this is a standard process (e.g., used for updating references), implementing it correctly requires a solid understanding of LATEX's expansion mechanism. In this context, the recently introduced `\expanded{}`-primitive (see https://www.texdev.net/2018/12/06/a-new-primitive-expanded) is particularly useful. Examples of its use can be found, e.g., in the ontology-styles ontologies/scholarly_paper/DOF-scholarly_paper.sty or ontologies/CENELEC_50128/DOF-CENELEC_50128.sty. For details about the expansion mechanism in general, we refer the reader to the LATEX literature (e.g., [8, 11, 15]).

### Authors and Affiliation Information

In the context of academic papers, the defining the representations for the author and affiliation information is particularly challenges as, firstly, they inherently are breaking the declare-before-use-principle and, secondly, each publisher uses a different LATEX-setup for their declaration. Moreover, the mapping from the ontological modeling to the document representation might also need to bridge the gap between different common modeling styles of authors and their affiliations, namely: affiliations as attributes of authors vs. authors and affiliations both as entities with a many-to-many relationship.

The ontology representation ontologies/scholarly_paper/DOF-scholarly_paper.sty contains an example that, firstly, shows how to write the author and affiliation information into the auxiliary file for re-use in the next LATEX-run and, secondly, shows how to collect the author and affiliation information into an `\author` and a `\institution` statement, each of

which containing the information for all authors. The collection of the author information is provided by the following LaTeX-code:

```
\def\dof@author{}%
\newcommand{\DOFauthor}{\author{\dof@author}}
\AtBeginDocument{\DOFauthor}
\def\leftadd#1#2{\expandafter\leftaddaux\expandafter{#1}{#2}{#1}}
\def\leftaddaux#1#2#3{\gdef#3{#1#2}}
\newcounter{dof@cnt@author}
\newcommand{\addauthor}[1]{%
  \ifthenelse{\equal{\dof@author}{}}{%
    \gdef\dof@author{#1}%
  }{%
    \leftadd\dof@author{\protect\and #1}%
  }
}
```

The new command `\addauthor` and a similarly defined command `\addaffiliation` can now be used in the definition of the representation of the concept **text**.scholarly_paper.author, which writes the collected information in the job's aux-file. The intermediate step of writing this information into the job's aux-file is necessary, as the author and affiliation information is required right at the begin of the document (i.e., when LaTeX's `\maketitle` is invoked) while Isabelle/DOF allows to define authors at any place within a document:

```
\provideisadof{text.scholarly_paper.author}%
[label=,type=%
,scholarly_paper.author.email=%
,scholarly_paper.author.affiliation=%
,scholarly_paper.author.orcid=%
,scholarly_paper.author.http_site=%
][1]{%
  \stepcounter{dof@cnt@author}
  \def\dof@a{\commandkey{scholarly_paper.author.affiliation}}
  \ifthenelse{\equal{\commandkey{scholarly_paper.author.orcid}}{}}{%
    \immediate\write\@auxout%
       {\noexpand\addauthor{#1\noexpand\inst{\thedof@cnt@author}}}%
  }{%
    \immediate\write\@auxout%
        {\noexpand\addauthor{#1\noexpand%
            \inst{\thedof@cnt@author}%
                 \orcidID{\commandkey{scholarly_paper.author.orcid}}}}%
  }
  \protected@write\@auxout{}{%
             \string\addaffiliation{\dof@a\\\string\email{%
                  \commandkey{scholarly_paper.author.email}}}}%
}
```

Finally, the collected information is used in the \author command using the AtBeginDocument -hook:

LaTeX
```
\newcommand{\DOFauthor}{\author{\dof@author}}
\AtBeginDocument{%
  \DOFauthor
}
```

### Restricting the Use of Ontologies to Specific Templates

As ontology representations might rely on features only provided by certain templates (LaTeX-classes), authors of ontology representations might restrict their use to specific classes. This can, e.g., be done using the \@ifclassloaded{} command:

LaTeX
```
\@ifclassloaded{llncs}{}%
{% LLNCS class not loaded
    \PackageError{DOF-scholarly_paper}
    {Scholarly Paper only supports LNCS as document class.}{}\stop%
}
```

For a real-world example testing for multiple classes, see ontologies/scholarly_paper/DOF-scholarly_paper.sty):

We encourage this clear and machine-checkable enforcement of restrictions while, at the same time, we also encourage to provide a package option to overwrite them. The latter allows inherited ontologies to overwrite these restrictions and, therefore, to provide also support for additional document templates. For example, the ontology technical_report extends the scholarly_paper ontology and its LaTeXsupports provides support for the scrrept-class which is not supported by the LaTeXsupport for scholarly_paper.

### Outdated Version of comment.sty

Isabelle's LaTeX-setup relies on an ancient version of comment.sty that, moreover, is used in plainTeX-mode. This is known to cause issues with some modern LaTeX-classes such as LPICS. Such a conflict might require the help of an Isabelle wizard.

# 5 Extending Isabelle/DOF

In this chapter, we describe the basic implementation aspects of Isabelle/DOF, which is based on the following design-decisions:

- the entire Isabelle/DOF is a "pure add-on," i.e., we deliberately resign on the possibility to modify Isabelle itself.
- we made a small exception to this rule: the Isabelle/DOF package modifies in its installation about 10 lines in the LaTeX-generator (src/patches/thy_output.ML).
- we decided to make the markup-generation by itself to adapt it as well as possible to the needs of tracking the linking in documents.
- Isabelle/DOF is deeply integrated into the Isabelle's IDE (PIDE) to give immediate feedback during editing and other forms of document evolution.

Semantic macros, as required by our document model, are called *document antiquotations* in the Isabelle literature [22]. While Isabelle's code-antiquotations are an old concept going back to Lisp and having found via SML and OCaml their ways into modern proof systems, special annotation syntax inside documentation comments have their roots in documentation generators such as Javadoc. Their use, however, as a mechanism to embed machine-checked *formal content* is usually very limited and also lacks IDE support.

## 5.1 Isabelle/DOF: A User-Defined Plugin in Isabelle/Isar

A plugin in Isabelle starts with defining the local data and registering it in the framework. As mentioned before, contexts are structures with independent cells/compartments having three primitives `init`, `extend` and `merge`. Technically this is done by instantiating a functor `Generic_Data`, and the following fairly typical code-fragment is drawn from Isabelle/DOF:

SML

```
structure Data = Generic_Data
(  type T = docobj_tab * docclass_tab * ...
   val empty = (initial_docobj_tab, initial_docclass_tab, ...)
   val extend = I
   fun merge((d1,c1,...),(d2,c2,...)) = (merge_docobj_tab (d1,d2,...),
                                         merge_docclass_tab(c1,c2,...))
);
```

where the table `docobj_tab` manages document classes and `docclass_tab` the environment for class definitions (inducing the inheritance relation). Other tables capture, e.g.,

the class invariants, inner-syntax antiquotations. Operations follow the MVC-pattern, where Isabelle/Isar provides the controller part. A typical model operation has the type:

SML
```
val opn :: <args_type> -> Context.generic -> Context.generic
```

representing a transformation on system contexts. For example, the operation of declaring a local reference in the context is presented as follows:

SML
```
fun declare_object_local oid ctxt =
let fun decl {tab,maxano} = {tab=Symtab.update_new(oid,NONE) tab,
                             maxano=maxano}
in  (Data.map(apfst decl)(ctxt)
  handle Symtab.DUP _ =>
             error("multiple␣declaration␣of␣document␣reference"))
end
```

where Data.map is the update function resulting from the instantiation of the functor Generic_Data. This code fragment uses operations from a library structure Symtab that were used to update the appropriate table for document objects in the plugin-local state. Possible exceptions to the update operation were mapped to a system-global error reporting function.

Finally, the view-aspects were handled by an API for parsing-combinators. The library structure Scan provides the operators:

SML
```
op ||    : ('a -> 'b) * ('a -> 'b) -> 'a -> 'b
op --    : ('a -> 'b * 'c) * ('c -> 'd * 'e) -> 'a -> ('b * 'd) * 'e
op >>    : ('a -> 'b * 'c) * ('b -> 'd) -> 'a -> 'd * 'c
op option : ('a -> 'b * 'a) -> 'a -> 'b option * 'a
op repeat : ('a -> 'b * 'a) -> 'a -> 'b list * 'a
```

for alternative, sequence, and piping, as well as combinators for option and repeat. Parsing combinators have the advantage that they can be smoothlessly integrated into standard programs, and they enable the dynamic extension of the grammar. There is a more high-level structure Parse providing specific combinators for the command-language Isar:

SML
```
val attribute = Parse.position Parse.name
             -- Scan.optional(Parse.$$$ "=" |-- Parse.!!! Parse.name)"";
val reference = Parse.position Parse.name
             -- Scan.option (Parse.$$$ "::" |-- Parse.!!!
                              (Parse.position Parse.name));
val attributes =(Parse.$$$ "[" |-- (reference
              -- (Scan.optional(Parse.$$$ ","
                  |--(Parse.enum ","attribute)))[]))--| Parse.$$$ "]"
```

The "model" declare_reference_opn and "new" attributes parts were combined via the

piping operator and registered in the Isar toplevel:

SML
```
fun declare_reference_opn (((oid,_),_),_) =
                (Toplevel.theory (DOF_core.declare_object_global oid))
  val _ = Outer_Syntax.command @{command_keyword "declare_reference"}
          "declare_document_reference"
          (attributes >> declare_reference_opn);
```

Altogether, this gives the extension of Isabelle/HOL with Isar syntax and semantics for the new *command*:

Isar
```
declare_reference [lal::requirement, alpha=main, beta=42]
```

The construction also generates implicitly some markup information; for example, when hovering over the **declare_reference** command in the IDE, a popup window with the text: "declare document reference" will appear.

## 5.2 Programming Antiquotations

The definition and registration of text antiquotations and ML-antiquotations is similar in principle: based on a number of combinators, new user-defined antiquotation syntax and semantics can be added to the system that works on the internal plugin-data freely. For example, in

SML
```
val _ = Theory.setup(
          Thy_Output.antiquotation @{binding docitem}
                                   docitem_antiq_parser
                                   (docitem_antiq_gen default_cid) #>
          ML_Antiquotation.inline @{binding docitem_value}
                                   ML_antiq_docitem_value)
```

the text antiquotation `docitem` is declared and bounded to a parser for the argument syntax and the overall semantics. This code defines a generic antiquotation to be used in text elements such as

Isar
```
text‹as defined in @{docitem ‹d1›} ...›
```

The subsequent registration `docitem_value` binds code to a ML-antiquotation usable in an ML context for user-defined extensions; it permits the access to the current "value" of document element, i.e.; a term with the entire update history.

It is possible to generate antiquotations *dynamically*, as a consequence of a class definition in ODL. The processing of the ODL `class definition` also *generates* a text antiquotation **@{definition ‹d1›}**, which works similar to **@{docitem ‹d1›}** except for an additional type-

check that assures that d1 is a reference to a definition. These type-checks support the subclass hierarchy.

## 5.3 Implementing Second-level Type-Checking

On expressions for attribute values, for which we chose to use HOL syntax to avoid that users need to learn another syntax, we implemented an own pass over type-checked terms. Stored in the late-binding table ISA_transformer_tab, we register for each inner-syntax-annotation (ISA's), a function of type

SML
```
theory -> term * typ * Position.T -> term option
```

Executed in a second pass of term parsing, ISA's may just return None. This is adequate for ISA's just performing some checking in the logical context **theory**; ISA's of this kind report errors by exceptions. In contrast, *transforming* ISA's will yield a term; this is adequate, for example, by replacing a string-reference to some term denoted by it. This late-binding table is also used to generate standard inner-syntax-antiquotations from a **doc_class**.

## 5.4 Programming Class Invariants

For the moment, there is no high-level syntax for the definition of class invariants. A formulation, in SML, of the first class-invariant in Section 4.2.3 is straight-forward:

SML
```
fun check_result_inv oid {is_monitor:bool} ctxt =
  let val kind = compute_attr_access ctxt "kind" oid @{here} @{here}
      val prop = compute_attr_access ctxt "property" oid @{here} @{here}
      val tS = HOLogic.dest_list prop
  in  case kind_term of
       @{term "proof"} => if not(null tS) then true
                          else error("class_result_invariant_violation")
      | _ => false
  end
 val _ = Theory.setup (DOF_core.update_class_invariant
                             "tiny_cert.result" check_result_inv)
```

The setup-command (last line) registers the check_result_inv function into the Isabelle/DOF kernel, which activates any creation or modification of an instance of result. We cannot replace compute_attr_access by the corresponding antiquotation @{**docitem_value** kind::oid}, since oid is bound to a variable here and can therefore not be statically expanded.

## 5.5 Implementing Monitors

Since monitor-clauses have a regular expression syntax, it is natural to implement them as deterministic automata. These are stored in the docobj_tab for monitor-objects in the Isabelle/DOF component. We implemented the functions:

SML
```
val enabled : automaton -> env -> cid list
   val next    : automaton -> env -> cid -> automaton
```

where env is basically a map between internal automaton states and class-id's (cid's). An automaton is said to be *enabled* for a class-id, iff it either occurs in its accept-set or its reject-set (see Section 4.2.3). During top-down document validation, whenever a text-element is encountered, it is checked if a monitor is *enabled* for this class; in this case, the **next**-operation is executed. The transformed automaton recognizing the rest-language is stored in docobj_tab if possible; otherwise, if **next** fails, an error is reported. The automata implementation is, in large parts, generated from a formalization of functional automata [16].

## 5.6 The LaTeX-Core of Isabelle/DOF

The LaTeX-implementation of Isabelle/DOF heavily relies on the "keycommand" [6] package. In fact, the core Isabelle/DOF LaTeX-commands are just wrappers for the corresponding commands from the keycommand package:

LaTeX
```
\newcommand\newisadof[1]{%
  \expandafter\newkeycommand\csname isaDof.#1\endcsname}%
\newcommand\renewisadof[1]{%
  \expandafter\renewkeycommand\csname isaDof.#1\endcsname}%
\newcommand\provideisadof[1]{%
  \expandafter\providekeycommand\csname isaDof.#1\endcsname}%
```

The LaTeX-generator of Isabelle/DOF maps each doc_item to an LaTeX-environment (recall Section 4.2.4). As generic doc_item are derived from the text element, the enviornment {isamarkuptext*} builds the core of Isabelle/DOF's LaTeX implementation. For example, the SRAC 1 from page 25 is mapped to

LaTeX
```
\begin{isamarkuptext*}%
[label = {ass122},type = {CENELEC_50128.SRAC},
 args={label = {ass122}, type = {CENELEC_50128.SRAC},
       CENELEC_50128.EC.assumption_kind = {formal}}
] The overall sampling frequence of the odometer subsystem is therefore
 14 khz, which includes sampling, computing and result communication
 times ...
\end{isamarkuptext*}
```

This environment is mapped to a plain LATEXcommand via (again, recall Section 4.2.4):

```
\NewEnviron{isamarkuptext*}[1][]{\isaDof[env={text},#1]{\BODY}}
```

For the command-based setup, Isabelle/DOF provides a dispatcher that selects the most specific implementation for a given **doc_class**:

```
%% The Isabelle/DOF dispatcher:
\newkeycommand+[\|]\isaDof[env={UNKNOWN},label=,type={dummyT},args={}][1]{%
  \ifcsname isaDof.\commandkey{type}\endcsname%
      \csname isaDof.\commandkey{type}\endcsname%
              [label=\commandkey{label},\commandkey{args}]{#1}%
  \else\relax\fi%
  \ifcsname isaDof.\commandkey{env}.\commandkey{type}\endcsname%
      \csname isaDof.\commandkey{env}.\commandkey{type}\endcsname%
              [label=\commandkey{label},\commandkey{args}]{#1}%
  \else%
      \message{Isabelle/DOF: Using default LaTeX representation for concept %
        "\commandkey{env}.\commandkey{type}".}%
      \ifcsname isaDof.\commandkey{env}\endcsname%
         \csname isaDof.\commandkey{env}\endcsname%
                [label=\commandkey{label}]{#1}%
      \else%
      \errmessage{Isabelle/DOF: No LaTeX representation for concept %
        "\commandkey{env}.\commandkey{type}" defined and no default %
        definition for "\commandkey{env}" available either.}%
      \fi%
  \fi%
}
```

# Bibliography


[1] B. Barras, L. D. C. González-Huesca, H. Herbelin, Y. Régis-Gianas, E. Tassi, M. Wenzel, and B. Wolff. Pervasive parallelism in highly-trustable interactive theorem proving systems. In *MKM*, pages 359–363, 2013. doi: 10.1007/978-3-642-39320-4_29.

[2] S. Bezzecchi, P. Crisafulli, C. Pichot, and B. Wolff. Making agile development processes fit for v-style certification procedures. In *ERTS'18*, ERTS Conference Proceedings, 2018.

[3] J.-L. Boulanger. *CENELEC 50128 and IEC 62279 Standards*. Wiley-ISTE, Boston, 2015.

[4] A. D. Brucker and B. Wolff. Isabelle/DOF: Design and implementation. In P. C. Ölveczky and G. Salaün, editors, *Software Engineering and Formal Methods (SEFM)*, number 11724 in Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, 2019. ISBN 3-540-25109-X. doi: 10.1007/978-3-030-30446-1_15. URL https://www.brucker.ch/bibliography/abstract/brucker.ea-isabelledof-2019.

[5] A. D. Brucker, I. Ait-Sadoune, P. Crisafulli, and B. Wolff. Using the Isabelle ontology framework: Linking the formal with the informal. In *Conference on Intelligent Computer Mathematics (CICM)*, number 11006 in Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, 2018. doi: 10.1007/978-3-319-96812-4_3. URL https://www.brucker.ch/bibliography/abstract/brucker.ea-isabelle-ontologies-2018.

[6] F. Chervet. The free and open source keycommand package: key-value interface for commands and environments in LATEX., 2010.

[7] Common Criteria. Common criteria for information technology security evaluation (version 3.1), Part 3: Security assurance components, Sept. 2006. Available as document CCMB-2006-09-003.

[8] V. Eijkhout. *The Computer Science of TeX and LaTeX*. 2012.

[9] A. Faithfull, J. Bengtson, E. Tassi, and C. Tankink. Coqoon. *Int. J. Softw. Tools Technol. Transf.*, 20(2):125–137, Apr. 2018. ISSN 1433-2779. doi: 10.1007/s10009-017-0457-2.

[10] IBM. IBM engineering requirements management DOORS family, 2019. https://www.ibm.com/us-en/marketplace/requirements-management.

[11] D. E. Knuth. *The TeXbook*. Addison-Wesley Professional, 1986. ISBN 0201134470.

[12] M. Kohm. KOMA-Script: a versatile LATEX $2_\varepsilon$ bundle, 2019.

[13] A. Kraus. Defining recursive functions in isabelle/hol, 2019. https://isabelle.in.tum.de/doc/functions.pdf.

[14] A. Krauss and T. Nipkow. Regular sets and expressions. *Archive of Formal Proofs*, May 2010. ISSN 2150-914x. http://isa-afp.org/entries/Regular-Sets.html, Formal proof development.

[15] F. Mittelbach, M. Goossens, J. Braams, D. Carlisle, and C. Rowley. *The LaTeX Companion*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2004.

[16] T. Nipkow. Functional automata. *Archive of Formal Proofs*, Mar. 2004. ISSN 2150-914x. http://isa-afp.org/entries/Functional-Automata.html, Formal proof development.

[17] T. Nipkow. What's in main, 2019. https://isabelle.in.tum.de/doc/main.pdf.

[18] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. doi: 10.1007/3-540-45949-9.

[19] W3C. Ontologies, 2015. URL https://www.w3.org/standards/semanticweb/ontology.

[20] M. Wenzel. Asynchronous user interaction and tool integration in Isabelle/PIDE. In G. Klein and R. Gamboa, editors, *ITP*, volume 8558 of *LNCS*, pages 515–530. Springer, 2014. doi: 10.1007/978-3-319-08970-6_33.

[21] M. Wenzel. System description: Isabelle/jEdit in 2014. In *UITP*, pages 84–94, 2014. doi: 10.4204/EPTCS.167.10.

[22] M. Wenzel. *The Isabelle/Isar Reference Manual*, 2019. Part of the Isabelle distribution.

[23] M. Wenzel. The Isabelle system manual, 2019. Part of the Isabelle distribution.

[24] M. Wenzel and B. Wolff. Building formal method tools in the Isabelle/Isar framework. In K. Schneider and J. Brandt, editors, *TPHOLs 2007*, number 4732 in LNCS, pages 352–367. Springer, 2007. doi: 10.1007/978-3-540-74591-4_26.

# Index


## A
accept-clause, 46
*accepts_clause*, 38
antiquotation, **13**
*attribute_decl*, 38

## C
class
  see document class, 34
  see monitor class, 34
*class_id*, 38
COL, **40**
see COL, 40
*constant_definition*, 37
context, 12

## D
*datatype_specification*, 37
*default_clause*, 38
*doc_class_specification*, 38
document class, **34**, **37**
  PDF, 39
document model, 12
document template, 18, 33, **48**
  directory structure, 33
*dt_ctor*, 37
*dt_name*, 35

## E
*expr*, 37

## H
header, **12**

## I
inner syntax, *see* syntax, inner
Isabelle, **15**

## M
mkroot_DOF, 18
monitor, **46**
monitor class, **34**

## N
*name*, 35
\newisadof , 39

## O
ontology
  CENELEC_50128, 23
  directory structure, 32
  math_exam, 26
  scholarly_paper, 19
outer syntax, *see* syntax, outer

## P
praemble.tex, 19
\provideisadof , 39

## R
*regexpr*, 39
*rejects_clause*, 38
\renewisadof , 39
ROOT, 19

## S
scrartcl, 18
syntax
  inner, **12**
  outer, **12**

## T
TeXLive, **15**
theory
  file, **12**
*tyargs*, 35
*type*, 35

*type_synonym_specification*, 37
*type_spec*, 35

## W
where clause, 34