Isabelle/DOF

User and Implementation Manual

Achim D. Brucker Burkhart Wolff

July 8, 2022

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

Laboratoire des Methodes Formelles (LMF)
Université Paris-Saclay
91405 Orsay Cedex
France

This manual describes Isabelle/DOF version 1.3.0/2021-1. The latest official release is 1.3.0/Isabelle2021-1 (doi:10.5281/zenodo.6810799). The DOI 10.5281/zenodo.3370482 will allways point to the latest release. 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 Nicolas Méric.

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

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.

It is an unique feature of Isabelle/DOF that ontologies may be used to control the link between formal and informal content in documents in a machine checked way. These links can connect both text elements and formal modeling elements such as terms, definitions, code and logical formulas, altogether *integrated* in a state-of-the-art interactive theorem prover.

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 [20]), 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 address 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 an instance of this 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 (called PIDE, [21]), 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 [25].

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):

  Document

  The axiom refl

- a red background 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`.

- for an application of Isabelle/DOF in the context of certifications:

  > A. D. Brucker and B. Wolff. Using Ontologies in Formal Developments Targeting Certification. In W. Ahrendt and S. Tarifa, editors. *Integrated Formal Methods (IFM)*, number 11918. Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, 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 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.*" [25]

The current system framework offers moreover the following features:

- a build management grouping components into to pre-compiled sessions,
- a prover IDE (PIDE) framework [21] 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 λ-calculus and some Higher-order Logic (HOL).

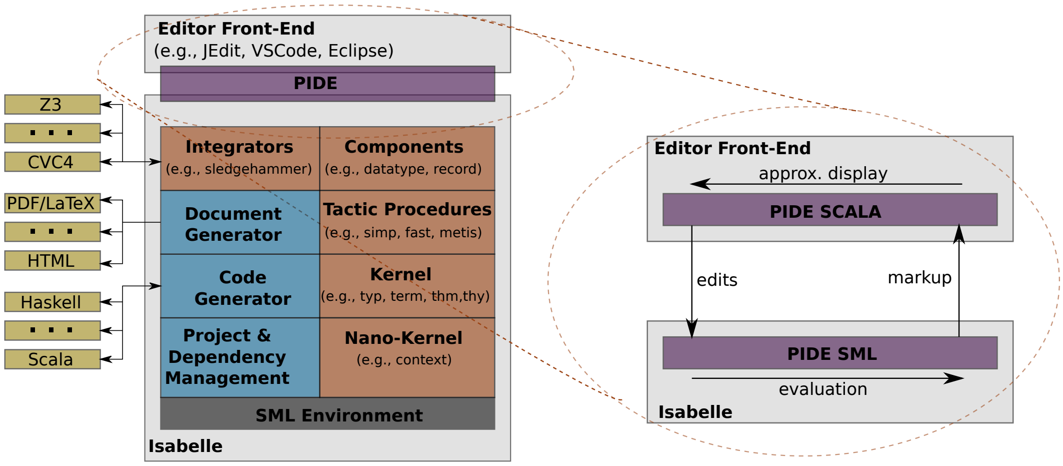


Figure 2.1: The system architecture of Isabelle (left-hand side) and the asynchronous communication between the Isabelle system and the IDE (right-hand side).

The Isabelle Framework is based on a *document-centric view* of a document, treating the input in its integrality as set of (user-programmable) *document element* that may mutually depend on and link to each other; A *document* in our sense is what is configured in a set of `ROOT`- and `ROOTS`-files.

Isabelle assumes a hierarchical document model, i. e., an *integrated* document consist of a hierarchy of *sub-documents* (files); dependencies are restricted to be acyclic at this level. 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 document elements called *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.

A text-element may look like this:

Isar

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

... so it is a command *text* followed by an argument (here in ‹ ... › parenthesis) which contains characters and a special notation for semantic macros (here @{*term* *fac 5*}).

While we concentrate in this manual on *text*-document elements — this is the main use of DOF in its current stage — it is important to note that there are actually three families of "ontology aware" document elements with analogous syntax to standard ones. The difference is a bracket with meta-data of the form:

```
text*[label::classid, attr₁=E₁, ... attrₙ=Eₙ]‹ some semi−formal text ›
ML*[label::classid, attr₁=E₁, ... attrₙ=Eₙ]‹ some SML code ›
value*[label::classid, attr₁=E₁, ... attrₙ=Eₙ]‹ some annotated λ−term ›
```

Depending on the family, we will speak about *(formal) text-contexts*, *(ML) code-contexts* and *term-contexts* if we refer to sub-elements inside the ‹...› cartouches of these command families. Note that the Isabelle framework allows for nesting cartouches that permits to support switching into a different context. In general, this has also the effect that the evaluation of antiquotations changes. [1]

On the semantic level, we assume a validation process for an integrated document, where the semantics of a command is a transformation $\vartheta \rightarrow \vartheta$ for some system state ϑ. This document model can be instantiated depending on the text-code-, or term-contexts. For common text elements, e. g., *section*‹...› or *text*‹...›, users can add informal text to a sub-document using a text command:

Isar

```
text‹This is a description.›
```

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:

Isar

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

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

Document

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

Semantic macros are partial functions of type $\vartheta \rightarrow$ *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).

[1] In the literature, this concept has been referred to *Cascade−Syntax* and was used in the Centaur-system and is existing in some limited form in some Emacs-implementations these days.

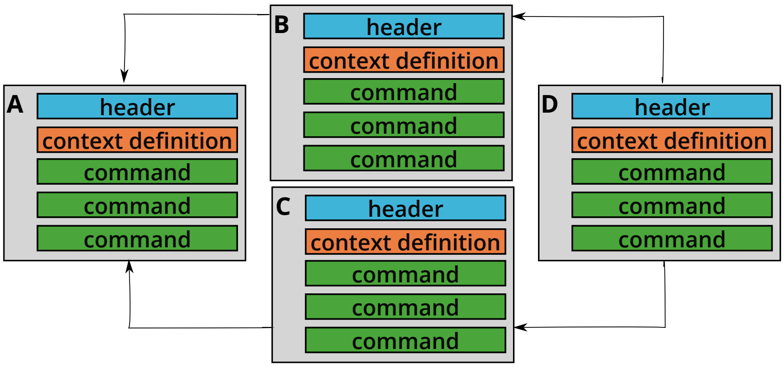


Figure 2.2: A Theory-Graph in the Document Model.

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 Document Model in other ITP's

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 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, 21, 22] 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. Besides UTF8-support for characters used in text-elements, Isabelle offers built-in already a mechanism for user-programmable antiquotations which we use to implement semantic macros in Isabelle/DOF (We will actually use these two terms as synonym in the context of Isabelle/DOF). Moreover, Isabelle/DOF allows for the asynchronous evaluation and checking of the document content [1, 21, 22] 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

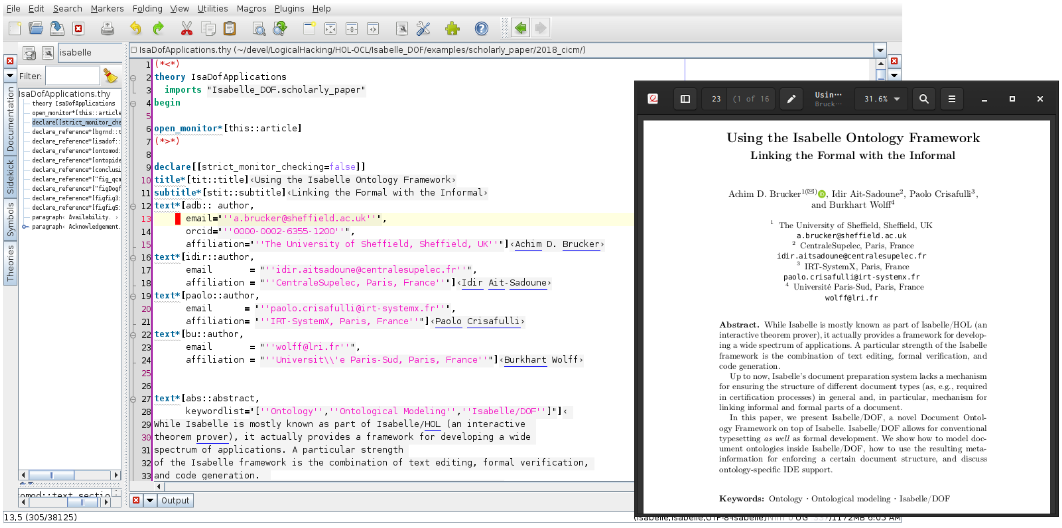


Figure 2.3: The Isabelle/DOF IDE (left) and the corresponding PDF (right), showing the first page of [5].

into sessions, i. e., sub-graphs of the document-structure that can be "pre-compiled" and loaded instantaneously, i. e., without re-processing, which is an important means to scale up.

3 Isabelle/DOF: A Guided Tour

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

3.1 Getting Started

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).

Isabelle/DOF requires Isabelle (2021-1) with a recent LaTeX-distribution (e.g., Tex Live 2022 or later). Isabelle/DOF uses a two-part version system (e.g., 1.2.0/Isabelle2021), 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.

Installing Isabelle. Please download and install Isabelle (version: 2021-1) from the Isabelle website (`https://isabelle.in.tum.de/website-Isabelle2021-1/2021-1`). After the successful installation of Isabelle, you should be able to call the `isabelle` tool on the command line:

```bash
achim@logicalhacking:~$ isabelle version
2021-1
```

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/Isabelle2021-1/bin/isabelle version
2021-1
```

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-full
```

Installing Isabelle/DOF

In the following, we assume that you already downloaded the Isabelle/DOF distribution (Isabelle_DOF-1.3.0_2021-1.tar.xz) from the Isabelle/DOF web site. We start by extracting the Isabelle/DOF archive:

```bash
achim@logicalhacking:~$ tar xf Isabelle_DOF-1.3.0_2021-1.tar.xz
```

This will create a directory `Isabelle_DOF-1.3.0_2021-1` containing Isabelle/DOF distribution.

Next, we need to register Isabelle/DOF as an Isabelle component:

```bash
achim@logicalhacking:~$ isabelle components -u Isabelle_DOF-1.3.0_2021-1
```

Moreover, Isabelle/DOF depends on the the AFP (`https://www.isa-afp.org`), namely the AFP entries “Functional Automata” [16] and “Regular Sets and Expressions” [14]. You can either install the complete AFP, following the instructions given at `https://www.isa-afp.org/using.html`), or use the provided `install` script to install a minimal AFP setup into the local Isabelle/DOF directory. The script needs to be prefixed with the `isabelle eëënv` command:

```bash
achim@logicalhacking:~$ cd Isabelle_DOF-1.3.0_2021-1
achim@logicalhacking:~/Isabelle_DOF-1.3.0_2021-1$ isabelle env  \
    ./install-afp

Isabelle/DOF Installer
======================
* Checking Isabelle version:
  Success: found supported Isabelle version (2021-1)
* Checking availability of AFP entries:
```

```bash
     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/Isabelle2021-1/ROOTS
          Registering Functional-Automata in
                /home/achim/.isabelle/Isabelle2021-1/ROOTS
          AFP installation successful.
* Installation successful. Enjoy Isabelle/DOF, you can build the session
  Isabelle/DOF and all example documents by executing:
  /usr/local/Isabelle2021-1/bin/isabelle build -D .
```

After the successful installation, you can 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-1.3.0_2021-1$ isabelle build -D .
```

3.1.2 Creating an Isabelle/DOF Project

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

```bash
achim@logicalhacking:~$ isabelle dof_mkroot myproject

Preparing session "myproject" in "myproject"
  creating "myproject/ROOT"
  creating "myproject/myproject.thy"
  creating "myproject/document/preamble.tex"

Now use the following command line to build the session:

  isabelle build -D myproject
```

The created project uses the default configuration (the ontology for writing academic papers (scholarly_paper) using a report layout based on the article class (`scrartcl`) of the KOMA-Script bundle [12]). The directory `myproject` contains 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:~/myproject$ isabelle build -d . myproject
```

The directory `myproject` contains the following files and directories:

```
└─myproject
  ├─document
  │ └─preamble.tex ..................... Manual LaTeX-configuration
  ├─myproject.thy ........................... Example theory file
  └─ROOT ........................... Isabelle build-configuration
```

The main two configuration files[1] 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 [24]. In addition to the standard features of, this file also contains Isabelle/DOF specific configurations:

[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.

  - `dof_ontologies` a list of (fully qualified) ontologies, separated by spaces, used by the project.
  - `dof_template` the (fully qualified) document template.
  - `document_build=dof` needs to be present, to tell Isabelle, to use the Isabelle/DOF backend for the document generation.

- The file `preamble.tex`, which allows users to add additional LaTeX-packages or to add/modify LaTeX-commands.

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 require that the users manually obtains and adds the necessary LaTeX class file. This is due to licensing restrictions).

This can be configured by using the command-line options of `dof_mkroot`. In Particular, `-o` allows selecting the ontology and `-t` allows selecting the document template. The built-in help (using `-h`) shows all available options:

Bash

```
achim@logicalhacking:~$ isabelle dof_mkroot -h

Usage: isabelle dof_mkroot [OPTIONS] [DIRECTORY]

  Options are:
    -I          init Mercurial repository and add generated files
    -n NAME     alternative session name (default: directory base name)
    -o ONTOLOGY ontology (default: scholarly_paper)
    -t TEMPLATE tempalte (default: scrartcl)

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

3.2 Writing Documents: General Remarks

3.2.1 Name-Spaces, Long- and Short-Names

Isabelle/DOF is built upon the name space and lexical conventions of Isabelle. Long-names were composed of a name of the session, the name of the theory, and a sequence of local names referring to, e. g., nested specification constructs that were used to identify types, constant symbols, definitions, etc. The general format of a long-name is
session_name.theory_name.local_name. ... .local_name
By lexical conventions, theory-names must be unique inside a session (and session names must be unique too), such that long-names are unique by construction. There are actually

different name categories that form a proper name space, e. g., the name space for constant symbols and type symbols are distinguished.

Isabelle identifies names already with the shortest suffix that is unique in the global context and in the appropriate name category. This also holds for pretty-printing, which can at times be confusing since names stemming from the same specification construct may be printed with different prefixes according to their uniqueness.

3.2.2 Caveat: Lexical Conventions of Cartouches, Strings, Names, ...

WARNING: The embedding of strings, terms, names etc, as parts of commands, anti-quotations, terms, etc, is unfortunately not always so consistent as one might expect, when it comes to variants that should be lexically equivalent in principle. This can be a nuisance for users, but is again a consequence that we build on existing technology that has been developed over decades.

At times, this causes idiosyncrasies like the ones cited in the following incomplete list:

- text-antiquotations `text`‹**thm** "normally_behaved_def"› versus `text`‹@{**thm** "srac$_1$_def"}› (while `text`‹@{**thm** ‹srac$_1$_def ›}› fails)
- commands `thm` *fundamental_theorem_of_calculus* and `thm` "fundamental_theorem_of_calculus" or `lemma` "H" and `lemma` ‹*H*› and `lemma` *H*
- string expressions `term`‹"abc" @ "cd"› and equivalent `term`‹‹abc› @ ‹cd››; but `term`‹‹A ⟶ B›› not equivalent to `term`‹"A ⟶ B"› which fails.

3.3 Writing Academic Publications in *scholarly_paper*

3.3.1 Writing Academic Papers

The ontology `scholarly_paper` is an ontology modeling academic/scientific papers, with a slight bias towards texts in the domain of mathematics and engineering. We explain first the principles of its underlying ontology, and then we present two "real" examples from our own publication practice.

1. The iFM 2020 paper [19] is a typical mathematical text, heavy in definitions with complex mathematical notation and a lot of non-trivial cross-referencing between statements, definitions and proofs which are ontologically tracked. However, wrt. the possible linking between the underlying formal theory and this mathematical presentation, it follows a pragmatic path without any "deep" linking to types, terms and theorems, deliberately not exploiting Isabelle/DOF 's full potential with this regard.
2. In the CICM 2018 paper [5], we deliberately refrain from integrating references to formal content in order to demonstrate that Isabelle/DOF is not a framework from Isabelle users to Isabelle users only, but people just avoiding as much as possible LaTeX notation.

The Isabelle/DOF distribution contains both examples using the ontology `scholarly_paper` in the directory `examples/scholarly_paper/2018-cicm-isabelle_dof-applications/` or `examples/scholarly_paper/2020-iFM-CSP`.

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`,
- starting Isabelle/jEdit from the command line by, e. g., calling:

```bash
achim@logicalhacking:~/Isabelle_DOF-1.3.0_2021-1$
  isabelle jedit -d . examples/scholarly_paper/2020-iFM-CSP/paper.thy
```

You can build the PDF-document at the command line by calling:

```bash
achim@logicalhacking:~$ isabelle build -d . 2020-iFM-csp
```

3.3.2 A Bluffers Guide to the `scholarly_paper` Ontology

In this section we give a minimal overview of the ontology formalized in *Isabelle_DOF.scholarly_paper*.

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 =
  email       :: string <= ''''
  http_site   :: string <= ''''
  orcid       :: string <= ''''
  affiliation :: string

doc_class abstract =
  keywordlist        :: string list   <= []
  principal_theorems :: thm list
```

Note *short_title* and *abbrev* are optional and have the default *None* (no value). Note further, that *abstract*s may have a *principal_theorems* list, where the built-in Isabelle/DOF type *thm list* contains references to formally proven theorems that must exist in the logical context of this document; this is a decisive feature of Isabelle/DOF that conventional ontological languages lack.

We continue by the introduction of a main class: the text-element *text_section* (in contrast to *figure* or *table* or similar). Note that the *main_author* is typed with the class *author*, a HOL type that is automatically derived from the document class definition *author* shown above. It is used to express which author currently "owns" this *text_section*, an information that can give rise to presentational or even access-control features in a suitably adapted front-end.

```
Isar
doc_class text_section = text_element +
  main_author :: author option  <=  None
  fixme_list  :: string list    <=  []
  level       :: int  option    <=  None
```

The *text_element*.*level*-attibute enables doc-notation support for headers, chapters, sections, and subsections; we follow here the LaTeX terminology on levels to which Isabelle/DOF is currently targeting at. The values are interpreted accordingly to the LaTeX standard. The correspondence between the levels and the structural entities is summarized as follows:

- part *Some* -1
- chapter *Some 0*
- section *Some 1*
- subsection *Some 2*
- subsubsection *Some 3*

Additional means assure that the following invariant is maintained in a document conforming to `scholarly_paper`:

$$level > 0$$

The rest of the ontology introduces concepts for *introduction*, *conclusion*, *related_work*, *bibliography* etc. More details can be found in `scholarly_paper` contained in the theory *Isabelle_DOF*.*scholarly_paper*.

3.3.3 Writing Academic Publications: A Freeform Mathematics Text

We present a typical mathematical paper focusing on its form, not referring in any sense to its content which is out of scope here. As mentioned before, we chose the paper [19] for this purpose, which is written in the so-called free-form style: Formulas are superficially parsed and type-set, but no deeper type-checking and checking with the underlying logical context is undertaken.

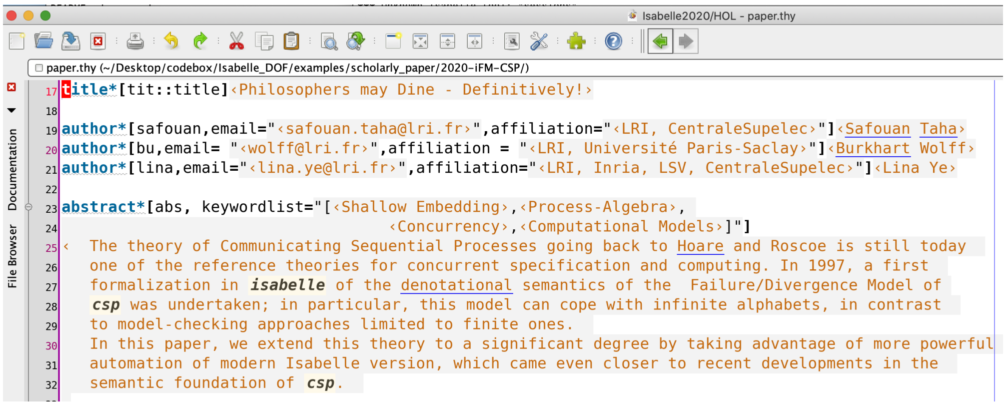
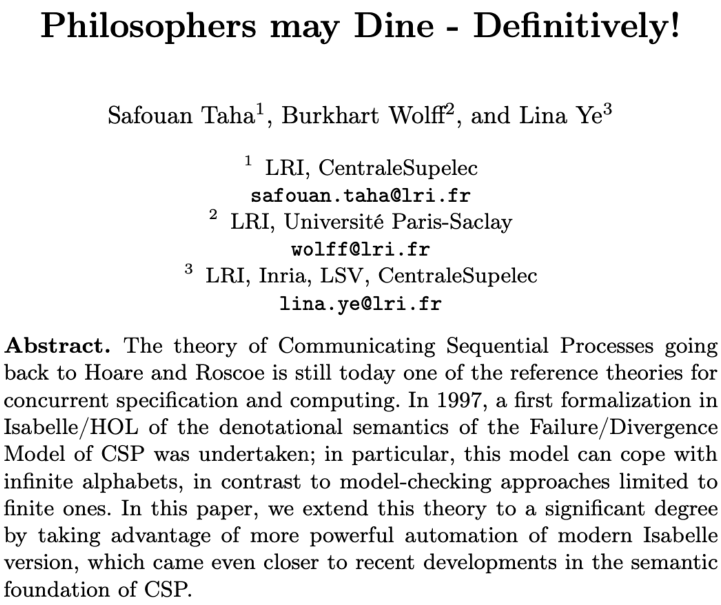


Figure 3.1: A mathematics paper as integrated document source ...



Philosophers may Dine - Definitively!

Safouan Taha[1], Burkhart Wolff[2], and Lina Ye[3]

[1] LRI, CentraleSupelec
`safouan.taha@lri.fr`
[2] LRI, Université Paris-Saclay
`wolff@lri.fr`
[3] LRI, Inria, LSV, CentraleSupelec
`lina.ye@lri.fr`

Abstract. The theory of Communicating Sequential Processes going back to Hoare and Roscoe is still today one of the reference theories for concurrent specification and computing. In 1997, a first formalization in Isabelle/HOL of the denotational semantics of the Failure/Divergence Model of CSP was undertaken; in particular, this model can cope with infinite alphabets, in contrast to model-checking approaches limited to finite ones. In this paper, we extend this theory to a significant degree by taking advantage of more powerful automation of modern Isabelle version, which came even closer to recent developments in the semantic foundation of CSP.

Figure 3.2: ... and as corresponding PDF-output.

The integrated source of this paper-excerpt is shown in Figure 3.1, while the document build process converts this to the corresponding PDF-output shown in Figure 3.2.

Recall that the standard syntax for a text-element in Isabelle/DOF is *text**[<*id*>::<*class_id*>,<*attrs*>]‹ ... *text* ...›, but there are a few built-in abbreviations like *title**[<*id*>,<*attrs*>]‹ ... *text* ...› that provide special command-level syntax for text-elements. The other text-elements provide the authors and the abstract as specified by their *class_id* referring to the *doc_class*es of `scholarly_paper`; we say that these text elements are *instances* of the *doc_class*es of the underlying ontology.

The paper proceeds by providing instances for introduction, technical sections, examples, etc. We would like to concentrate on one — mathematical paper oriented — detail in the ontology `scholarly_paper`:

Isar

```
doc_class technical = text_section + ...

type_synonym tc = technical

datatype math_content_class = defn | axm | thm  | lem | cor | prop | ...

doc_class math_content = tc + ...

doc_class definition  = math_content +
  mcc         :: math_content_class <= defn ...

doc_class theorem     = math_content +
  mcc         :: math_content_class <= thm  ...
```

The class *technical* regroups a number of text-elements that contain typical technical content in mathematical or engineering papers: code, definitions, theorems, lemmas, examples. From this class, the stricter class of *math_content* is derived, which is grouped into *definition*s and *theorem*s (the details of these class definitions are omitted here). Note, however, that class identifiers can be abbreviated by standard *type_synonym*s for convenience and enumeration types can be defined by the standard inductive *datatype* definition mechanism in Isabelle, since any HOL type is admitted for attribute declarations. Vice-versa, document class definitions imply a corresponding HOL type definition.

An example for a sequence of (Isabelle-formula-)texts, their ontological declarations as *definition*s in terms of the `scholarly_paper`-ontology and their type-conform referencing later is shown in Figure 3.3 in its presentation as the integrated source.

Note that the use in the ontology-generated antiquotation @{*definition* *X4*} is type-checked; referencing `X4` as *theorem* would be a type-error and be reported directly by Isabelle/DOF in Isabelle/jEdit. Note further, that if referenced correctly wrt. the sub-typing hierarchy makes `X4` *navigable* in Isabelle/jEdit; a click will cause the IDE to present the defining occurrence of this text-element in the integrated source.

Note, further, how Isabelle/DOF-commands like *text** interact with standard Isabelle document antiquotations described in the Isabelle Isar Reference Manual in Chapter 4.2 in paper.thy (~/Desktop/codebox/Isabelle_DOF/examples/scholarly_paper/2020-iFM-CSP/)

```
Definition*[X22]‹‹RUN A ≡ μ X. □ x ∈ A → X›                    vs‹-0.7cm››
Definition*[X32]‹‹CHAOS A ≡ μ X. (STOP ⊓ (□ x ∈ A → X))›        vs‹-0.7cm››
Definition*[X42]‹‹CHAOS_SKIP A ≡ μ X. (SKIP ⊓ STOP ⊓ (□ x ∈ A → X))›vs‹-0.7cm››

text‹ The ‹RUN›-process defined @{definition X22} represents the process that accepts all
events, but never stops nor deadlocks. The ‹CHAOS›-process comes in two variants shown in
@{definition X32} and @{definition X42}: the process that non-deterministically stops or
accepts any offered event, whereas ‹CHAOS_SKIP› can additionally terminate.›
```

Figure 3.3: A screenshot of the integrated source with definitions ...

Definition 2. $RUN\ A \equiv \mu\ X.\ \Box\ x \in A \rightarrow X$
Definition 3. $CHAOS\ A \equiv \mu\ X.\ (STOP \sqcap (\Box\ x \in A \rightarrow X))$
Definition 4. $CHAOS_{SKIP}\ A \equiv \mu\ X.\ (SKIP \sqcap STOP \sqcap (\Box\ x \in A \rightarrow X))$

The RUN-process defined 2 represents the process that accepts all events, but never stops nor deadlocks. The $CHAOS$-process comes in two variants shown in 3 and 4: the process that non-deterministically stops or accepts any offered event, wheras $CHAOS_{SKIP}$ can additionaly terminate.

Figure 3.4: ... and the corresponding PDF-output.

great detail. We refrain ourselves here to briefly describe three freeform antiquotations used in this text:

- the freeform term antiquotation, also called *cartouche*, written by @{*cartouche* [*style−parms*] ‹...›} or just by ‹...› if the list of style parameters is empty,
- the freeform antiquotation for theory fragments written @{*theory_text* [*style−parms*] ‹...›} or just `\<^theory_text>`‹...› if the list of style parameters is empty,
- the freeform antiquotations for verbatim, emphasized, bold, or footnote text elements.

Isabelle/DOF text-elements such as `text*` allow to have such standard term-antiquotations inside their text, permitting to give the whole text entity a formal, referentiable status with typed meta-information attached to it that may be used for presentation issues, search, or other technical purposes. The corresponding output of this snippet in the integrated source is shown in Figure 3.4.

3.3.4 More Freeform Elements, and Resulting Navigation

In the following, we present some other text-elements provided by the Common Ontology Library in *Isabelle_DOF.Isa_COL*. It provides a document class for figures:

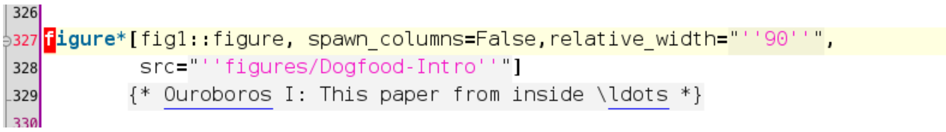
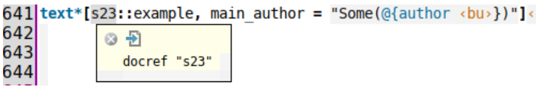
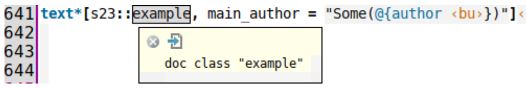


Figure 3.5: Declaring figures in the integrated source.



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



(b) Exploring the class of a text element.

Figure 3.6: Exploring text elements.

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

The document class *figure* (supported by the Isabelle/DOF command abbreviation *figure∗*) makes it possible to express the pictures and diagrams as shown in Figure 3.5, which presents its own representation in the integrated source as screenshot.

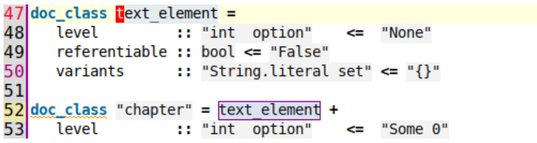
Finally, we define a *monitor class* that enforces a textual ordering in the document core by a regular expression:

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

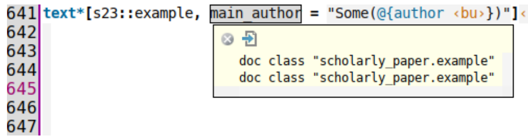
In a integrated document source, the body of the content can be paranthesized into:

Isar
```
open_monitor* [this::article]
...
close_monitor*[this]
```

which signals to Isabelle/DOF begin and end of the part of the integrated source in which the text-elements instances are expected to appear in the textual ordering defined by *article*.



(a) Hyperlink to class-definition.



(b) Exploring an attribute (hyperlinked to the class).

Figure 3.7: Navigation via generated hyperlinks.

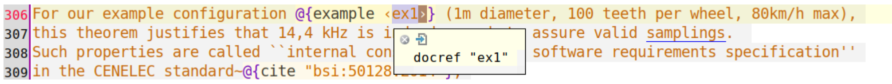


Figure 3.8: Exploring an ontological reference.

From these class definitions, Isabelle/DOF also automatically generated editing support for Isabelle/jEdit. In Figure 3.6a and Figure 3.6b 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 3.7a); hovering over an attribute-definition (which is qualified in order to disambiguate; Figure 3.7b) shows its type.

An ontological reference application in Figure 3.8: the ontology-dependant antiquotation @{*example* ‹*ex1*›} refers to the corresponding text-element *ex1*. Hovering allows for inspection, clicking for jumping to the definition. If the link does not exist or has a non-compatible type, the text is not validated, i. e., Isabelle/jEdit will respond with an error.

We advise users to experiment with different notation variants. Note, further, that the Isabelle @{cite ...}-text-anti-quotation makes its checking on the level of generated `.aux`-files, which are not necessarily up-to-date. Ignoring the PIDE error-message and compiling a with a consistent bibtex usually makes disappear this behavior.

3.4 Writing Certification Documents *CENELEC_50128*

3.4.1 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 integrated source example 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-1.3.0_2021-1$
  isabelle jedit examples/CENELEC_50128/mini_odo/mini_odo.thy
```

Finally, you

- can build the PDF-document by calling:

  Bash
  ```
  achim@logicalhacking:~/Isabelle_DOF-1.3.0_2021-1$ isabelle build  \
      mini_odo
  ```

3.4.2 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 substantial amount of the work of evaluators in formal certification processes consists in tracing down the links from requirements over assumptions down to elements of evidence, be it in form of semi-formal documentation, models, code, or tests. In a certification process, traceability becomes a major concern; and providing mechanisms to ensure complete traceability already at the development of the integrated source can in our view increase the speed and reduce the risk certification processes. 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), has the potential in our view to decrease the cost of software developments targeting certifications.

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 traceability 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 hypothesis = requirement +
      hyp_type :: hyp_type <= physical

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 of 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 *ass_kind.formal*, *ass_kind.semiformal* and *informal* sub-categories. They have to be tracked and discharged by appropriate validation procedures within a certification process, be 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 assumptions, 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, be 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 traceability throughout the certification is therefore particularly critical. This is naturally modeled as follows:

```
Isar
doc_class EC = assumption +
    assumption_kind :: ass_kind <= formal

doc_class SRAC = EC +
    assumption_kind :: ass_kind <= 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 frequency of the odometer subsystem is therefore 14 khz, which includes sampling, computing and result communication times ...*

Note that this PDF-output is the result of a specific setup for *SRAC*s.

3.4.3 Editing Support for CENELEC 50128

The corresponding view in Figure 3.9 shows core part of a document conforming to the `CENELEC_50128` ontology. The first sample shows standard Isabelle antiquotations [23] into

```
text‹
The resolution of time, distance, speed and acceleration data, in International System Unit (SI),
shall be:
▪ term‹Time›: ‹10⁻² s› the resolution needed for calculation.
▪ term‹Distance› ‹10⁻³ m› ( ie 1mm)
▪ term‹Speed›: ‹1.3 * 10⁻³ m/s› ( ie ‹0.005› km/h)
▪ term‹Acceleration›: ‹0.005 m/s²›
▪ term‹Jerk›

The precisio                                     alculations and propagated to the external
interface da
        constant "Odo_ReqAna.Acceleration"
        :: (real ⇒ real) ⇒ real ⇒ real
```

Figure 3.9: Standard antiquotations referring to theory elements.

```
text*[enough_samples::SRAC]‹ Note that the theorem above establishes a constraint between
 const‹wheel_circumference›,  const‹teeth_per_wheelturn›,   const‹Speed_Max› and
 const‹sampling_frequency›; since this exported constraint is fundamental for the safe
functioning of  odometer and therefore a safety-related exported application constraint.
It is formally expressed as follows:›
```

Figure 3.10: Defining a *SRAC* in the integrated source ...

formal entities of a theory. This way, the informal parts of a document get "formal content" and become more robust under change.

The subsequent sample in Figure 3.10 shows the definition of a *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 3.11 this safety-related condition; however, this happens in a context where general *exported constraints* are listed. Isabelle/DOF's checks and establishes that this is legal in the given ontology.

3.5 Writing Technical Reports in *technical_report*

While it is perfectly possible to write documents in the `technical_report` ontology in freeform-style (the present manual is mostly an example for this category), we will briefly explain here the tight-checking-style in which most Isabelle reference manuals themselves are written.

The idea has already been put forward by Isabelle itself; besides the general infrastructure on which this work is also based, current Isabelle versions provide around 20 built-in document

```
text‹ Summing up, the property that the odometer provides sufficient sampling
precision --- meaning no wheel encodings were ``lost'' compared to any sampling done with
a higher sampling rate --- can be established under the set of general hypothesis captured
in assumption‹general_hyps› (formally expressed in thm"normally_behaved_distance_function_def")
and the SRAC EC‹enough_samples› formally expressed by  @{thm "srac₁_def"}. ›

subsection‹ Fault
text‹
        docref "enough_samples"
```

Figure 3.11: Using a *SRAC* as *EC* document element.

and code antiquotations described in the Reference Manual pp.75 ff. in great detail.

Most of them provide strict-checking, i. e. the argument strings were parsed and machine-checked in the underlying logical context, which turns the arguments into *formal content* in the integrated source, in contrast to the free-form antiquotations which basically influence the presentation.

We still mention a few of these document antiquotations here:

- @{thm ‹refl›} or @{thm [display] ‹refl›} check that *refl* is indeed a reference to a theorem; the additional "style" argument changes the presentation by printing the formula into the output instead of the reference itself,
- @{lemma ‹prop› } by ‹method› allows deriving *prop* on the fly, thus guarantee that it is a corrollary of the current context,
- @{term ‹term› } parses and type-checks *term*,
- @{value ‹term› } performs the evaluation of *term*,
- @{ML ‹ml−term› } parses and type-checks *ml−term*,
- @{ML_file ‹ml−file› } parses the path for *ml−file* and verifies its existance in the (Isabelle-virtual) file-system.

There are options to display sub-parts of formulas etc., but it is a consequence of tight-checking that the information must be given complete and exactly in the syntax of Isabelle. This may be over-precise and a burden to readers not familiar with Isabelle, which may motivate authors to choose the aforementioned freeform-style.

3.5.1 A Technical Report with Tight Checking

An example of tight checking is a small programming manual developed by the second author in order to document programming trick discoveries while implementing in Isabelle. While not necessarily a meeting standards of a scientific text, it appears to us that this information is often missing in the Isabelle community.

So, if this text addresses only a very limited audience and will never be famous for its style, it is nevertheless important to be *exact* in the sense that code-snippets and interface descriptions should be accurate with the most recent version of Isabelle in which this document is generated. So its value is that readers can just reuse some of these snippets and adapt them to their purposes.

TR_MyCommentedIsabelle is written according to the `technical_report` ontology in *Isabelle_DOF.technical_report*. Figure 3.12 shows a snippet from this integrated source and gives an idea why its tight-checking allows for keeping track of underlying Isabelle changes: Any reference to an SML operation in some library module is type-checked, and the displayed SML-type really corresponds to the type of the operations in the underlying SML environment. In the PDF output, these text-fragments were displayed verbatim.

```
text*[squiggles::technical]
‹← Finally, a number of commonly used "squigglish" combinators is listed:

■ @{ML "op !  : 'a Unsynchronized.ref->'a"},  access operation on a program variable  vs‹-0.3cm›
■ @{ML "op := : ('a Unsynchronized.ref * 'a)->unit"}, update operation on program variable vs‹-0.3cm›
■ @{ML "op #> : ('a->'b) * ('b->'c)->'a->'c"}, a reversed function composition  vs‹-0.3cm›
■ @{ML "I: 'a -> 'a"}, the I combinator vs‹-0.3cm›
■ @{ML "K: 'a -> 'b -> 'a"}, the K combinator vs‹-0.3cm›
■ @{ML "op o  : (('b->'c) * ('a->'b))->'a->'c"}, function composition vs‹-0.3cm›
■ @{ML "op || : ('a->'b) * ('a->'b) -> 'a -> 'b"}, parse alternative vs‹-0.3cm›
■ @{ML "op -- : ('a->'b*'c) * ('c->'d*'e)->'a->('b*'d)*'e"}, parse pair vs‹-0.3cm›
```

Figure 3.12: A table with a number of SML functions, together with their type.

3.6 Style Guide

The document generation 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, but 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 LaTeX has two advantages: first, LaTeX commands can circumvent the consistency checks of Isabelle/DOF and, hence, only if no LaTeX commands 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 too complex native LaTeX-commands.

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

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

4 Ontologies and their Development

In this chapter, we explain the concepts of Isabelle/DOF in a more systematic way, and give guidelines for modeling new ontologies, present underlying concepts for a mapping to a representation, and give hints for the development of new document templates.

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 provides a number of new command definitions in Isabelle's document model.

Isabelle/DOF consists consists basically of five components:

- the *core* in *Isabelle_DOF.Isa_DOF* providing the *ontology definition language* (ODL) which allow for the definitions of document-classes and necessary datatypes,
- the *core* also provides an own *family of commands* such as **text***, **declare_reference***, etc.; They allow for the annotation of text-elements with meta-information defined in ODL,
- the Isabelle/DOF library *Isabelle_DOF.Isa_COL* providing ontological basic (documents) concepts as well as supporting infrastructure,
- 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.

Similarly to Isabelle, which is based on a core logic *Pure* and then extended by libraries to major systems like HOL, Isabelle/DOF has a generic core infrastructure DOF and then presents itself to users via major library extensions, which add domain-specific system-extensions. Ontologies in Isabelle/DOF are not just a sequence of descriptions in Isabelle/DOF's Ontology Definition Language (ODL). Rather, they are themselves presented as integrated sources that provide textual descriptions, abbreviations, macro-support and even ML-code. Conceptually, the library of Isabelle/DOF is currently organized as follows[1]:

```
COL ...................................................... The Common Ontology Library
└─ scholarly_paper ......................................... Scientific Papers
   └─ technical_report ...................................... Extended Papers
      ├─ CENELEC_50128 ....................... Papers according to CENELEC_50128
      ├─ CC_v3_1_R5 ........................ Papers according to Common Criteria
      └─ ...
```

[1]The *technical* organization is slightly different and shown in Section 4.5.

These libraries not only provide ontological concepts, but also syntactic sugar in Isabelle's command language Isar that is of major importance for users (and may be felt as Isabelle/DOF key features by many authors). In reality, they are derived concepts from more generic ones; for example, the commands *title**, *section**, *subsection**, etc, are in reality a kind of macros for *text**[<*label*>::*title*]..., *text**[<*label*>::*section*]..., respectively. These example commands are defined in COL (the common ontology library).

As mentioned earlier, our ontology framework is currently particularly geared towards *document* editing, structuring and presentation (future applications might be advanced "knowledge-based" search procedures as well as tool interaction). For this reason, ontologies are coupled with *layout definitions* allowing an automatic mapping of an integrated source into LaTeX and finally PDF. The mapping of an ontology to a specific representation in LaTeX is steered via associated LaTeX style files which were included during Isabelle's document generation process. This mapping is potentially a one-to-many mapping; this implies a certain technical organization and some resulting restrictions described in Section 4.5 in more detail.

4.1 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, the keyword (*onto_class*) is accepted equally,
- 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 an *accepts* clause, or via a list in a *rejects* clause,
- attributes may have default values in order to facilitate notation.

The Isabelle/DOF ontology specification language consists basically of a notation for document classes, where the attributes were typed with HOL-types and can be instantiated by

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.

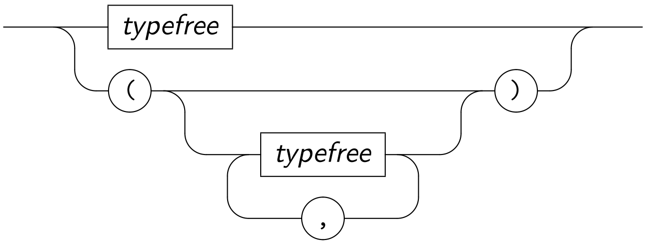
Document classes support **accepts**-clauses containing a regular expression over class names. Classes with an **accepts**-clause were called *monitor classes*. While document classes and their inheritance relation structure meta-data of text-elements in an object-oriented manner, monitor classes enforce 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 _ $\times$ _. 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-type in order to allow *links* to and between ontological concepts.

4.1.1 Some Isabelle/HOL Specification Constructs Revisited

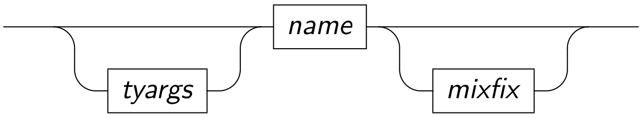
As ODL is an extension of Isabelle/HOL, document class definitions can therefore be arbitrarily mixed with standard HOL specification constructs. To make this manual self-contained, we present syntax and semantics of the specification constructs that are most likely relevant for the developer of ontologies (for more details, see [23]. 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_ident*'s in [23]) and identifiers in ... which might contain certain "quasi-letters" such as _, −, . (see [23] for details).
- *tyargs*:



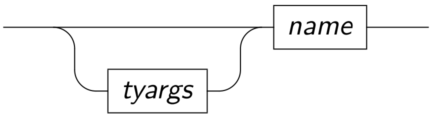
typefree denotes fixed type variable (*'a*, *'b*, ...) (see [23])

- *dt_name*:



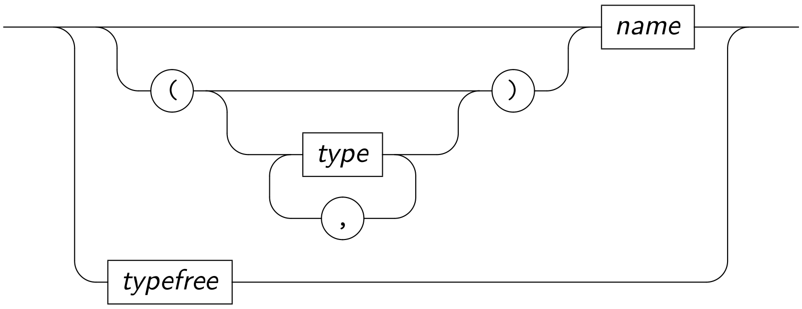
The syntactic entity *name* denotes an identifier, *mixfix* denotes the usual parenthesized mixfix notation (see [23]). The *name*'s referred here are type names such as *int*, *string*, *list*, *set*, etc.

- *type_spec*:

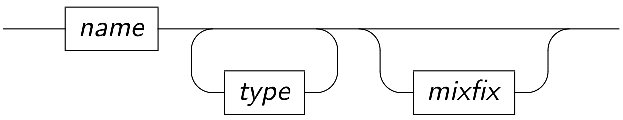


The *name*'s referred here are type names such as *int*, *string*, *list*, *set*, etc.

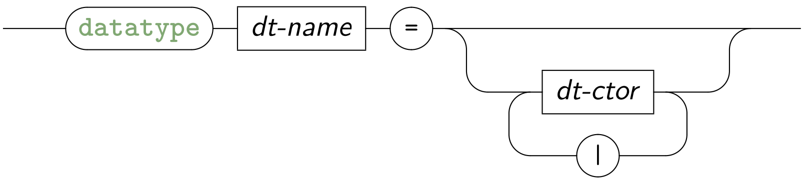
- *type*:



- *dt_ctor*:

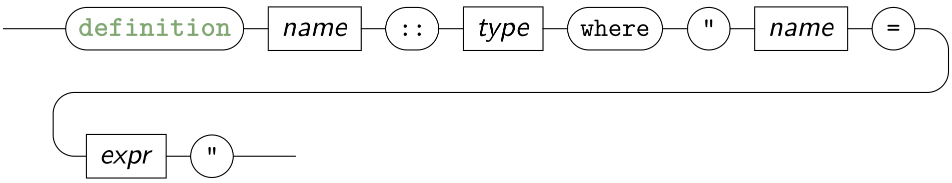


- *datatype_specification*:



- *type_synonym_specification*:

- *constant_definition* :



- *expr*: the syntactic category *expr* here denotes the very rich "inner-syntax" language of mathematical notations for λ-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), $\forall\ x.\ P(x) \wedge Q\ x = C$ (formulas). For details, see [17].

Advanced ontologies can, e. g., use recursive function definitions with pattern-matching [13], extensible record specifications [23], and abstract type declarations.

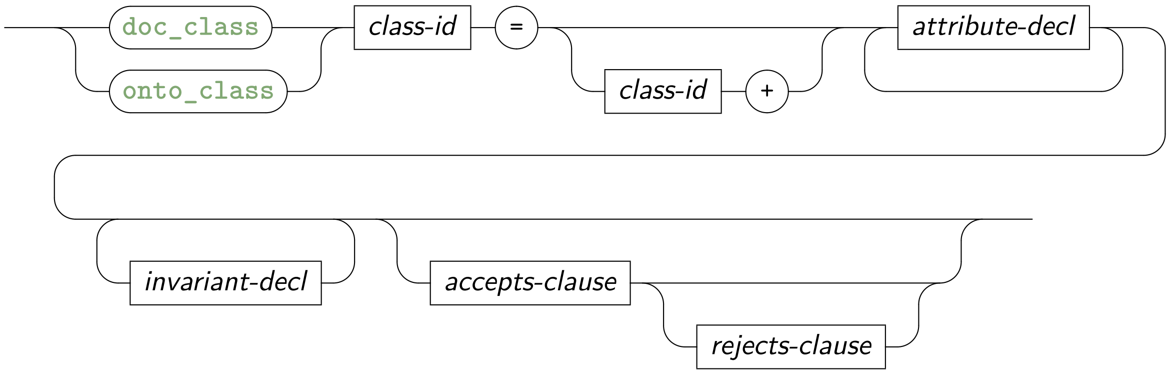
Isabelle/DOF works internally with fully qualified names in order to avoid confusions occurring otherwise, for example, in disjoint class hierarchies. This also extends to names for `doc_class`es, which must be representable as type-names as well since they can be used in attribute types. Since theory 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.

4.1.2 Defining Document Classes

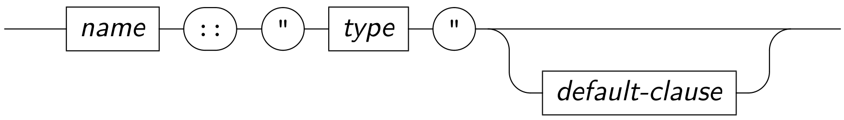
A document class can be defined using the `doc_class` keyword:

- *class_id*: a type-*name* that has been introduced via a *doc_class_specification*.

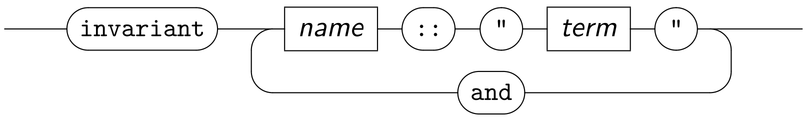
- *doc_class_specification*: We call document classes with an *accepts_clause monitor classes* or *monitors* for short.



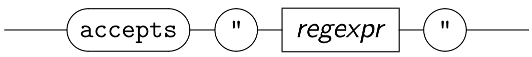
- *attribute_decl*:



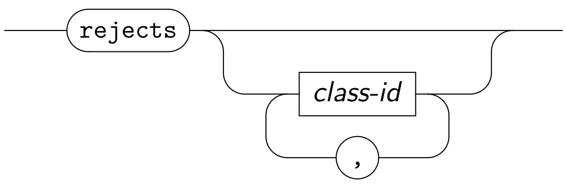
- *invariant_decl*: Invariants can be specified as predicates over document classes represented as records in HOL. Sufficient type information must be provided in order to disambiguate the argument of the expression and the σ symbol is reserved to reference the instance of the class itself.



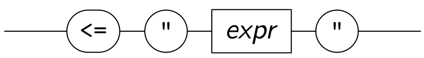
- *accepts_clause*:



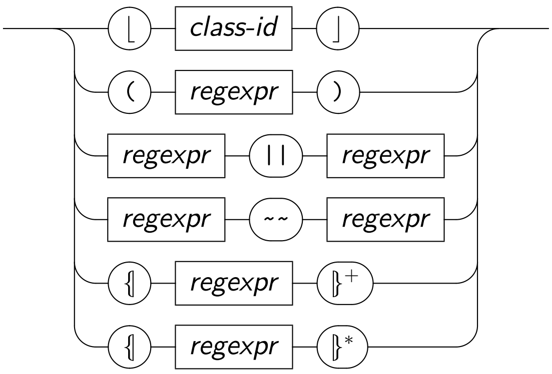
- *rejects_clause*:



- *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 the table of contents or an index. Overall, the `\newisadof[]{}` command follows the structure of the *`doc_class`*-command:

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

The *class_id* (or *cid* for short) 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 to the style files provided in the Isabelle/DOF distribution. In particular the definitions of the concepts *title∗* and *author∗* in the file `../../../src/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.

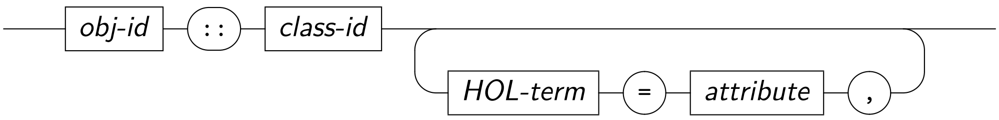
4.2 Fundamental Commands of the Isabelle/DOF Core

Besides the core-commands to define an ontology as presented in the previous section, the Isabelle/DOF core provides a number of mechanisms to *use* the resulting data to annotate text-elements and, in some cases, terms.

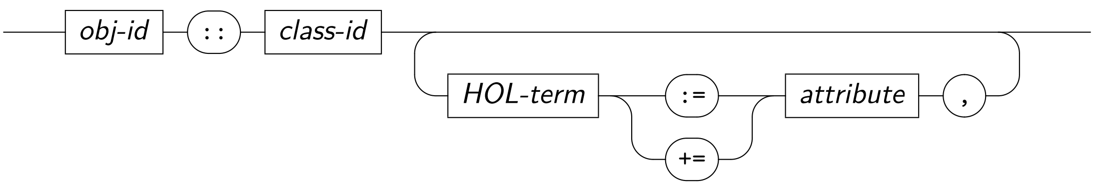
4.2.1 Syntax

In the following, we formally introduce the syntax of the core commands as supported on the Isabelle/Isar level. Some more advanced functionality of the core is currently only available in the SML API's of the kernel.

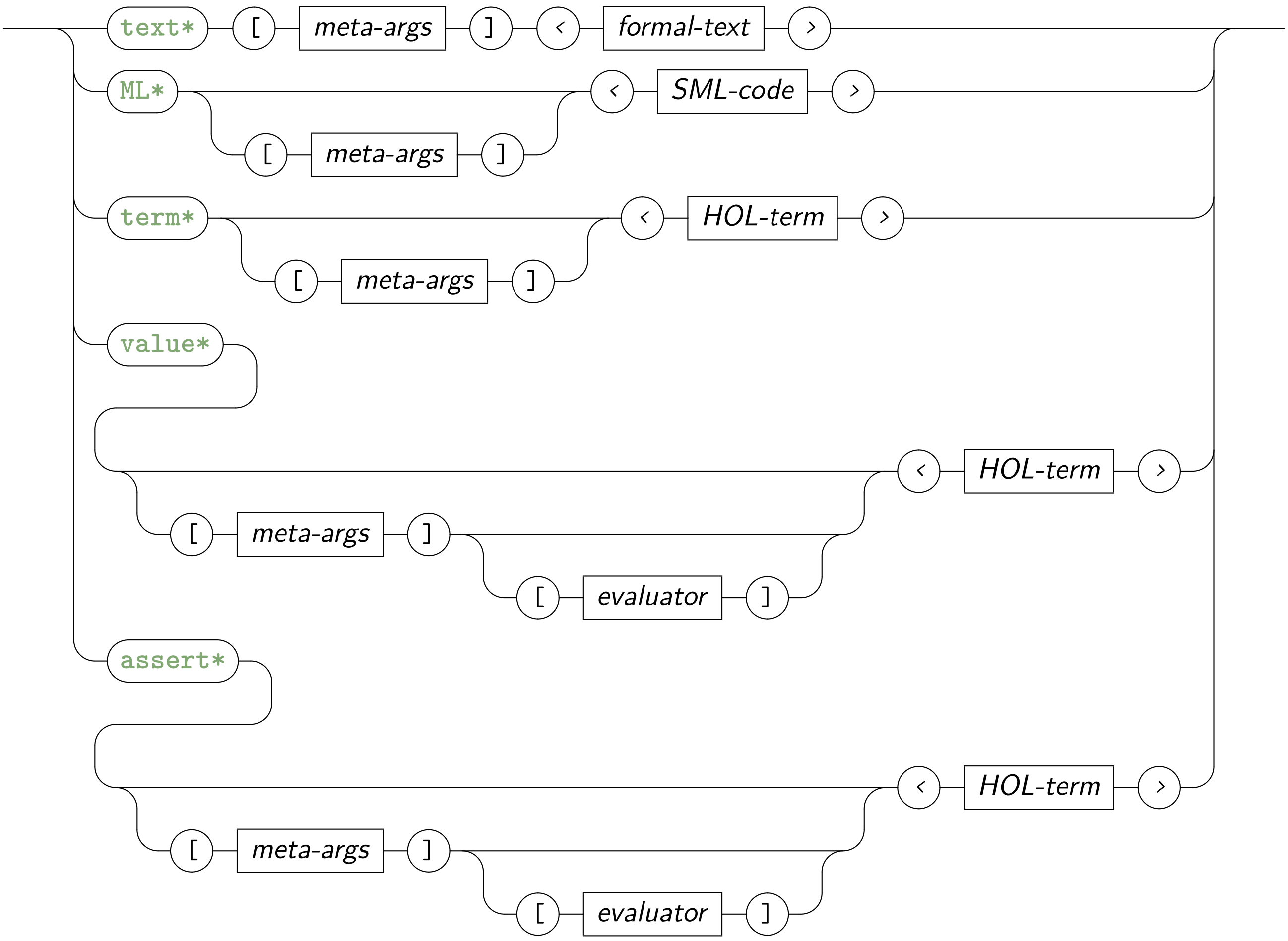
- *obj_id*: (or *oid* for short) a *name* as specified in Section 4.1.1.
- *meta_args* :

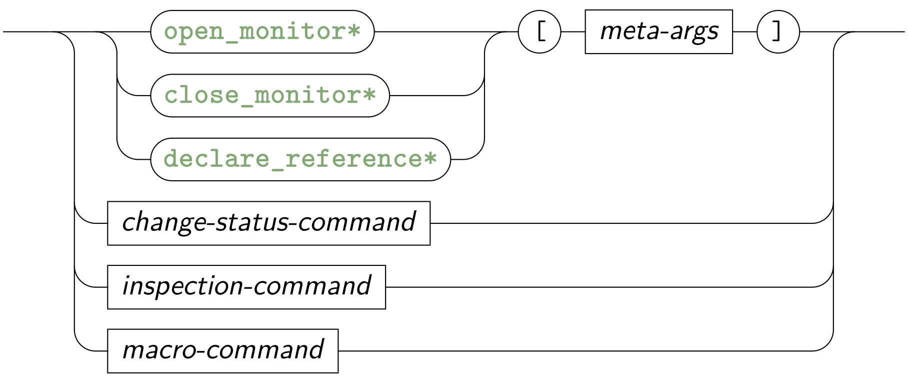


- *evaluator*: from [23], evaluation is tried first using ML, falling back to normalization by evaluation if this fails. Alternatively a specific evaluator can be selected using square brackets; typical evaluators use the current set of code equations to normalize and include *simp* for fully symbolic evaluation using the simplifier, *nbe* for *normalization by evaluation* and *code* for code generation in SML.
- *upd_meta_args* :

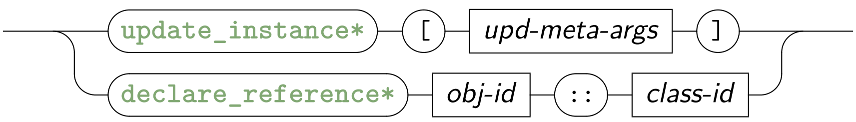


- *annotated_text_element* :

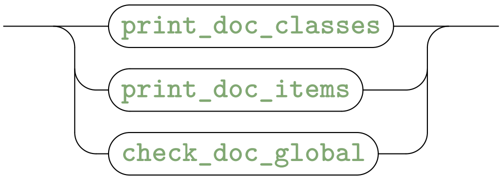




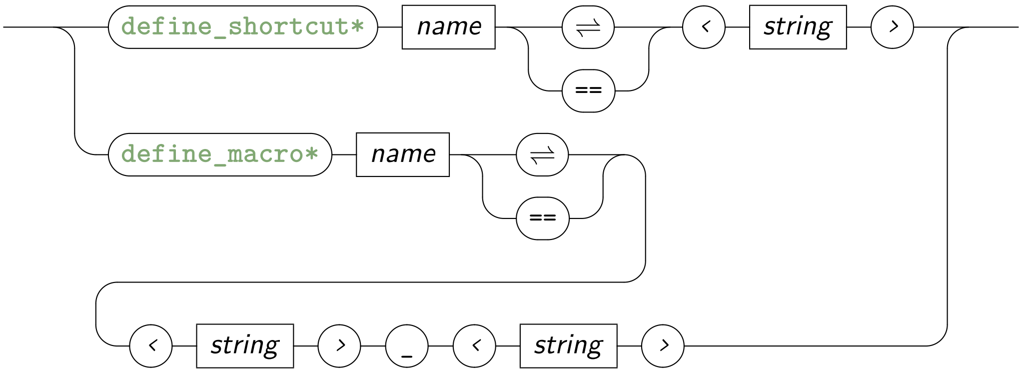
- Isabelle/DOF *change_status_command* :



- Isabelle/DOF *inspection_command* :



- Isabelle/DOF *macro_command* :



Recall that except `text*`[]‹...›, all Isabelle/DOF commands were mapped to visible layout; these commands have to be wrapped into `(*<*) ... (*>*)` if this is undesired.

4.2.2 Ontological Text-Contexts and their Management

With respect to the family of text elements, `text*`[*oid*::*cid*, ...] ‹ ... *text* ... › is the core-command of Isabelle/DOF: it permits to create an object of meta-data belonging to the class *cid*. This is viewed as the *definition* of an instance of a document class. The class invariants were checked for all attribute values at creation time if not specified otherwise. Unspecified attributed values were represented by fresh free variables. This instance object is attached to the text-element and makes it thus "trackable" for Isabelle/DOF, i. e., it can be referenced via the *oid*, its attributes can be set by defaults in the class-definitions, or set at creation time, or modified at any point after creation via `update_instance*`[*oid*, ...]. The *class_id* is syntactically optional; if ommitted, an object belongs to an anonymous superclass of all classes. The *class_id* is used to generate a *class-type* in HOL; note that this may impose lexical restrictions as well as to name-conflicts in the surrounding logical context. In many cases, it is possible to use the class-type to denote the *class_id*; this also holds for type-synonyms on class-types.

References to text-elements can occur textually before creation; in these cases, they must be declared via `declare_reference*`[...] in order to compromise to Isabelle's fundamental "declaration-before-use" linear-visibility evaluation principle. The forward-declared class-type must be identical with the defined class-type.

For a declared class *cid*, there exists a text antiquotation of the form @{*cid* ‹*oid*›}. The precise presentation is decided in the *layout definitions*, for example by suitable LaTeX-template code. Declared but not yet defined instances must be referenced with a particular pragma in order to enforce a relaxed checking @{*cid* (**unchecked**) ‹*oid*›}.

4.2.3 Ontological Code-Contexts and their Management

The `ML*`[*oid*::*cid*, ...] ‹ ... *SML–code* ... ›-document elements proceed similarly: a referentiable meta-object of class *cid* is created, initialized with the optional attributes and bound to *oid*. In fact, the entire the meta-argument list is optional. The SML-code is type-checked and executed in the context of the SML toplevel of the Isabelle system as in the corresponding `ML`‹ ... *SML–code* ... ›-command.

4.2.4 Ontological Term-Contexts and their Management

The major commands providing term-contexts are `term*`[*oid*::*cid*, ...] ‹ ... *HOL–term* ... ›, `value*`[*oid*::*cid*, ...] ‹ ... *HOL–term* ... › and `assert*`[*oid*::*cid*, ...] ‹ ... *HOL–term* ... ›[2]. Wrt. creation, track-ability and checking they are analogous to the ontological text and code-commands. However the argument terms may contain term-antiquotations stemming from an ontology definition. Both term-contexts were type-checked and *validated* against the global context (so: in the term @{*A* ‹*oid*›}, *oid* is indeed a string which refers to a meta-object belonging to the document class *A*, for example). The term-context in the *value*∗-command and *assert**-command is additionally expanded (e. g. replaced) by a term denoting the meta-object. This expansion happens *before* evaluation of the term, thus permitting executable HOL-functions to interact with meta-objects. The *assert**-command allows for logical statements to be checked in the global context (see Section 4.3.1). This is particularly useful to explore formal definitions wrt. their border cases.

Note unspecified attribute values were represented by free fresh variables which constrains DOF to choose either the normalization-by-evaluation strategy *nbe* or a proof attempt via the *auto* method. A failure of these strategies will be reported and regarded as non-validation of this meta-object. The latter leads to a failure of the entire command.

4.2.5 Status and Query Commands

Isabelle/DOF provides a number of inspection commands.

- `print_doc_classes` allows to view the status of the internal class-table resulting from ODL definitions,
- `DOF_core.print_doc_class_tree` allows for presenting (fragments) of class-inheritance trees (currently only available at ML level),
- `print_doc_items` allows to view the status of the internal object-table of text-elements that were tracked, and

[2] The meta-argument list is optional.

- *check_doc_global* checks if all declared object references have been defined, all monitors are in a final state, and checks the final invariant on all objects (cf. Section 4.4)

4.2.6 Macros

There is a mechanism to define document-local macros which were PIDE-supported but lead to an expansion in the integrated source; this feature can be used to define

- *shortcuts*, i. e., short names that were expanded to, for example, LaTeX-code,
- *macro*'s (= parameterized short-cuts), which allow for passing an argument to the expansion mechanism.

The argument can be checked by an own SML-function with respect to syntactic as well as semantic regards; however, the latter feature is currently only accessible at the SML level and not directly in the Isar language. We would like to stress, that this feature is basically an abstract interface to existing Isabelle functionality in the document generation.

Examples

- common short-cut hiding LaTeX code in the integrated source:

  ```
  define_shortcut* eg  ⇌ ‹\eg›
                   clearpage ⇌ ‹\clearpage{}›
  ```

- non-checking macro:

  ```
  define_macro* index  ⇌ ‹\index{› _ ‹}›
  ```

- checking macro:

  ```
  setup‹ DOF_lib.define_macro binding‹vs›  \\vspace{ } (check_latex_measure) ›
  ```

  where `check_latex_measure` is a hand-programmed function that checks the input for syntactical and static semantic constraints.

4.3 The Standard Ontology Libraries

We will describe the backbone of the Standard Library with the already mentioned hierarchy `COL` (the common ontology library), `scholarly_paper` (for MINT-oriented scientific papers), `technical_report` (for MINT-oriented technical reports), and the example for a domain-specific ontology `CENELEC_50128`.

4.3.1 Common Ontology Library (COL)

Isabelle/DOF provides a Common Ontology Library (COL) [3] that introduces several ontology concepts; its overall class-tree it provides looks as follows:

```
├── Isa_COL.text_element
│   ├── Isa_COL.chapter
│   ├── Isa_COL.section
│   ├── Isa_COL.subsection
│   └── Isa_COL.subsubsection
├── Isa_COL.figure
│   └── Isa_COL.side_by_side_figure
├── Isa_COL.figure_group
└── ...
```

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''}
```

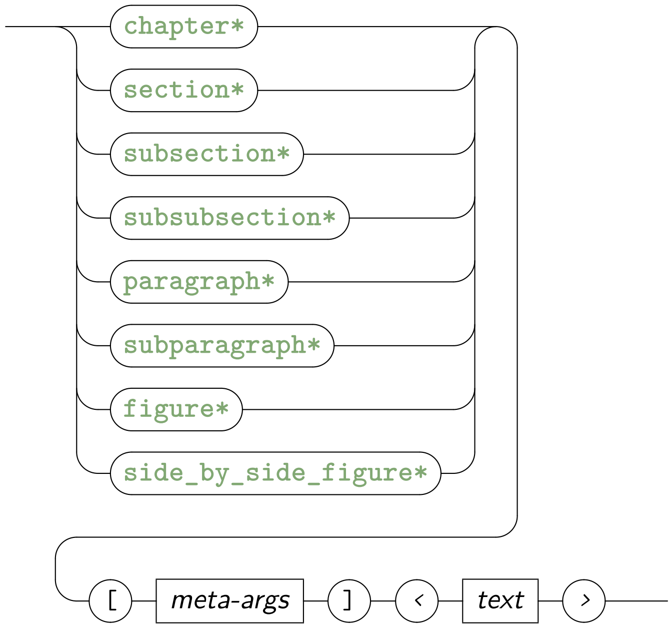
As mentioned in Section 3.3.2, *index.level* defines the section-level (e. g., using a LaTeX-inspired hierarchy: from *Some −1* (corresponding to `\part`) to *Some 0* (corresponding to `\chapter`, respectively, **chapter***) to *Some 3* (corresponding to `\subsubsection`, respectively, **subsubsection***). 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 requires that technical text section are introduce by a section element).

The attribute *referentiable* captures the information if a text-element can be a target for a reference, which is the case for sections or subsections, for example, but not arbitrary elements such as, i. e., paragraphs (this mirrors restrictions of the target LaTeX representation). The attribute *variants* refers to an Isabelle-configuration attribute that permits to steer the different versions of a LaTeX-presentation of the integrated source.

For further information of the root classes such as *figure*'s, please consult the ontology in *Isabelle_DOF.Isa_COL* directly. `COL` finally provides macros that extend the command-language of the DOF core by the following abbreviations:

- *derived_text_element* :

[3] contained in *Isabelle_DOF.Isa_COL*



The command syntax follows the implicit convention to add a "*" to distinguish them from the (similar) standard Isabelle text-commands which are not ontology-aware.

4.3.2 The Ontology `scholarly_paper`

The `scholarly_paper` ontology is oriented towards the classical domains in science: mathematics, informatics, natural sciences, technology, or engineering.

It extends `COL` by the following concepts:

```
 ├── scholarly_paper.title
 ├── scholarly_paper.subtitle
 ├── scholarly_paper.author ............................. An Author Entity Declaration
 ├── scholarly_paper.abstract
 ├── Isa_COL.text_element
 │   └── scholarly_paper.text_section...................... Major Paper Text-Elements
 │       ├── scholarly_paper.introduction ............................................
 │       ├── scholarly_paper.conclusion ..............................................
 │       │   └── scholarly_paper.related_work ........................................
 │       ├── scholarly_paper.bibliography ............................................
 │       ├── scholarly_paper.annex ...................................................
 │       ├── scholarly_paper.example......................... Example in General Sense
 │       └── scholarly_paper.technical .................... Root for Technical Content
 │           ├── scholarly_paper.math_content ........................................
 │           │   ├── scholarly_paper.definition.................................Freeform
 │           │   ├── scholarly_paper.lemma......................................Freeform
 │           │   ├── scholarly_paper.theorem ...................................Freeform
 │           │   ├── scholarly_paper.corollary..................................Freeform
 │           │   ├── scholarly_paper.math_example ..............................Freeform
 │           │   ├── scholarly_paper.math_semiformal............................Freeform
 │           │   └── scholarly_paper.math_formal ............. Formal(=Checked) Content
 │           │       └── scholarly_paper.assertion............................Assertions
 │           ├── scholarly_paper.tech_example ........................................
 │           ├── scholarly_paper.math_motivation......................................
 │           ├── scholarly_paper.math_explanation ....................................
 │           ├── scholarly_paper.engineering_content..................................
 │           │   ├── scholarly_paper.data
 │           │   ├── scholarly_paper.evaluation
 │           │   └── scholarly_paper.experiment
 │           └── ..
 ├── ..
 ├── scholarly_paper.article........................................ The Paper Monitor
 └── ...
```

A pivotal abstract class in the hierarchy is:

Isar

```
doc_class text_section = text_element +
  main_author :: author option  <=  None
  fixme_list  :: string list    <=  []
  level       :: int  option    <=  None
```

Besides attributes of more practical considerations like a *fixme_list*, that can be modified during the editing process but is only visible in the integrated source but usually ignored in the

LaTeX, this class also introduces the possibility to assign an "ownership" or "responsibility" of a *text_element* to a specific *author*. Note that this is possible since Isabelle/DOF assigns to each document class also a class-type which is declared in the HOL environment.

Recall that concrete authors can be denoted by term-antiquotations generated by Isabelle/DOF; for example, this may be for a text fragment like

```
text*[...::example, main_author = Some(@{author ''bu''})] ‹ ... ›
```
Isar

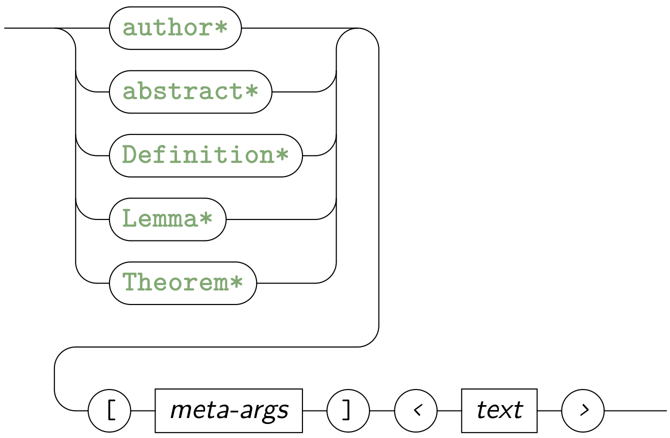
or

```
text*[...::example, main_author = Some(@{author ‹bu›})] ‹ ... ›
```
Isar

where *''bu''* is a string presentation of the reference to the author text element (see below in Section 4.3.1).

Some of these concepts were supported as command-abbreviations leading to the extension of the Isabelle/DOF language:

- *derived_text_elements* :



Usually, command macros for text elements will assign the generated instance to the default class corresponding for this class. For pragmatic reasons, `Definition*`, `Lemma*` and `Theorem*` represent an exception to this rule and are set up such that the default class is the super class *math_content* (rather than to the class *definition*). This way, it is possible to use these macros for several sorts of the very generic concept "definition", which can be used as a freeform mathematical definition but also for a freeform terminological definition as used in certification standards. Moreover, new subclasses of *math_content* might be introduced in a derived ontology with an own specific layout definition.

While this library is intended to give a lot of space to freeform text elements in order to counterbalance Isabelle's standard view, it should not be forgotten that the real strength of Isabelle is its ability to handle both, and to establish links between both worlds. Therefore, the formal assertion command has been integrated to capture some form of formal content.

Examples

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›
author*[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›
author*[bu::author, email = ‹wolff@lri.fr›,
       affiliation = ‹Université Paris−Saclay, LRI, Paris, France›]‹Burkhart Wolff›
```

Assertions allow for logical statements to be checked in the global context. This is particularly useful to explore formal definitions wrt. their border cases.

```
Isar
assert*[ass1::assertion, short_name = ‹This is an assertion›] ‹last [3] < (4::int)›
```

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

```
Isar
text*[claim::assertion]‹For non−empty lists, our definition yields indeed
                          the last element of a list.›
assert*[claim1::assertion] ‹last[4::int] = 4›
assert*[claim2::assertion] ‹last[1,2,3,4::int] = 4›
```

As mentioned before, the command macros of Definition*, Lemma* and Theorem* set the default class to the super-class of *definition*. However, in order to avoid the somewhat tedious consequence:

```
Isar
Theorem*[T1::theorem, short_name=‹DF definition captures deadlock−freeness›] ‹ ... ›
```

the choice of the default class can be influenced by setting globally an attribute such as

```
Isar
declare[[Definition_default_class = definition]]
declare[[Theorem_default_class = theorem]]
```

which allows the above example be shortened to:

```
Isar
Theorem*[T1, short_name=‹DF definition captures deadlock−freeness›] ‹ ... ›
```

4.3.3 The Ontology `technical_report`

The `technical_report` ontology in *Isabelle_DOF.technical_report* extends `scholarly_paper` by concepts needed for larger reports in the domain of mathematics and engineering. The concepts are fairly high-level arranged at root-class level,

```
├── technical_report.front_matter ........................................................
├── technical_report.table_of_contents ...................................................
├── Isa_COL.text_element .................................................................
│   └── scholarly_paper.text_section .....................................................
│           └── technical_report.code ....................................................
│               ├── technical_report.SML .................................................
│               ├── technical_report.ISAR ................................................
│               └── technical_report.LATEX ...............................................
├── technical_report.index ...............................................................
├── ..
└── technical_report.report ..............................................................
```

4.3.4 A Domain-Specific Ontology: `CENELEC_50128`

The `CENELEC_50128` ontology in *Isabelle_DOF.CENELEC_50128* is an example of a domain-specific ontology. It is based on `technical_report` since we assume that this kind of format will be most appropriate for this type of long-and-tedious documents,

- CENELEC_50128.judgement
- CENELEC_50128.test_item
  - CENELEC_50128.test_case
  - CENELEC_50128.test_tool
  - CENELEC_50128.test_result
  - CENELEC_50128.test_adm_role
  - CENELEC_50128.test_environment
  - CENELEC_50128.test_requirement
  - CENELEC_50128.test_specification
- CENELEC_50128.objectives
- CENELEC_50128.design_item
  - CENELEC_50128.interface
- CENELEC_50128.sub_requirement
- CENELEC_50128.test_documentation
- Isa_COL.text_element
  - CENELEC_50128.requirement
    - CENELEC_50128.TC
    - CENELEC_50128.FnI
    - CENELEC_50128.SIR
    - CENELEC_50128.CoAS
    - CENELEC_50128.HtbC
    - CENELEC_50128.SILA
    - CENELEC_50128.assumption
      - CENELEC_50128.AC
        - CENELEC_50128.EC
          - CENELEC_50128.SRAC
    - CENELEC_50128.hypothesis
      - CENELEC_50128.security_hyp
    - CENELEC_50128.safety_requirement
  - CENELEC_50128.cenelec_text
    - CENELEC_50128.SWAS
    - [...]
  - scholarly_paper.text_section
    - scholarly_paper.technical
      - scholarly_paper.math_content
        - CENELEC_50128.semi_formal_content
- ...

Examples

The category "exported constraint (EC)" is, in the file `../../../src/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 assumption = requirement +
    assumption_kind :: ass_kind <= informal
doc_class AC = assumption +
  is_concerned :: role set <= UNIV
doc_class EC = AC  +
    assumption_kind :: ass_kind <=  formal
```

We now define the document representations, in the file `../../../src/ontologies/CENELEC_50128/DOF-CENELEC_50128.sty`. Let us assume that we want to register the definition of EC's 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 *requirement.long_name*, inherited from the document class *requirement*. The representation of *EC*'s 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}%
}
```

For Isabelle Hackers: 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 [23]. 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 dispatchers 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:

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

4.4 Advanced ODL Concepts

4.4.1 Example

We assume in this section the following local ontology:

```
doc_class title =
  short_title :: string option <= None
doc_class author =
  email :: string <= ''''
datatype classification = SIL0 | SIL1 | SIL2 | SIL3 | SIL4
doc_class abstract =
  keywordlist :: string list <= []
  safety_level :: classification <= SIL3
doc_class text_section =
  authored_by :: author set <= {}
  level :: int option <= None
type_synonym notion = string
doc_class introduction = text_section +
  authored_by :: author set  <= UNIV
  uses :: notion set
doc_class claim = introduction +
  based_on :: notion list
doc_class technical = text_section +
  formal_results :: thm list
doc_class definition = technical +
  is_formal :: bool
  property  :: term list <= []
datatype kind = expert_opinion | argument | proof
doc_class result = technical +
  evidence :: kind
  property :: thm list <= []
doc_class example = technical +
  referring_to :: (notion + definition) set <= {}
doc_class conclusion = text_section +
  establish :: (claim × result) set
```

4.4.2 Meta-types as Types

To express the dependencies between text elements to the formal entities, e. g., `term` (λ-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 types declared in HOL, which are "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 ϑ. For example, the *establish* attribute in our example 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 ϑ where this theorem is established. Similarly, attribute values like *property* = @{*term* ‹*A* $\longleftrightarrow$ *B*›} require that the HOL-string $A \longleftrightarrow B$ is again type-checked and represents indeed a formula in ϑ. Another instance of this process, which we call *second-level type-checking*, are term-constants generated from the ontology such as @{`definition` <*string*>}.

4.4.3 ODL Class Invariants

Ontological classes as described so far are too liberal in many situations. There is a first high-level syntax implementation for class invariants. These invariants can be checked when an instance of the class is defined. To enable the checking of the invariants, the *invariants_checking* theory attribute must be set:

Isar
```
declare[[invariants_checking = true]]
```

For example, let's define the following two classes:

Isar
```
doc_class class_inv1 =
  int1 :: int
  invariant inv1 :: int1 σ ≥ 3

doc_class class_inv2 = class_inv1 +
  int2 :: int
  invariant inv2 :: int2 σ < 2
```

The σ symbol is reserved and references the future instance class. By relying on the implementation of the Records in Isabelle/HOL [23], one can reference an attribute of an instance using its selector function. For example, *int1* σ denotes the value of the *int1* attribute of the future instance of the class *class_inv1*.

Now let's define two instances, one of each class:

```
Isar
text*[testinv1::class_inv1, int1=4]‹lorem ipsum...›
text*[testinv2::class_inv2, int1=3, int2=1]‹lorem ipsum...›
```

The value of each attribute defined for the instances is checked against their classes invariants. As the class *class_inv2* is a subsclass of the class *class_inv1*, it inherits *class_inv1* invariants. Hence, the *inv1* invariant is checked when the instance *testinv2* is defined.

Now let's add some invariants to our example in Section 4.4.1. 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 total. In a high-level syntax, this type of constraints could be expressed, e. g., by:

```
Isar
doc_class introduction = text_section +
  authored_by :: author set  <= UNIV
  uses :: notion set
  invariant author_finite :: finite (authored_by σ)
doc_class result = technical +
  evidence :: kind
  property :: thm list <= []
  invariant has_property :: evidence σ = proof ⟷ property σ ≠ []
doc_class example = technical +
  referring_to :: (notion + definition) set <= {}
doc_class conclusion = text_section +
  establish :: (claim × result) set
  invariant total_rel :: ∀ x. x ∈ Domain (establish σ)
                              ⟶ (∃ y ∈ Range (establish σ). (x, y) ∈ establish σ)
```

All specified constraints are already checked in the IDE of DOF while editing. The invariant *author_finite* enforces that the user sets the *authored_by* set. There are still some limitations with this high-level syntax. For now, the high-level syntax does not support monitors (see Section 4.4.4). For example, one would like to delay a final error message till the closing of a monitor. For this use-case you can use low-level class invariants (see Section 4.4.5).

4.4.4 ODL Monitors

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

```
Isar
doc_class article =
   style_id :: string              <= ''LNCS''
   version  :: (int × int × int)  <= (0,0,0)
   accepts (title ~~ ⟦subtitle⟧ ~~ ⦃author⦄⁺ ~~ abstract ~~ ⦃introduction⦄⁺
                ~~ ⦃background⦄* ~~ ⦃technical || example ⦄⁺ ~~ ⦃conclusion⦄⁺
                ~~ bibliography ~~ ⦃annex⦄* )
```

Semantically, monitors introduce a behavioral element into ODL:

Isar
```
open_monitor*[this::article]
  ...
close_monitor*[this]
```

Inside the scope of a monitor, all instances of classes mentioned in its `accepts_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 `rejects_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. For now, the high-level syntax of invariants is not supported for monitors and you must use the low-level class invariants (see Section 4.4.5.

4.4.5 ODL Low-level Class Invariants

If one want to go over the limitations of the actual high-level syntax of the invariant, one can define a function using SML. A formulation, in SML, of the class-invariant *has_property* in Section 4.4.3, defined in the supposedly *Low_Level_Syntax_Invariants* theory (note the long name of the class), is straight-forward:

SML

```
fun check_result_inv oid {is_monitor:bool} ctxt =
  let
    val kind =
      AttributeAccess.compute_attr_access ctxt "evidence" oid <@>{here} <@>{here}
    val prop =
      AttributeAccess.compute_attr_access ctxt "property" oid <@>{here} <@>{here}
    val tS = HOLogic.dest_list prop
  in  case kind 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
                        "Low_Level_Syntax_Invariants.result" check_result_inv)
```

The `Theory.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.

4.4.6 Queries On Instances

Any class definition generates term antiquotations checking a class instance or the set of instances in a particular logical context; these references were elaborated to objects they refer to. This paves the way for a new mechanism to query the "current" instances presented as a HOL *list*. Arbitrarily complex queries can therefore be defined inside the logical language. To get the list of the properties of the instances of the class *result*, or to get the list of the authors of the instances of *introduction*, it suffices to treat this meta-data as usual:

```
value*‹map (result.property) @{result−instances}›
value*‹map (text_section.authored_by) @{introduction−instances}›
```

In order to get the list of the instances of the class *myresult* whose *evidence* is a *proof*, one can use the command:

```
value*‹filter (λσ. result.evidence σ = proof) @{result−instances}›
```

The list of the instances of the class *introduction* whose *level* > 1, can be filtered by:

```
value*‹filter (λσ. the (text_section.level σ) > 1)
     @{introduction−instances}›
```

4.5 Technical Infrastructure

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 dof_mkroot -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, which cannot be re-distributed due to copyright restrictions.

4.5.1 Developing Ontologies and their Representation Mappings

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 an Isabelle theory file and a LaTeX-style file:

```
src
  ontologies ........................................................ Ontologies
    ontologies.thy ................................. Ontology Registration
    scholarly_paper ......................................... scholarly_paper
      scholarly_paper.thy
      DOF-scholarly_paper.sty
    technical_report ....................................... technical_paper
      technical_report.thy
      DOF-technical_report.sty
    CENELEC_50128 ............................................. CENELEC_50128
      CENELEC_50128.thy
      DOF-CENELEC_50128.sty
    ...
```

Developing a new ontology "`foo`" requires 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-afp` option:

Bash
```
achim@logicalhacking:~/Isabelle_DOF-1.3.0_2021-1$ ./install  \
    --skip-afp
```

4.5.2 Document Templates

Document-templates define the overall layout (page size, margins, fonts, etc.) of the generated documents. 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-afp` option:

Bash
```
achim@logicalhacking:~/Isabelle_DOF-1.3.0_2021-1$ ./install  \
    --skip-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.

4.6 Defining Document Templates

4.6.1 The Core Template

Document-templates define the overall layout (page size, margins, fonts, etc.) of the generated documents. If a new layout is already supported by a LaTeX-class, then developing basic support for it is straightforward: 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 for 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 15-17). 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 4.5). The common structure of an Isabelle/DOF document template looks as follows:

LaTeX

```
1  \documentclass{article}    % The LaTeX-class of your template
2  %% The following part is (mostly) required by Isabelle/DOF, do not modify
3  \usepackage[T1]{fontenc}   % Font encoding
4  \usepackage[utf8]{inputenc} % UTF8 support
5  \usepackage{xcolor}
6  \usepackage{isabelle,isabellesym,amssymb} % Required (by Isabelle)
7  \usepackage{amsmath}       % Used by some ontologies
8  \bibliographystyle{abbrv}
9  \input{dof-common} % setup shared between all Isabelle/DOF templates
10 \usepackage{graphicx}      % Required for images.
11 \usepackage[caption]{subfig}
12 \usepackage[size=footnotesize]{caption}
13 \usepackage{hyperref}      % Required by Isabelle/DOF
14
15 %% Begin of template specific configuration
16 \urlstyle{rm}
17 \isabellestyle{it}
18
19 %% Main document, do not modify
20 \begin{document}
21 \maketitle\input{session}
22 \IfFileExists{root.bib}{\bibliography{root}}{}
23 \end{document}
```

4.6.2 Tips, Tricks, and Known Limitations

In this section, we will discuss several tips and tricks for developing new or adapting existing document templates or LaTeX-representations of ontologies.

Getting Started

In general, we recommend creating a test project (e. g., using `isabelle dof_mkroot`) 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$ lualatex 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 for individual LaTeX invocations:

```bash
achim@logicalhacking:~/MyProject/output/document$ max_print_line=200  \
    error_line=200 half_error_line=100 lualatex 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 of 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 `../../../src/ontologies/scholarly_paper/DOF-scholarly_paper.sty` or `../../../src/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 of the representations for the author and affiliation information is particularly challenging 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 `../../../src/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:

LaTeX

```
\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 beginning of the document while Isabelle/DOF allows defining 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:

```
\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:

```
\@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 `../../../src/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 LaTeX supports provides support for the `scrrept` -class which is not supported by the LaTeX support 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 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 [23]. 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 class instances 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 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

See Section 4.4.5.

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 4.4.4). 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 4.3.2). As generic *doc_items* are derived from the text element, the environment `isamarkuptext*` builds the core of Isabelle/DOF's LaTeX implementation. For example, the SRAC 1 from page 30 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 LaTeX command via (again, recall Section 4.3.2):

```latex
 \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`:

```latex
%% 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*. Texas Advanced Computing Center, 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, 2020. 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, 2020. 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] S. Taha, B. Wolff, and L. Ye. Philosophers may dine — definitively! In *International Conference on Integrated Formal Methods (IFM)*, number to appear in Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, 2020.

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

[21] 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.

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

[23] M. Wenzel. *The Isabelle/Isar Reference Manual*, 2020. Part of the Isabelle distribution.

[24] M. Wenzel. The Isabelle system manual, 2020. Part of the Isabelle distribution.

[25] 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