

Isabelle/DOF

User and Implementation Manual

Achim D. Brucker

Nicolas Méric

Burkhart Wolff

April 24, 2025



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

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

Copyright © 2019–2024 University of Exeter, UK
2018–2024 Université Paris-Saclay, France
2018–2019 The University of Sheffield, UK

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

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

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

SPDX-License-Identifier: BSD-2-Clause

This manual describes Isabelle/DOF as available in the Archive of Formal Proofs (AFP). The latest development version as well as releases that can be installed as Isabelle component 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 and Paolo Crisafulli.

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

Contents

1	Introduction	7
2	Background	11
2.1	The Isabelle System Architecture	11
2.2	The Document Model Required by DOF	11
2.3	Implementability of the Document Model in other ITP's	15
3	Isabelle/DOF: A Guided Tour	17
3.1	Getting Started	17
3.1.1	Installation	17
3.2	Writing Documents	18
3.2.1	Document Generation	18
3.2.2	Name-Spaces, Long- and Short-Names	19
3.2.3	Caveat: Lexical Conventions of Cartouches, Strings, Names,	20
3.3	Writing Academic Publications in <i>scholarly_paper</i>	20
3.3.1	Editing Major Examples	20
3.3.2	A Bluffers Guide to the <i>scholarly_paper</i> Ontology	20
3.3.3	Writing Academic Publications: A Freeform Mathematics Text	22
3.3.4	More Freeform Elements, and Resulting Navigation	25
3.3.5	Using Term-Antiquotations	26
3.4	Writing Technical Reports in <i>technical_report</i>	28
3.4.1	A Technical Report with Tight Checking	29
3.5	Some Recommendations: A little Style Guide	30
4	Proofs over Ontologies	33
4.1	Proving Properties over Ontologies	33
4.1.1	Ontology-Morphisms: a Prototypical Example	33
4.1.2	Proving the Preservation of Ontological Mappings : A Document- Ontology Morphism	34
4.1.3	Proving the Preservation of Ontological Mappings : A Domain- Ontology Morphism	36
4.1.4	Proving Monitor-Refinements	39
5	Ontologies and their Development	41
5.1	The Ontology Definition Language (ODL)	42
5.1.1	Some Isabelle/HOL Specification Constructs Revisited	43
5.1.2	Defining Document Classes	45

Contents

5.2	The main Ontology-aware Document Elements	48
5.2.1	General Syntactic Elements for Document Management	49
5.2.2	Ontological Code-Contexts and their Management	50
5.2.3	Ontological Term-Contexts and their Management	51
5.2.4	Status and Query Commands	53
5.2.5	Macros	54
5.3	The Standard Ontology Libraries	55
5.3.1	Common Ontology Library (COL)	55
5.3.2	The Ontology <code>scholarly_paper</code>	57
5.3.3	The Ontology <code>technical_report</code>	61
5.4	Advanced ODL Concepts	63
5.4.1	Example	63
5.4.2	Meta-types as Types	64
5.4.3	ODL Class Invariants	65
5.4.4	ODL Low-level Class Invariants	66
5.4.5	ODL Monitors	67
5.4.6	Queries On Instances	69
5.5	Technical Infrastructure	69
5.5.1	The Previewer	69
5.5.2	Developing Ontologies and their Representation Mappings	69
5.5.3	Document Templates	71
5.6	Defining Document Templates	71
5.6.1	The Core Template	71
5.6.2	Tips, Tricks, and Known Limitations	72
6	Extending Isabelle/DOF	77
6.1	Isabelle/DOF: A User-Defined Plugin in Isabelle/Isar	77
6.2	Programming Antiquotations	79
6.3	Implementing Second-level Type-Checking	80
6.4	Programming Class Invariants	80
6.5	Implementing Monitors	80
6.6	The \LaTeX -Core of Isabelle/DOF	81

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

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

Contents

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 plays is *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]. All these documents contain formal content and have to follow a given structure. In practice, large groups of developers have to produce a substantial set of documents where consistency is notoriously difficult to maintain. In particular, certifications are centred around the *traceability* of requirements throughout the entire set of documents. While technical solutions for the traceability problem exist (most notably: DOORS [12]), 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*), components (called *attributes*) of the concept, and properties (called *invariants*) on concepts. Logically, classes are represented by a type (the class type) and particular terms representing *instances* of them. Since components are typed, it is therefore possible to express *links* like *m-to-n* relations between classes. Another form of link between concepts is the *is-a* relation declaring the instances of a subclass to be instances of the super-class.

Engineering an ontological language for documents that contain both formal and informal elements as occurring in formal theories is a particular challenge. To address this latter, 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 an 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 the 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

1 Introduction

is a plugin into the Isabelle/Isar framework in the style of [24]. However, Isabelle/DOF will generate from ontologies a theory infrastructure consisting of types, terms, theorems and code that allows both interactive checking and formal reasoning over meta-data related to annotated documents.

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 on 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 5.
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 6.

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.:

```
lemma refl x = x  
  by simp
```

Isar

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

The axiom refl

Document

- a red background for SML-code:

```
fun id x = x
```

SML

- a yellow background for \LaTeX -code:

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

\LaTeX

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

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

Bash

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, N. Méric, and B. Wolff. Using Deep Ontologies in Formal Software Engineering. In *International Conference on Rigorous State-Based Methods (ABZ 2023)*, To appear in Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, 2023. 10.1007/978-3-031-33163-3_2.

A BibTeX-entry is available at: <https://www.lri.fr/~wolff/bibtex/wolff.html>.

- an older description of the 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. https://doi.org/10.1007/978-3-030-34968-4_4.

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, Isabelle provides an infrastructure for Isabelle plug-ins, comprising extensible state components and extensible syntax that can be bound to SML programs. Thus, the Isabelle architecture may be understood as an extension and refinement of the traditional ‘LCF approach’, with explicit infrastructure for building derivative systems.*” [24]

The current system framework offers moreover the following features:

- a build management grouping components into to pre-compiled sessions,
- a prover IDE (PIDE) framework [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 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.¹

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 docu-*

¹We use the term *plugin* for a collection of HOL-definitions, SML and Scala code in order to distinguish it from the official Isabelle term *component* which implies a particular format and support by the Isabelle build system.

2 Background

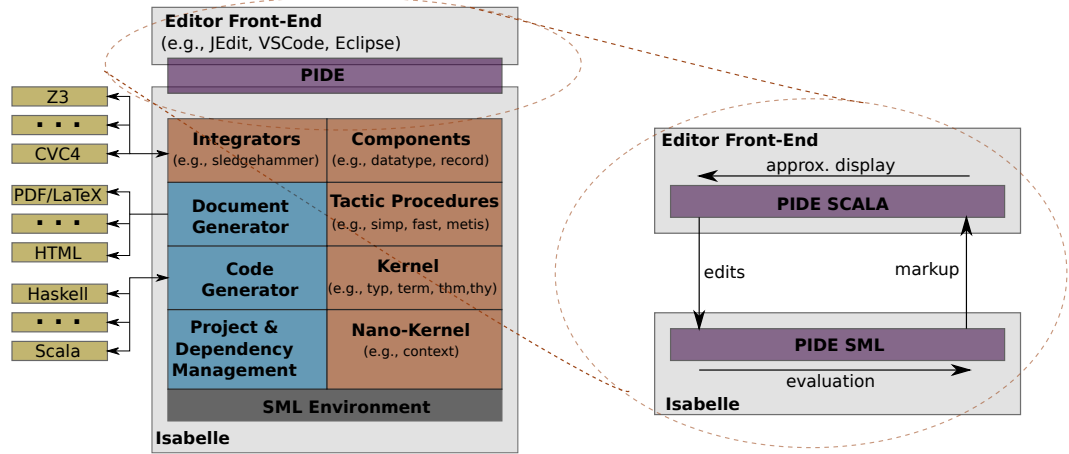


Figure 2.1: The system architecture of Isabelle (left-hand side) and the asynchronous communication between the Isabelle system and the IDE (right-hand side).

ment, *sub-document*, *document-element*, and *semantic macros* occurring inside document-elements. This type of document structure is quite common for scripts interactively evaluated in an incremental fashion. Furthermore, we assume a bracketing mechanism that unambiguously allows to separate different syntactic fragments and that can be nested. In the case of Isabelle, these are the guillemot symbols $\langle \dots \rangle$, which represent the begin and end of a *cartouche*.

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 (c.f. Figure 2.2). Document parts 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 (left-hand side)). 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:

theory <i>Example</i>	— Name of the 'theory'
imports	— Declaration of 'theory' dependencies
<i>Main</i>	— Imports a library called 'Main'
keywords	— Registration of keywords defined locally
<i>requirement</i>	— A command for describing requirements

Isar

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

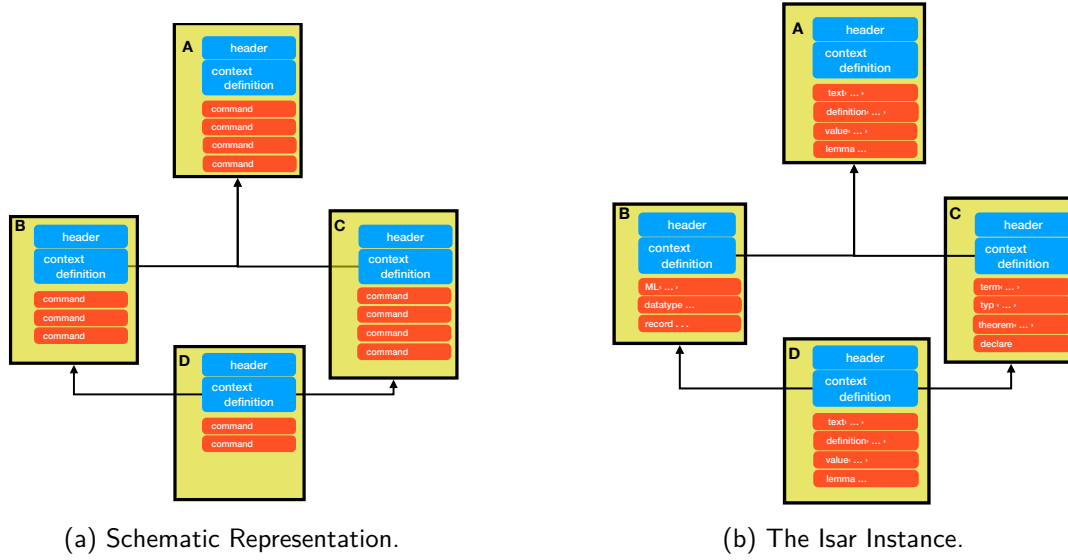


Figure 2.2: A Representation of a Document Model.

from each other.

The body of a theory file consists of a sequence of *commands* that must be introduced by a command keyword such as *requirement* above. Command keywords may mark the the begin of a text that is parsed by a command-specific parser; the end of the command-span is defined by the next keyword. Commands were used to define definitions, lemmas, code and text-elements (see Figure 2.2 (right-hand side)).

A simple text-element may look like this:

text⟨ *This is a simple text.* ⟩

Isar

...so it is a command **text** followed by an argument (here in ⟨ ... ⟩ parenthesis) which contains characters. While **text**-elements play a major role in this manual—document generation is the main use-case 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[*labelclassid*, *attr*₁=*E*₁, ... *attr*=*E*]⟨ *some semiformal text* ⟩
ML[*labelclassid*, *attr*₁=*E*₁, ... *attr*=*E*]⟨ *some SML code* ⟩
value[*labelclassid*, *attr*₁=*E*₁, ... *attr*=*E*]⟨ *some annotated term* ⟩

Other instances of document elements belonging to these families are, for example, the free-form **Definition** and **Lemma** as well as their formal counterparts **definition** and **lemma**, which allow in addition to their standard Isabelle functionality the creation and management of ontology-generated meta-data associated to them (cf. -Section 3.2).

Depending on the family, we will speak about (*formal*) *text-contexts*, (*ML*) *code-contexts*

2 Background

and *term-contexts* if we refer to sub-elements inside the $\langle \dots \rangle$ cartouches of these command families.

Text- code- or term contexts may contain a special form comment, that may be considered as a "semantic macro" or a machine-checked annotation: the so-called antiquotations. Its general syntactic format reads as follows:

```
@{antiquotation_name (args) [more_args] ⟨subcontext⟩ }
```

Isar

The sub-context may be different from the surrounding one; therefore, it is possible to switch from a text-context to a term-context, for example. Therefore, antiquotations allow the nesting of cartouches, albeit not all combinations are actually supported.² Isabelle comes with a number of built-in antiquotations for text- and code-contexts; a detailed overview can be found in [23]. DOF reuses this general infrastructure but *generates* its own families of antiquotations from ontologies.

An example for a text-element using built-in antiquotations may look like this:

```
text⟨ According to the *⟨reflexivity⟩ axiom @-thm refl",  
  we obtain in for @-term fac 5" the result @-value fac 5".⟩
```

Isar

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

The above text element is represented in the final document (e. g., a PDF) by:

According to the *reflexivity* axiom $x = x$, we obtain in
for *fac 5* the result 120.

Document

Antiquotations seen as semantic macros are partial functions of type *logical_context 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).

Therefore, semantic macros can 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 formal with the informal content.

Since Isabelle's commands are freely programmable, it is possible to implement DOF as an extension of the system. In particular, the ontology language of DOF provides an ontology definition language ODL that *generates* anti-quotations and the infrastructure to check and evaluate them. This allows for checking an annotated document with respect to a given ontology, which may be specific for a given domain-specific universe of discourse (see Figure 2.3). ODL will be described in Chapter 3 in more detail.

²In the literature, this concept has been referred to *CascadeSyntax* and was used in the Centaur-system and is existing in some limited form in some Emacs-implementations these days.

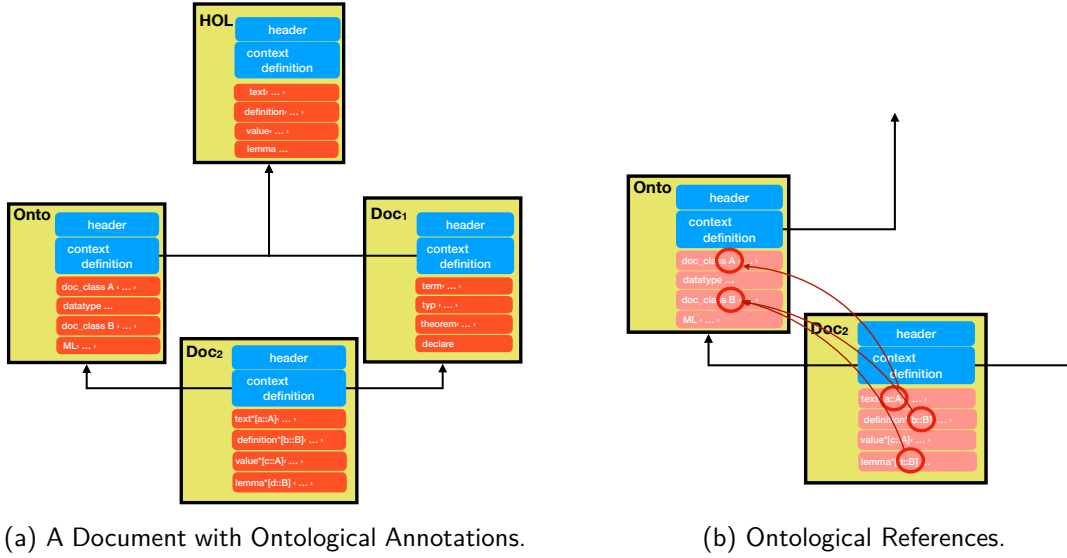


Figure 2.3: Documents conform to Ontologies.

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 [2, 11, 21, 22] which in many features over-accomplishes the required features of DOF.

We call the present implementation of DOF on the Isabelle platform Isabelle/DOF. Figure 2.4 shows a screenshot 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 [2, 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 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.

2 Background

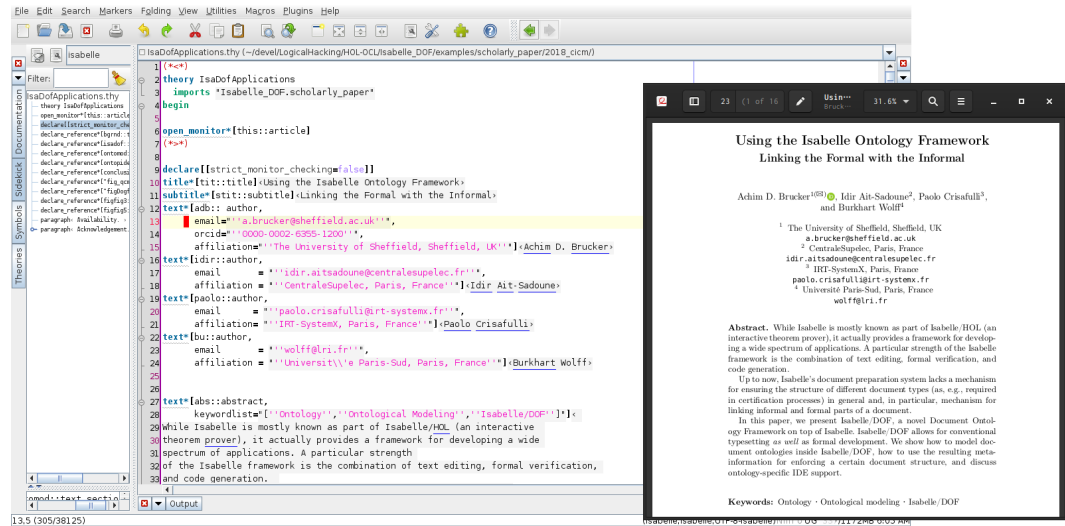


Figure 2.4: The Isabelle/DOF IDE (left) and the corresponding PDF (right), showing the first page of [5].

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 in ODL.

3.1 Getting Started

3.1.1 Installation

In this section, we will show how to install Isabelle/DOF. We assume a basic familiarity with a Linux/Unix-like command line (i.e., a shell). We focus on the installation of the latest official release of Isabelle/DOF as available in the Archive of Formal Proofs (AFP).¹ Isabelle/DOF requires Isabelle with a recent \LaTeX -distribution (e.g., Tex Live 2022 or later).

Installing Isabelle and the AFP. Please download and install the latest official Isabelle release from the Isabelle Website (<https://isabelle.in.tum.de>). After the successful installation of Isabelle, you should be able to call the `isabelle` tool on the command line:

```
achim@logicalhacking:~$ isabelle version
```

Bash

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.

Next, download the the AFP from <https://www.isa-afp.org/download/> and follow the instructions given at <https://www.isa-afp.org/help/> for installing the AFP as an Isabelle component.

Installing TeXLive. On a Debian-based Linux system (e.g., Ubuntu), the following command should install all required \LaTeX packages:

```
achim@logicalhacking:~$ sudo aptitude install texlive-full
```

Bash

Installing Isabelle/DOF

By installing the AFP in the previous steps, you already installed Isabelle/DOF. In fact, Isabelle/DOF is currently consisting out of three AFP entries:

¹If you want to work with the development version of Isabelle/DOF, please obtain its source code from the Isabelle/DOF Git repository (https://git.logicalhacking.com/Isabelle_DOF/Isabelle_DOF) and follow the instructions in provided `README.MD` file.

- `Isabelle_DOF`: This entry contains the Isabelle/DOF system itself, including the Isabelle/DOF manual.
- `Isabelle_DOF-Example-I`: This entry contains an example of an academic paper written using the Isabelle/DOF system oriented towards an introductory paper. The text is based on [5]; in the document, we deliberately refrain from integrating references to formal content in order to demonstrate that Isabelle/DOF can be used for writing documents with very little direct use of \LaTeX .
- `Isabelle_DOF-Example-II`: This entry contains another example of a mathematics-oriented academic paper. It is based on [19]. It represents 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, with respect to 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, and therefore does deliberately not exploit Isabelle/DOF’s full potential.

3.2 Writing Documents

3.2.1 Document Generation

Isabelle/DOF provides an enhanced setup for generating PDF document. In particular, it does not make use of a file called `document/root.tex`. Instead, the use of document templates and ontology representations is done within theory files. To make use of this feature, one needs to add the option `document_build = dof` to the `ROOT` file. An example `ROOT` file looks as follows:

```
session example = Isabelle_DOF +  
  options [document = pdf, document_output = "output", document_build = dof]  
(*theories [document = false]  
  A  
  theories  
  B*)
```

ROOT

The document template and ontology can be selected as follows:

```
theory C imports Isabelle_DOF.technical_report Isabelle_DOF.scholarly_paper begin  
  list_templates  
  use_template scrreprtmodern  
  list_ontologies  
  use_ontology technical_report and scholarly_paper  
end
```

Isar

The commands `list_templates` and `list_ontologies` can be used for inspecting (and selecting) the available ontologies and templates:

```
list_templates
list_ontologies
```

Isar

Note that you need to import the theories that define the ontologies that you want to use. Otherwise, they will not be available.

Warning. Note that the session `Isabelle_DOF` needs to be part of the “main” session hierarchy. Loading the Isabelle/DOF theories as part of a session section, e.g.,

```
session example = HOL +
  options [document = pdf, document_output = "output", document_build = dof]
  session
    Isabelle_DOF.scholarly_paper
  theories
    C
```

ROOT

will not work. Trying to build a document using such a setup will result in the following error message:

```
achim@logicalhacking:~$
isabelle build -D .
Running example ...
Bad document_build engine "dof"
example FAILED
```

Bash

3.2.2 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. Additionally, Isabelle/DOF objects also come with a proper name space: classes (and monitors), instances, low-level class invariants (SML-defined invariants) all follow the lexical conventions of Isabelle. For instance, a class can be referenced outside its theory using its short-name or its long-name if another class with

the same name is defined in the current theory. 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.

3.2.3 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 "srac₁_def"⟩ while **text**⟨@-thm ⟨srac₁_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 Editing Major Examples

The ontology *scholarly_paper* is an ontology modeling academic/scientific papers, with a slight bias towards texts in the domain of mathematics and engineering.

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 `Isabelle_DOF-Example-I/IsaDofApplications.thy`

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

```
achim@logicalhacking:~$ isabelle build Isabelle_DOF-Example-I
```

Bash

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:

```

doc_class title =
  short_title string option i= None

doc_class subtitle =
  abbrev      string option i= None

doc_class author =
  email       string i=
  http_site   string i=
  orcid       string i=
  affiliation  string

doc_class abstract =
  keywordlist  string list i= []
  principal_theorems thm list

```

Isar

Note *short_title* and *abbrev* are optional and have the default *None* (no value). Note further, that *abstracts* 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.

```

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

```

Isar

The *text_element.level*-attribute 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 .

3 Isabelle/DOF: A Guided Tour

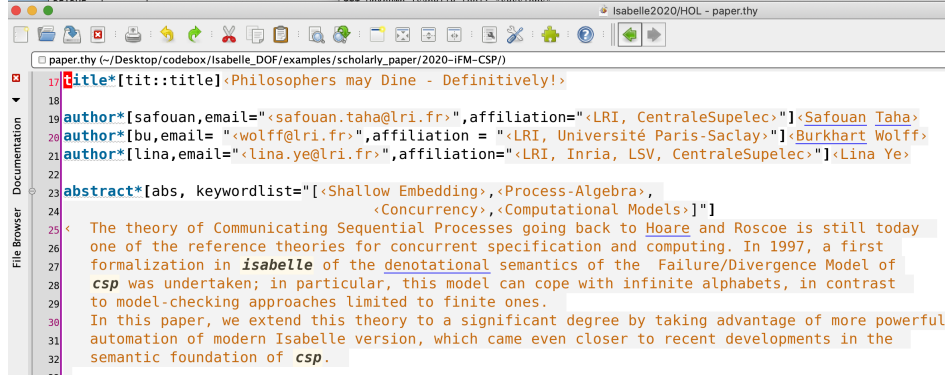


Figure 3.1: A mathematics paper as integrated document source ...

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.

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[[idi,class_idi,iattri]]< ... text ...>`, but there are a few built-in abbreviations like `title[[idi,iattri]]< ... 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_classes` of `scholarly_paper`; we say that these text elements are *instances* of the `doc_classes` 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`:

Philosophers may Dine - Definitively!

Safouan Taha¹, Burkhart Wolff², and Lina Ye³

¹ LRI, CentraleSupélec

`safouan.taha@lri.fr`

² LRI, Université Paris-Saclay

`wolff@lri.fr`

³ LRI, Inria, LSV, CentraleSupélec

`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.

```
doc_class technical = text_section + ...
```

Isar

```
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 j= defn ...
```

```
doc_class theorem = math_content +  
  mcc          math_content_class j= 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 *definitions* and *theorems* (the details of these class definitions are omitted here). Note, however, that class identifiers can be abbreviated by standard **type_synonyms** 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.

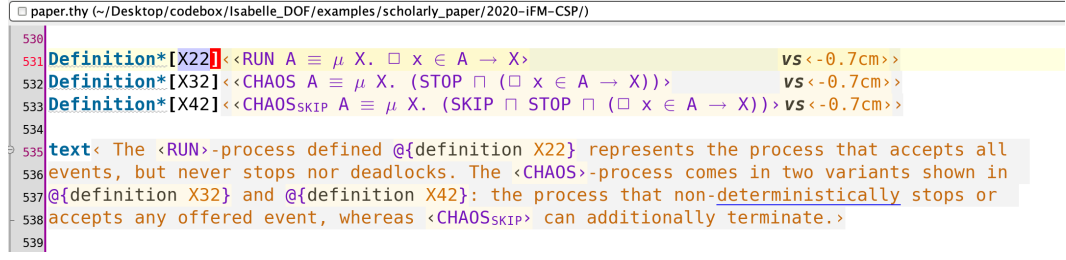


Figure 3.3: A screenshot of the integrated source with definitions ...

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, whereas *CHAOS_{SKIP}* can additionally terminate.

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

An example for a sequence of (Isabelle-formula-)texts, their ontological declarations as *definitions* 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 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 [styleparms] <...>` or just by `<...>` if the list of style parameters is empty,
- the freeform antiquotation for theory fragments written `@-theory_text [styleparms] <...>` 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.


```

326
327 figure*[fig1::figure, spawn_columns=False, relative_width="'90'",
328          src="'figures/Dogfood-Intro'"]
329      { * Ouroboros I: This paper from inside \ldots *}
330

```

Figure 3.5: Declaring figures in the integrated source.

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:

```

datatype placement = h t b ht hb
doc_class figure = text_section +
  relative_width int
  src string
  placement placement
  spawn_columns bool i= True

```

Isar

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:

```

doc_class article =
  style_id string i= LNCS
  version (int » int » int) i= (0,0,0)
  accepts (title [subtitles] -author"+ abstract -introduction"+
    -background' -technical example"+ -conclusion"+
    bibliography -annex' )

```

Isar

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

```

open_monitor [thisarticle]
...
close_monitor[this]

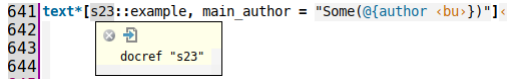
```

Isar

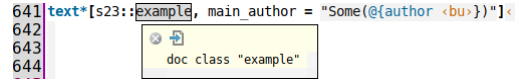
which signals to Isabelle/DOF begin and end of the part of the integrated source in which the text-elements instances are expected to appear in the textual ordering defined by *article*.

From these class definitions, Isabelle/DOF also automatically generated editing support for Isabelle/jEdit. In Figure 3.6(left) and Figure 3.6(right) we show how hovering over links permits to explore its meta-information. Clicking on a document class identifier permits

3 Isabelle/DOF: A Guided Tour

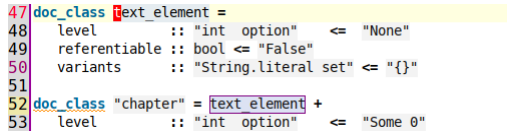


(a) Exploring a reference of a text-element.

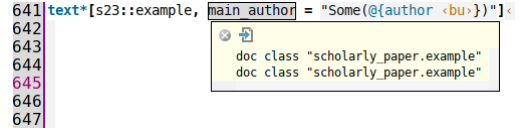


(b) Exploring the class of a text element.

Figure 3.6: Exploring text elements.



(a) Hyperlink to class-definition.



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

Figure 3.7: Navigation via generated hyperlinks.

to hyperlink into the corresponding class definition (Figure 3.7(left)); hovering over an attribute-definition (which is qualified in order to disambiguate; cf. Figure 3.7(right)) 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 it with a consistent bibtex usually makes disappear this behavior.

3.3.5 Using Term-Antiquotations

The present version of Isabelle/DOF is the first version that supports the novel feature of DOF-generated term-antiquotations, i. e., antiquotations embedded in HOL--terms possessing arguments that were validated in the ontological context. These -terms may occur in definitions, lemmas, or in values to define attributes in class instances. They have the format: `@-name arg1 ... arg1 " arg`

Logically, they are defined as an identity in the last argument `arg`; thus, ontologically checked prior arguments `arg1 ... arg1` can be ignored during a proof process; ontologically, they can be used to assure the traceability of, e. g., semi-formal assumptions throughout

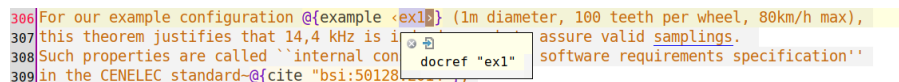


Figure 3.8: Exploring an ontological reference.

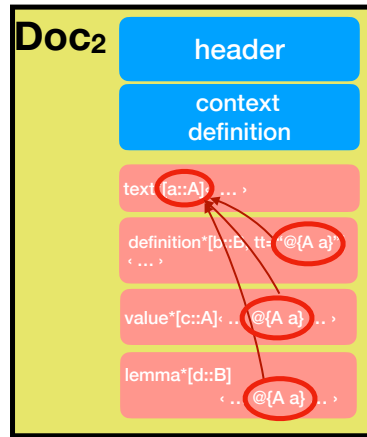


Figure 3.9: Term-Antiquotations Referencing to Annotated Elements

their way to formalisation and use in lemmas and proofs.

As shown in Figure 3.9, this feature of Isabelle/DOF substantially increases the expressibility of links between the formal and the informal in DOF documents.

In the following, we describe a common scenario linking semi-formal assumptions and formal ones:

```
declare_reference[e2definition]
```

Isar

```
Assumption[a1assumption, short_name=(safe_environment.)]
```

```
⟨The system shall only be used in the temperature range from 0 to 60 degrees Celsius.
```

```
Formally, this is stated as follows in @-definition (unchecked) ⟨e2⟩"⟩
```

```
definition[e2, status=formal] safe_env state bool
```

```
  where safe_env (temp -0 .. 60")
```

```
theorem[e3theorem] safety_preservation @-assumption ⟨a1⟩" (safe_env ) ...
```

Note that Isabelle proceeds in a strict “declaration-before-use”-manner, i. e. assumes linear visibility on all references. This also holds for the declaration of ontological references. In order to represent cyclic dependencies, it is therefore necessary to start with the forward declaration **declare_reference**. From there on, this reference can be used in text, term, and code-contexts, albeit its definition appears textually later. The corresponding freeform-formulation of this assumption can be explicitly referred in the assumption of a theorem establishing the link. The **theorem**-variant of the common Isabelle/Isar **theorem**-command will in contrast to the latter not ignore `⟨a1⟩`, i. e. parse just as string, but also validate it in the previous context.

Note that the **declare_reference** command will appear in the \LaTeX generated from this

document fragment. In order to avoid this, one has to enclose this command into the document comments : (i) ... (i).

3.4 Writing Technical Reports in *technical_report*

While it is perfectly possible to write documents in the `technical_report` ontology in freeform-style (this manual is mostly such an example), 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 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 corollary of the current context,
- `@{term <term> }` parses and type-checks *term*,
- `@{value <term> }` performs the evaluation of *term*,
- `@{ML <mlterm> }` parses and type-checks *mlterm*,
- `@{ML_file <mlfile> }` parses the path for *mlfile* and verifies its existence 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.

Additionally, documents antiquotations were added to check and evaluate terms with term antiquotations:

- `@{term_ <term> }` parses and type-checks *term* with term antiquotations, for instance `term_ <@-technical isadof>”` will parse and check that *isadof* is indeed an instance of the class *technical*,

```

215 text*[squiggles::technical]
216 <= Finally, a number of commonly used "squigglish" combinators is listed:
217
218 @ML "op ! : 'a Unsynchronized.ref->'a", access operation on a program variable vs<-0.3cm>
219 @ML "op := : ('a Unsynchronized.ref * 'a)->unit", update operation on program variable vs<-0.3cm>
220 @ML "op #> : ('a->'b) * ('b->'c)->'a->'c", a reversed function composition vs<-0.3cm>
221 @ML "I: 'a -> 'a", the I combinator vs<-0.3cm>
222 @ML "K: 'a -> 'b -> 'a", the K combinator vs<-0.3cm>
223 @ML "op o : (('b->'c) * ('a->'b))->'a->'c", function composition vs<-0.3cm>
224 @ML "op || : ('a->'b) * ('a->'b) -> 'a -> 'b", parse alternative vs<-0.3cm>
225 @ML "op -- : ('a->'b*'c) * ('c->'d*'e)->'a->('b*'d)*'e", parse pair vs<-0.3cm>

```

Figure 3.10: A table with a number of SML functions, together with their type.

- `@{value_ <term> }` performs the evaluation of *term* with term antiquotations, for instance `@{value_ <definition_list @-technical <isadof>'>}` will print the value of the *definition_list* attribute of the instance *isadof*. *value_* may have an optional argument between square brackets to specify the evaluator but this argument must be specified after a default optional argument already defined by the text antiquotation implementation. So one must use the following syntax if he does not want to specify the first optional argument: `@{value_ [] [nbe] <definition_list @-technical <isadof>'>}`. Note the empty brackets.

They are text-contexts equivalents to the **term** and **value** commands for term-contexts introduced in Section 5.2.3

3.4.1 A Technical Report with Tight Checking

An example of tight checking is a small programming manual 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.

This manual is written according to the *technical_report* ontology in *Isabelle_DOF.technical_report*. Figure 3.10 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.

3.5 Some Recommendations: A little 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.:

text \langle This is “*emph-emphasized*” and this is a
citation “*cite-brucker.eaisabelleontologies2018*” \rangle

Isar

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

text \langle This is \langle emphasized \rangle and this is a
citation @-cite brucker.eaisabelleontologies2018”.)

Isar

The list of standard Isabelle document antiquotations, as well as their options and styles, can be found in the Isabelle reference manual [23], section 4.2.

In practice, Isabelle/DOF documents with ambitious layout will contain a certain number of \LaTeX -commands, but this should be restricted to layout improvements that otherwise are (currently) not possible. As far as possible, raw \LaTeX should be restricted to the definition of ontologies and macros (see Chapter 5). If raw \LaTeX commands can not be avoided, it is recommended to use the Isabelle document comment $\backslash\langle^{\text{latex}}\rangle\langle\text{argument}\rangle$ to isolate these parts (cf. [23]).

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.

```

doc_class title = short_title string option i= None
doc_class affiliation =
  journal_style
doc_class author =
  affiliations affiliation list
  name string
  email string i=
  invariant ne_name name
doc_class text_element =
  authored_by ( author) set i= -"
  level int option i= None
  invariant authors_req authored_by -"
  and level_req the (level) i 1
doc_class introduction = text_element +
  authored_by ( author) set i= UNIV
doc_class technical = text_element +
  formal_results thm list
doc_class definition = technical +
  is_formal bool
doc_class theorem = technical +
  assumptions term list i= []
  statement term option i= None
doc_class conclusion = text_element +
  resume (definition set » theorem set)
  invariant ( xfst resume. is_formal x)
  ( ysnd resume. is_formal y)
doc_class article =
  style_id string i= LNCS
  accepts (title -author"+ -introduction"+
    -definition example"+ theorem"+ -conclusion"+)

```

lsar

Figure 3.11: A Basic Document Ontology: paper

4 Proofs over Ontologies

It is a distinguishing feature of DOF that it does not directly generate meta-data rather than generating a *theory of meta-data* that can be used in HOL-terms on various levels of the Isabelle-system and its document generation technology. Meta-data theories can be converted into executable code and efficiently used in validations, but also used for theoretic reasoning over given ontologies. While the full potential of this latter possibility still needs to be explored, we present in the following sections two applications:

1. Verified ontological morphisms, also called *ontological mappings* in the literature [1, 8, 10], i. e. proofs of invariant preservation along translation-functions of all instances of `doc_class`-es.
2. Verified refinement relations between the structural descriptions of theory documents, i. e. proofs of language inclusions of monitors of global ontology classes.

4.1 Proving Properties over Ontologies

4.1.1 Ontology-Morphisms: a Prototypical Example

We define a small ontology with the following classes:

```
doc_class AA = aa nat
doc_class BB = bb int
doc_class CC = cc int
```

```
doc_class DD = dd int
doc_class EE = ee int
doc_class FF = ff int
```

```
onto_morphism (AA, BB) to CC and (DD, EE) to FF
  where convert_» = ( CC.tag_attribute = 1int,
                      CC.cc = int(aa (fst )) + bb (snd ))
  and   convert_» = ( FF.tag_attribute = 1int,
                      FF.ff = dd (fst ) + ee (snd ))
```

Note that the *convert_»*-morphism involves a data-format conversion, and that the resulting transformation of *AA*-instances and *BB*-instances is surjective but not injective. The *CC.tag_attribute* is used to potentially differentiate instances with equal attribute-content and is irrelevant here.

This specification construct introduces the following constants and definitions:

- *convert_»* AA » BB CC

- `convert__ DD » EE FF`

and corresponding definitions.

4.1.2 Proving the Preservation of Ontological Mappings : A Document-Ontology Morphism

DOF as a system is currently particularly geared towards *document*-ontologies, in particular for documentations generated from Isabelle theories. We used it meanwhile for the generation of various conference and journal papers, notably using the *Isabelle_DOF.scholarly_paper* and *Isabelle_DOF.technical_report*-ontologies, targeting a (small) variety of \LaTeX style-files. A particular aspect of these ontologies, especially when targeting journals from publishers such as ACM, Springer or Elsevier, is the description of publication meta-data. Publishers tend to have their own styles on what kind meta-data should be associated with a journal publication; thus, the depth and precise format of affiliations, institution, their relation to authors, and author descriptions (with photos or without, hair left-combed or right-combed, etcpp...) varies.

In the following, we present an attempt to generalized ontology with several ontology mappings to more specialized ones such as concrete journals and/or the *Isabelle_DOF.scholarly_paper*- ontology which we mostly used for our own publications.

```
doc_class elsevier_org =  
  organization string  
  address_line string  
  postcode int  
  city string
```

```
doc_class acm_org =  
  position string  
  institution string  
  department int  
  street_address string  
  city string  
  state int  
  country string  
  postcode int
```

```
doc_class lncs_inst =  
  institution string
```

```
doc_class author =  
  name string  
  email string  
  invariant ne_fsnames name
```

```
doc_class elsevierAuthor = author +  
  affiliations elsevier_org list  
  firstname string
```

```

surname    string
short_author string
url string
footnote string
invariant ne_fsnames firstname    surname

```

```

doc_class acmAuthor = author +
  affiliations acm_org list
  firstname    string
  familyname    string
  orcid int
  footnote string
invariant ne_fsnames firstname    familyname

```

```

doc_class Incs_author = author +
  affiliations Incs list
  orcid int
  short_author string
  footnote string

```

```

definition acm_name where acm_name f s = f @    @ s

```

```

fun concatWith string string list string
  where concatWith str [] =
    concatWith str [a] = a
    concatWith str (a#R) = a@str@(concatWith str R)

```

```

lemma concatWith_non_mt (S[] ( sset S. s)) (concatWith sep S)
proof(induct S)
  case Nil
  then show ?case by simp
next
  case (Cons a S)
  then show ?case by (cases a; cases S; auto)
qed

```

```

onto_morphism (acm) to elsevier
  where convert =
    (elsevier.tag_attribute = acm.tag_attribute ,
     organization = acm.institution ,
     address_line = concatWith , [acm.street_address , acm.city ],
     postcode = acm.postcode ,
     city = acm.city )

```

Here is a more basic, but equivalent definition for the other way round:

```

definition elsevier_to_acm_morphism elsevier_org acm_org
  (↪ acm ↷ [1000]999)

```

4 Proofs over Ontologies

```

where acm = ( acm_org.tag_attribute = 1int,
               acm_org.position = no position,
               acm_org.institution = organization ,
               acm_org.department = 0,
               acm_org.street_address = address_line ,
               acm_org.city = elsevier_org.city ,
               acm_org.state = 0,
               acm_org.country = no country,
               acm_org.postcode = elsevier_org.postcode )

```

The following onto-morphism links *elsevierAuthor*'s and *acmAuthor*. Note that the conversion implies trivial data-conversions (renaming of attributes in the classes), string-representation conversions, and conversions of second-staged, referenced instances.

```

onto_morphism (elsevierAuthor) to acmAuthor
where convert =
  (author.tag_attribute = undefined,
   name = concatWith , [elsevierAuthor.firstname ,elsevierAuthor.surname ],
   email = author.email ,
   acmAuthor.affiliations = (elsevierAuthor.affiliations )
                           i, map (x. x acm ),
   firstname = elsevierAuthor.firstname ,
   familyname = elsevierAuthor.surname ,
   orcid = 0, — la triche !!!
   footnote = elsevierAuthor.footnote )

```

```

lemma elsevier_inv_preserved
  elsevierAuthor.ne_fsnames_inv
  acmAuthor.ne_fsnames_inv (convert )
  author.ne_fsnames_inv (convert )
unfolding elsevierAuthor.ne_fsnames_inv_def acmAuthor.ne_fsnames_inv_def
  convert _ _def author.ne_fsnames_inv_def
by auto

```

The proof is, in order to quote Tony Hoare, “as simple as it should be”. Note that it involves the lemmas like $?S \sqsubseteq (sset ?S. s \sqsubseteq) concatWith ?sep ?S \sqsubseteq$ which in itself require inductions, i. e., which are out of reach of pure ATP proof-techniques.

4.1.3 Proving the Preservation of Ontological Mappings : A Domain-Ontology Morphism

The following example is drawn from a domain-specific scenario: For conventional data-models, be it represented by UML-class diagrams or SQL-like “tables” or Excel-sheet like presentations of uniform data, we can conceive an ontology (which is equivalent here to a conventional style-sheet) and annotate textual raw data. This example describes how meta-data can be used to calculate and transform this kind of representations in a type-safe and verified way.

We model some basic enumerations as inductive data-types:

```
datatype Hardware_Type =
  Motherboard    Expansion_Card  Storage_Device  Fixed_Media
  Removable_Media Input_Device   Output_Device
```

```
datatype Software_Type =
  Operating_system  Text_editor  Web_Navigator  Development_Environment
```

In the sequel, we model a "Reference Ontology", i.e. a data structure in which we assume that standards or some de-facto-standard data-base refer to the data in the domain of electronic devices:

```
onto_class Resource =
  name string
```

```
onto_class Electronic = Resource +
  provider string
  manufacturer string
```

```
onto_class Component = Electronic +
  mass int
```

```
onto_class Simulation_Model = Electronic +
  simulate Component
  composed_of Component list
  version int
```

```
onto_class Informatic = Resource +
  description string
```

```
onto_class Hardware = Informatic +
  type Hardware_Type
  mass int
  composed_of Component list
invariant c1 mass = sum(map Component.mass (composed_of ))
```

```
onto_class Software = Informatic +
  type Software_Type
  version int
```

Finally, we present a *local ontology*, i.e. a data structure used in a local store in its data-base of cash-system:

```
onto_class Item =
  name string
```

```
onto_class Product = Item +
  serial_number int
  provider string
  mass int
```

```
onto_class Electronic_Component = Product +
```

4 Proofs over Ontologies

serial_number *int*

onto_class *Monitor* = *Product* +
 composed_of *Electronic_Component* *list*
 invariant *c2* *Product.mass* = *sum*(*map* *Product.mass* (*composed_of*))

term (*Product.mass* = *sum*(*map* *Product.mass* (*composed_of*)))

onto_class *Computer_Software* = *Item* +
 type *Software_Type*
 version *int*

These two example ontologies were linked via conversion functions called *morphisms*. The hic is that we can prove for the morphisms connecting these ontologies, that the conversions are guaranteed to preserve the data-invariants, although the data-structures (and, of course, the presentation of them) is very different. Besides, morphisms functions can be “forgetful” (i. e. surjective), “embedding” (i. e. injective) or even “one-to-one” ((i. e. bijective).

definition *Item_to_Resource_morphism* *Item* *Resource*
 (*⊢* *Resource* › [1000]999)
 where *Resource* =
 (*Resource.tag_attribute* = 1*int* ,
 Resource.name = *name*)

definition *Product_to_Resource_morphism* *Product* *Resource*
 (*⊢* *Resource* › [1000]999)
 where *Resource* =
 (*Resource.tag_attribute* = 2*int* ,
 Resource.name = *name*)

definition *Computer_Software_to_Software_morphism* *Computer_Software* *Software*
 (*⊢* *Software* › [1000]999)
 where *Software* =
 (*Resource.tag_attribute* = 3*int* ,
 Resource.name = *name* ,
 Informatic.description = *no description* ,
 Software.type = *type* ,
 Software.version = *version*)

definition *Electronic_Component_to_Component_morphism* *Electronic_Component* *Component*
 (*⊢* *Component* › [1000]999)
 where *Component* =
 (*Resource.tag_attribute* = 4*int* ,
 Resource.name = *name* ,
 Electronic.provider = *provider* ,
 Electronic.manufacturer = *no manufacturer* ,
 Component.mass = *mass*)

definition *Monitor_to_Hardware_morphism* *Monitor* *Hardware*

```

(⊆ Hardware ⊆ [1000]999)
where Hardware =
  ( Resource.tag_attribute = 5int ,
    Resource.name = name ,
    Informatic.description = no description,
    Hardware.type = Output_Device,
    Hardware.mass = mass ,
    Hardware.composed_of = map Electronic_Component_to_Component_morphism
(composed_of )
)

```

On this basis, we can state the following invariant preservation theorems:

```

lemma inv_c2_preserved
  c2_inv c1_inv ( Hardware )
  unfolding c1_inv_def c2_inv_def
  Monitor_to_Hardware_morphism_def Elec-
tronic_Component_to_Component_morphism_def
  by (auto simp comp_def)

```

```

lemma Monitor_to_Hardware_morphism_total
  Monitor_to_Hardware_morphism ' (¬XMonitor. c2_inv X)" (¬XHardware. c1_inv X)"
  using inv_c2_preserved
  by auto

```

```

type_synonym local_ontology = Item Electronic_Component Monitor
type_synonym reference_ontology = Resource Component Hardware

```

```

fun ontology_mapping local_ontology reference_ontology
  where ontology_mapping (x, y, z) = (xResource, yComponent, zHardware )

```

```

lemma ontology_mapping_total
  ontology_mapping ' ¬X. c2_inv (snd (snd X))" ¬X. c1_inv (snd (snd X))"
  using inv_c2_preserved
  by auto

```

Note that in contrast to conventional data-translations, the preservation of a class-invariant is not just established by a validation of the result, it is proven once and for all for all instances of the classes.

4.1.4 Proving Monitor-Refinements

Monitors are regular-expressions that allow for specifying instances of classes to appear in a particular order in a document. They are used to specify some structural aspects of a document. Based on an AFP theory by Tobias Nipkow on Functional Automata (i.e. a characterization of regular automata using functional polymorphic descriptions of transition functions avoiding any of the ad-hoc finitizations commonly used in automata theory), which comprises also functions to generate executable deterministic and non-deterministic automata, this theory is compiled to SML-code that was integrated in the DOF system. The neces-

4 Proofs over Ontologies

sary adaptations of this integration can be found in the theory *Isabelle_DOF.RegExpInterface*, which also contains the basic definitions and theorems for the concepts used here.

Recall that the monitor of *scholarly_paper.article* is defined by:

```
article_monitor title opt subtitle rep1 author abstract rep1 introduction -background'  
rep1 (technical example float) rep1 conclusion bibliography -annex'
```

However, it is possible to reason over the language of monitors and prove classical refinement notions such as trace-refinement on the monitor-level, so once-and-for-all for all instances of validated documents conforming to a particular ontology. The primitive recursive operators *Lang* and *L* generate the languages of the regular expression language, where *L* takes the sub-ordering relation of classes into account.

The proof of :

```
Lang article_monitor Lang report_monitor
```

can be found in theory *Isabelle_DOF.technical_report*; it is, again, "as simple as it should be" (to cite Tony Hoare).

The proof of:

```
L article_monitor L report_monitor
```

is slightly more evolved; this is due to the fact that DOF does not generate a proof of the acyclicity of the graph of the class-hierarchy *doc_class_rel* automatically. For a given hierarchy, this proof will always succeed (since DOF checks this on the meta-level, of course), which permits to deduce the anti-symmetry of the transitive closure of *doc_class_rel* and therefore to establish that the doc-classes can be organized in an order (i. e. the type *doc_class* is an instance of the type-class *order*). On this basis, the proof of the above language refinement is quasi automatic. This proof is also part of *Isabelle_DOF.technical_report*.

5 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 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**, **ML**, **value** , etc.; They allow for the annotation of document-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 follows¹ :

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

¹The *technical* organization is slightly different and shown in Section 5.5.

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[i/label/ititle]...`, `text[i/label/isection]...`, 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 L^AT_EX and finally PDF. The mapping of an ontology to a specific representation in L^AT_EX is steered via associated L^AT_EX 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 5.5 in more detail.

5.1 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 syntactically equivalent,
- an optional document base class expressing single inheritance class extensions, restricting the class-hierarchy basically to a tree,
- *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.

doc_class'es and **onto_class**'es respectively, have a semantics, i. e., a logical representation as extensible records in HOL ([23], pp. 11.6); there are therefore amenable to logical reasoning.

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 *term 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.

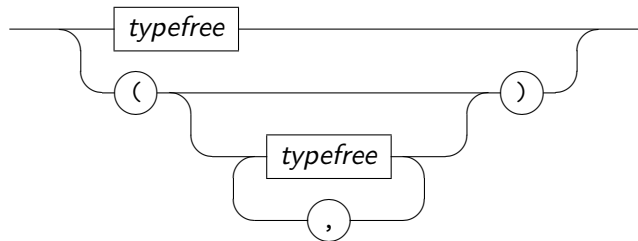
A major design decision of ODL is to denote attribute values by HOL-terms and HOL-types. Consequently, ODL can refer to any predefined type defined in the HOL library, e. g., *string* or *int* as well as parameterized types, e. g., *_ option*, *_ list*, *_ set*, or products *_ » _*. As a consequence 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.

5.1.1 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:

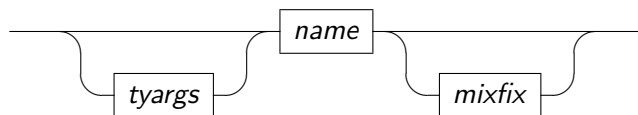
5 Ontologies and their Development

- *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*:



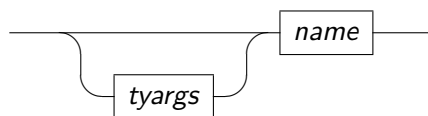
typefree denotes fixed type variable (*a*, *b*, ...) (see [23])

- *dt_name*:



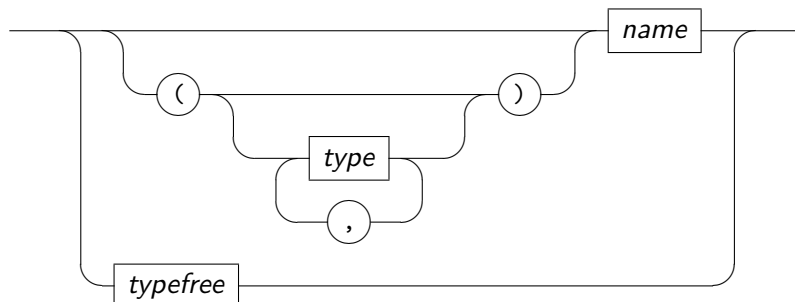
The syntactic entity *name* denotes an identifier, *mixfix* denotes the usual parenthesized mixfix notation (see [23]). The *name*'s referred here are type names such as *int*, *string*, *list*, *set*, etc.

- *type_spec*:

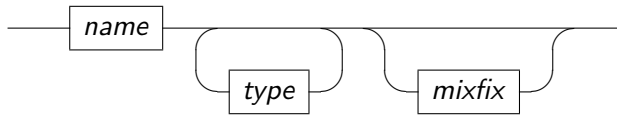


The *name*'s referred here are type names such as *int*, *string*, *list*, *set*, etc.

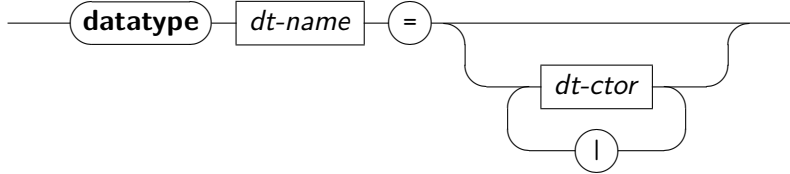
- *type*:



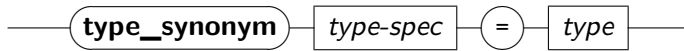
- *dt_ctor*:



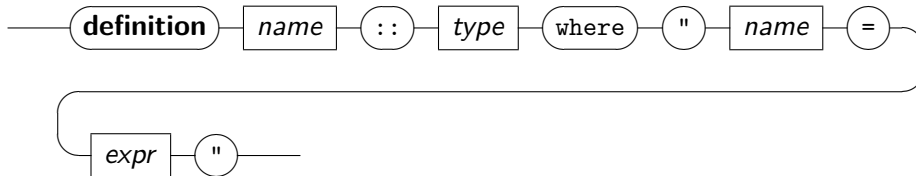
- *datatype_specification*:



- *type_synonym_specification*:



- *constant_definition* :



- *expr*: the syntactic category *expr* here denotes the very rich language of mathematical notations encoded in -terms in Isabelle/HOL. Example expressions are: $1+2$ (arithmetic), $[1,2,3]$ (lists), $ab\ c$ (strings), $\{1,2,3\}$ (sets), $(1,2,3)$ (tuples), $x. P(x) \ Q\ x = C$ (formulas). For comprehensive overview, see [17].

Advanced ontologies can, e.g., use recursive function definitions with pattern-matching [14], 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_classes**, 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.

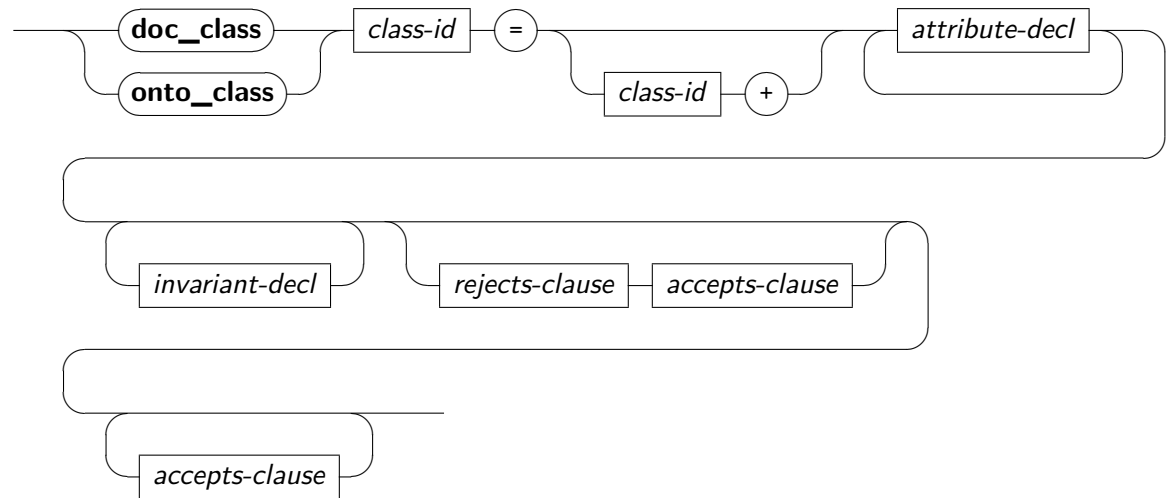
5.1.2 Defining Document Classes

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

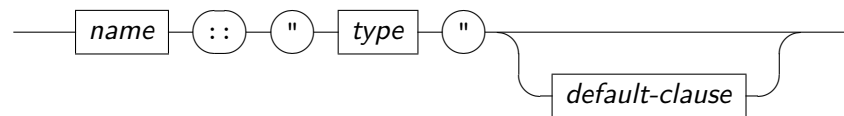
- *class_id*: a type-name that has been introduced via a *doc_class_specification*.

5 Ontologies and their Development

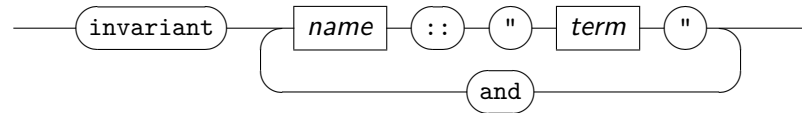
- *doc_class_specification*: We call document classes with an *accepts_clause* monitor classes or *monitors* for short.



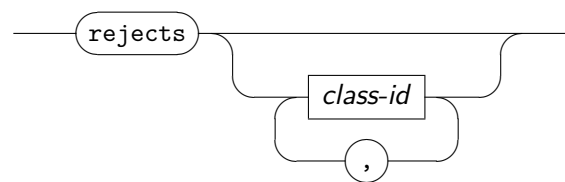
- *attribute_decl*:



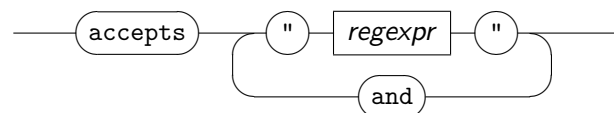
- *invariant_decl*: Invariants can be specified as predicates over document classes represented as records in HOL. Sufficient type information must be provided in order to disambiguate the argument of the expression and the symbol is reserved to reference the instance of the class itself.



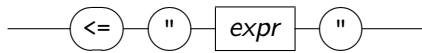
- *rejects_clause*:



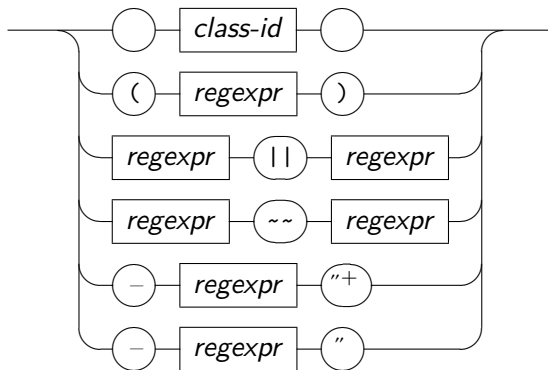
- *accepts_clause*:



- *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:

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

\LaTeX

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{isamarkup}...\end{isamarkup}`-environment, to ensure the correct

context within Isabelle's \LaTeX -setup. Moreover, Isabelle/DOF also provides the following two variants of `\newisadof{}[]{}{}`:

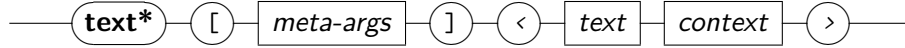
- `\renewisadof{}[]{}{}` for re-defining (over-writing) an already defined command, and
- `\provideisadof{}[]{}{}` for providing a definition if it is not yet defined.

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 \LaTeX -style for the ontology *Isabelle_DOF.scholarly_paper* shows examples of protecting special characters in definitions that need to make use of a entries in an aux-file.

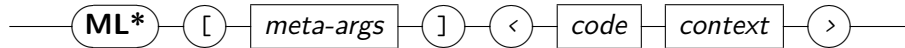
5.2 The main Ontology-aware Document Elements

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 texts, code, and terms. As mentioned already in the introduction, this boils down two three major groups of commands used to annotate text-, code-, and term contexts with instances of ontological classes, i. e., meta-information specified in an ontology. Representatives of these three groups, which refer by name to equivalent standard Isabelle commands by their name suffixed with a `_`, are presented as follows in a railroad diagram:

- *annotated_text_element* :



- *annotated_code_element* :



- *annotated_term_element* :

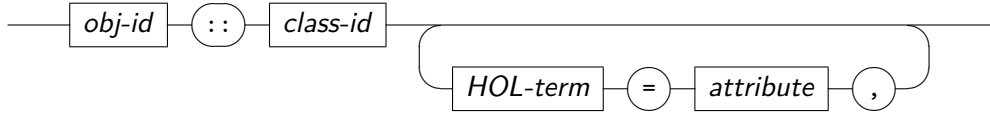


In the following, we will formally introduce the syntax of the core commands as supported on the Isabelle/Isar level. On this basis, concepts such as the freeform **Definition** and **Lemma** elements were derived from **text**. Similarly, the corresponding formal **definition** and **lemma** elements were built on top of functionality of the **value**-family.

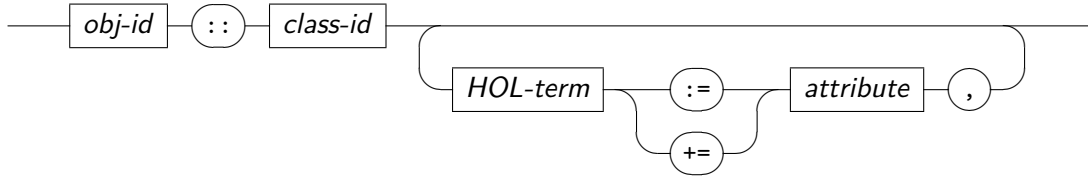
5.2.1 General Syntactic Elements for Document Management

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

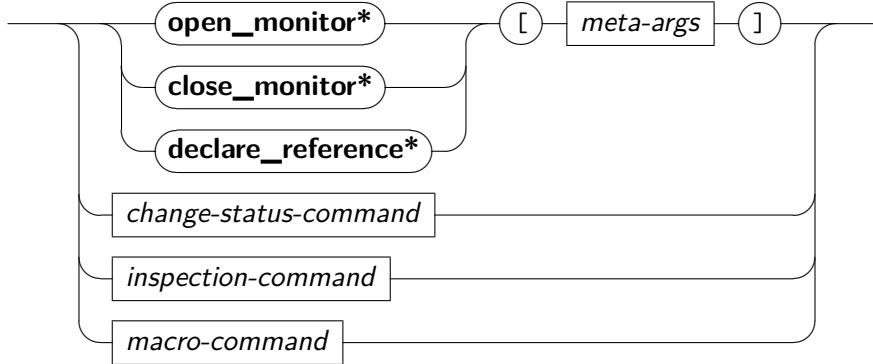
- *obj_id*: (or *oid* for short) a *name* as specified in Section 5.1.1.
- *meta_args* :



- *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* :



With respect to the family of text elements, `text[oidcid, ...] < 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 omitted, 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>}`. The choice of the default class in a `declare_reference`-command can be influenced by setting globally an attribute:

```
declare[[declare_reference_default_class = definition]]
```

Isar

Then in this example:

```
declare_reference[def1]
```

Isar

def1 will be a declared instance of the class *definition*.

5.2.2 Ontological Code-Contexts and their Management

- *annotated_code_element*:

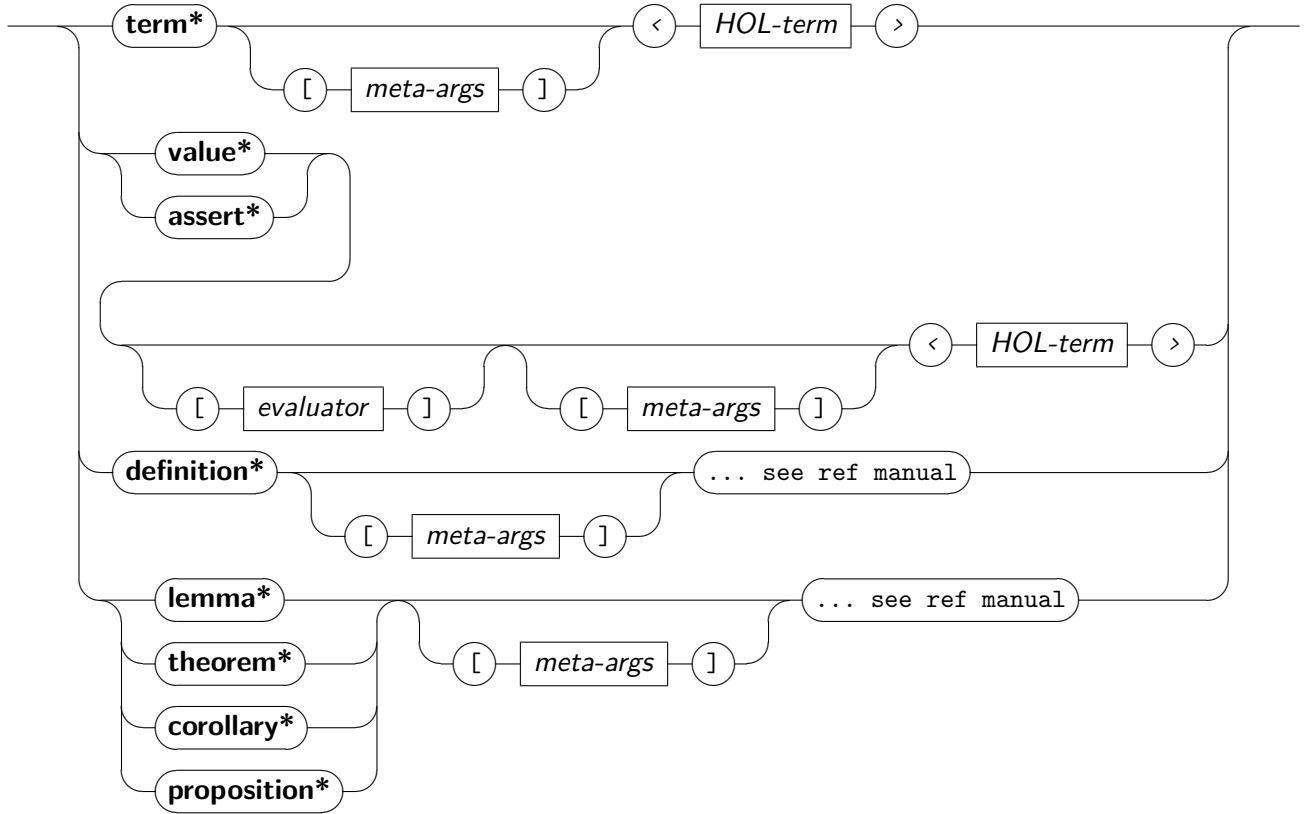


The `ML[oidcid, ...] < SMLcode >`-document elements proceed similarly: a referentiable meta-object of class *cid* is created, initialized with the optional attributes and bound to *oid*. In fact, the entire the meta-argument list is optional. The SML-code is type-checked and executed in the context of the SML toplevel of the Isabelle system as in the corresponding `ML< SMLcode >`-command. Additionally, ML antiquotations were added to check and evaluate terms with term antiquotations:

- $\text{@}\{term_ \langle term \rangle\}$ parses and type-checks *term* with term antiquotations, for instance $\text{@}\{term_ \langle \text{@-technical } \langle odmanual1 \rangle \rangle\}$ will parse and check that *odmanual1* is indeed an instance of the class *M_04_Document_Ontology.technical*,
- $\text{@}\{value_ \langle term \rangle\}$ performs the evaluation of *term* with term antiquotations, for instance $\text{@}\{value_ \langle definition_list \text{@-technical } \langle odmanual1 \rangle \rangle\}$ will get the value of the *definition_list* attribute of the instance *odmanual1*. *value_* may have an optional argument between square brackets to specify the evaluator: $\text{@}\{value_ [nbe] \langle definition_list \text{@-technical } \langle odmanual1 \rangle \rangle\}$ forces *value_* to evaluate the term using normalization by evaluation (see [23]).

5.2.3 Ontological Term-Contexts and their Management

- *annotated_term_element*



For a declared class *cid*, there exists a term-antiquotation of the form $\text{@}\{cid \langle oid \rangle\}$. The major commands providing term-contexts are²

- **term**[*oidcid*, ...] $\langle \text{HOLterm} \rangle$,

²The meta-argument list is optional.

- **value**[*oidcid*, ...] $\langle \text{HOLterm} \rangle$, and
- **assert**[*oidcid*, ...] $\langle \text{HOLterm} \rangle$
- **definition**[*oidcid*, ...] *const_name* **where** $\langle \text{HOLterm} \rangle$, and
- **lemma**[*oidcid*, ...] *name* $\langle \text{HOLterm} \rangle$.

Wrt. creation, checking and traceability, these commands are analogous to the ontological text and code-commands. However the argument terms may contain term-antiquotations stemming from an ontology definition. Term-contexts were type-checked and *validated* against the global context (so: in the term `@-scholarly_paper.author <bu>`, *bu* is indeed a string which refers to a meta-object belonging to the document class *M_05_Proofs_Ontologies.author*, for example). With the exception of the **term**-command, the term-antiquotations in the other term-contexts are additionally expanded (e.g. replaced) by the instance of the class, e.g., the HOL-term denoting this 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 5.3.1). This is particularly useful to explore formal definitions wrt. their border cases. For **assert**, the evaluation of the term can be disabled with the *disable_assert_evaluation* theory attribute:

```
declare[[disable_assert_evaluation]]
```

Isar

Then **assert** will act like **term**.

The **definition**-command allows *prop*, *spec_prem*s, and *for_fixes* (see the **definition** command in [23]) to contain term-antiquotations. For example:

```
doc_class A =
  level int option
  x int
definition[a1A, x=5, level=Some 1] xx where xx A.x @-A <a1>'' if A.x @-A <a1>'' =
5
```

Isar

The `@-A <a1>''` term-antiquotation is used both in *prop* and in *spec_prem*s.

lemma, **theorem**, etc., are extended versions of the goal commands defined in [23]. Term-antiquotations can be used either in a *long_statement* or in a *short_statement*. For instance:

```
lemma[e5E] testtest xx + A.x @-A <a1>'' = yy + A.x @-A <a1>'' xx = yy
by simp
```

Isar

This **lemma**-command is defined using the `@-A <a1>''` term-antiquotation and attaches the *e5* instance meta-data to the *testtest*-lemma.

```

doc_class cc_assumption_test =
  a int
  text[cc_assumption_test_ref cc_assumption_test]

definition tag_l a b b where tag_l x y. y

lemma tagged tag_l @-ccassumptiontest <cc_assumption_test_ref> AA AA
  by (simp add tag_l_def)

find_theorems nametagged (_cc_assumption_test _ _) _ _ _

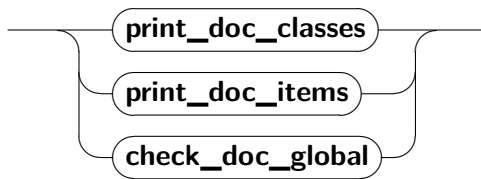
```

In this example, the definition `tag_l` adds a tag to the `tagged` lemma using the term-antiquotation `@-ccassumptiontest <cc_assumption_test_ref>` inside the `prop` declaration.

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.

5.2.4 Status and Query Commands

- Isabelle/DOF `inspection_command` :



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. The theory attribute `object_value_debug` allows to inspect the term of instances value before its elaboration and normalization. Adding:

```

declare[[object_value_debug = true]]

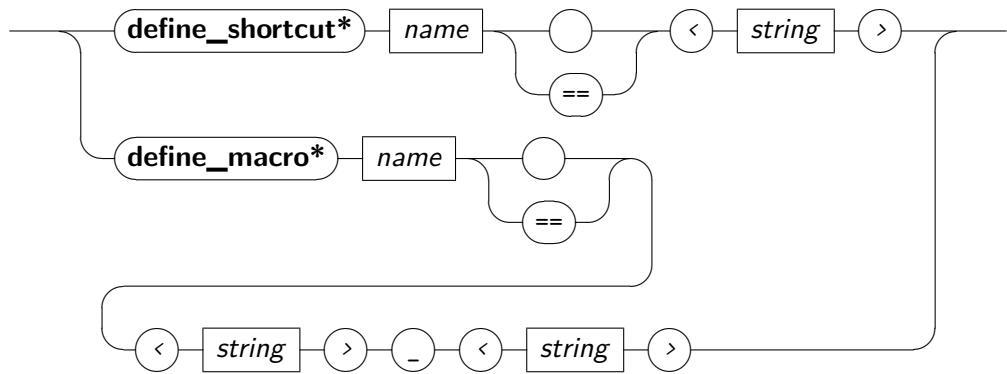
```

... to the theory will add the raw value term to the instance object-table for all the subsequent declared instances (using **text** for instance). The raw term will be available in the *input_term* field of **print_doc_items** output and,

- **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 5.4)

5.2.5 Macros

- Isabelle/DOF *macro_command* :



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) ‹
```

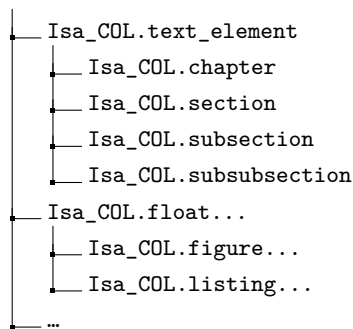
where `check_latex_measure` is a hand-programmed function that checks the input for syntactical and static semantic constraints.

5.3 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) or `technical_report` (for MINT-oriented technical reports).

5.3.1 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:



In particular it defines the super-class `M_04_Document_Ontology.text_element`: the root of all text-elements:

```
doc_class text_element =
  level      int option  j = None
  referentiable bool j = False
  variants    String.literal set j = –STR outline, STR document
```

Isar

As mentioned in Section 3.3.2, *A.level* defines the section-level (e.g., using a \LaTeX -inspired hierarchy: from *Some 1* (corresponding to `\part`) to *Some 0* (corresponding to

³contained in `Isabelle_DOF.Isa_COL`

`\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 introduced by a section element).

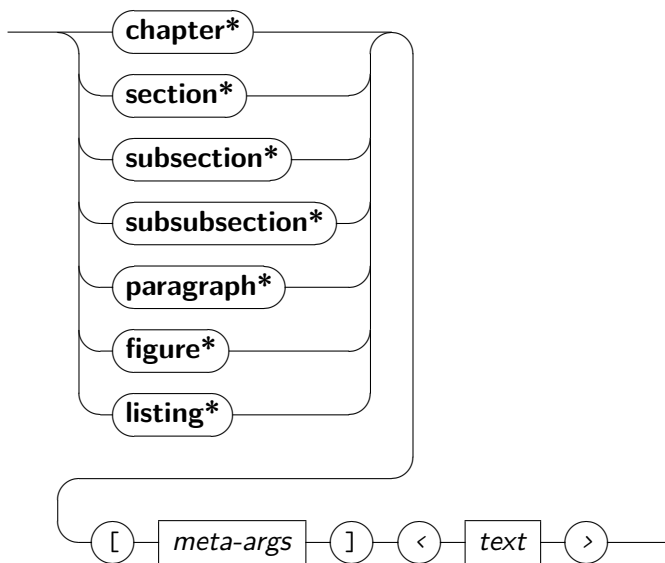
The attribute *referentiable* captures the information if a text-element can be a target for a reference, which is the case for sections or subsections, for example, but not arbitrary elements such as, i. e., paragraphs (this mirrors restrictions of the target \LaTeX representation). The attribute *variants* 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 *float*'s, please consult the ontology in *Isabelle_DOF.isa_COL* directly and consult the Example I and II for their pragmatics. The *Isabelle_DOF.isa_COL* also provides the subclasses *figure* and *listing* which together with specific text-antiquotations like:

1. `@-theory_text [options] path` (Isabelle)
2. `@-fig_content (width=, height=, caption=) path` (COL)
3. `@-boxed_theory_text [display] { ... }` (local, e.g. manual)
4. `@-boxed_sml [display] { ... }` (local, e.g. manual)
5. `@-boxed_pdf [display] { ... }` (local, e.g. manual)
6. `@-boxed_latex [display] { ... }` (local, e.g. manual)
7. `@-boxed_bash [display] { ... }` (local, e.g. manual)

COL finally provides macros that extend the command-language of the DOF core by the following abbreviations:

- *derived_text_element* :

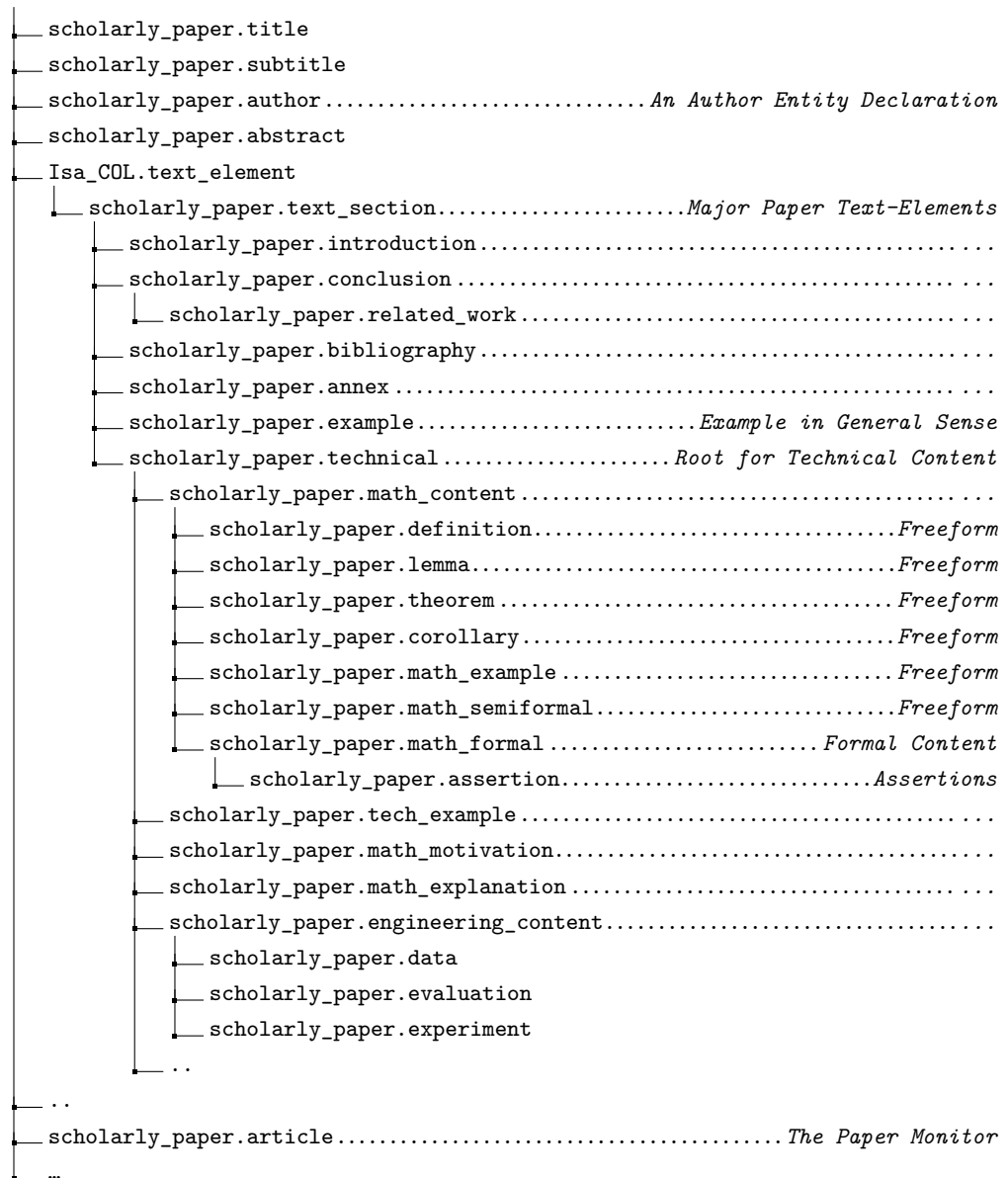


The command syntax follows the implicit convention to add a “*” to distinguish them from the (similar) standard Isabelle text-commands which are not ontology-aware.

5.3.2 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:



Recall that *Formal Content* means *machine-checked, validated content*.

A pivotal abstract class in the hierarchy is:

```

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

```

Isar

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 *M_04_Document_Ontology.text_element* to a specific *M_05_Proofs_Ontologies.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

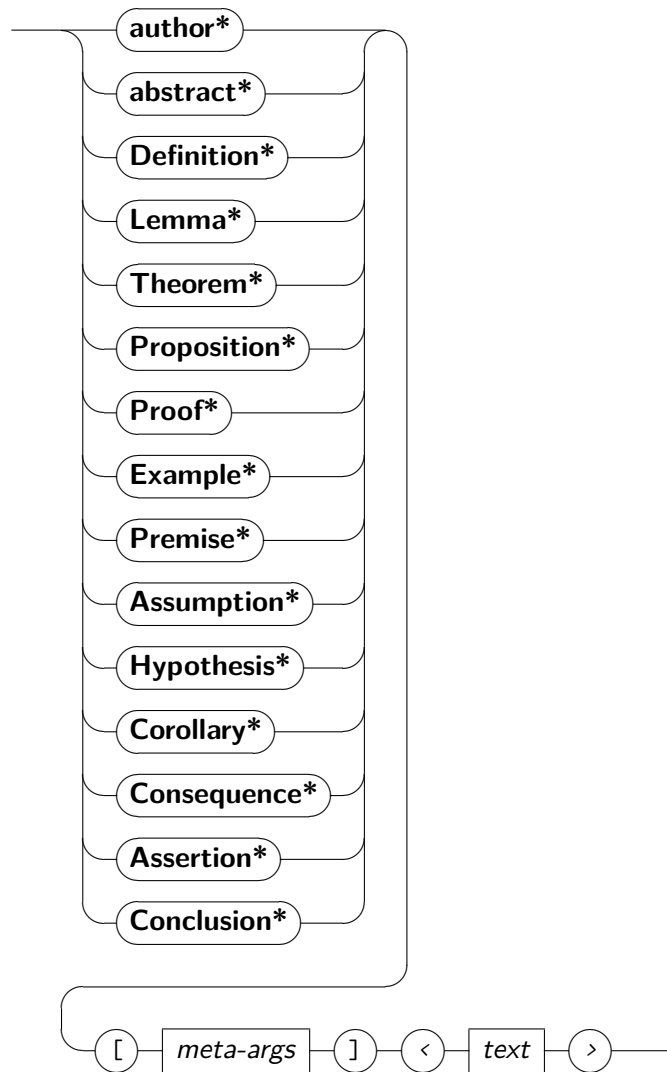
```
text[example, main_author = Some(@-author <bu>")] < >
```

Isar

where *bu* is a string presentation of the reference to the author text element (see below in Section 5.3.1).

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 *M_04_Document_Ontology.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.:

```
title[titletitle](Isabelle/DOF)
subtitle[subtitlesubtitle](User and Implementation Manual)
author[adbauthor, email=(a.brucker@exeter.ac.uk),
        orcid=(0000000263551200), http_site=(https://brucker.ch/),
        affiliation=(University of Exeter, Exeter, UK)] (Achim D. Brucker)
author[buauthor, email = (wolff@lri.fr),
        affiliation = (Université ParisSaclay, LRI, Paris, France)] (Burkhart Wolff)
```

Isar

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

```
assert[ass1assertion, short_name = (This is an assertion)] (last [3] i (4int))
```

Isar

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

```
text[claimassertion](For nonempty lists, our definition yields indeed
                     the last element of a list.)
assert[claim1assertion] (last[4int] = 4)
assert[claim2assertion] (last[1,2,3,4int] = 4)
```

Isar

As mentioned before, the command macros of **Definition**, **Lemma** and **Theorem** set the default class to the super-class of *M_04_Document_Ontology.definition*. However, in order to avoid the somewhat tedious consequence:

```
Theorem[T1theorem, short_name=(DF definition captures deadlockfreeness)] ( )
```

Isar

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

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

Isar

which allows the above example be shortened to:

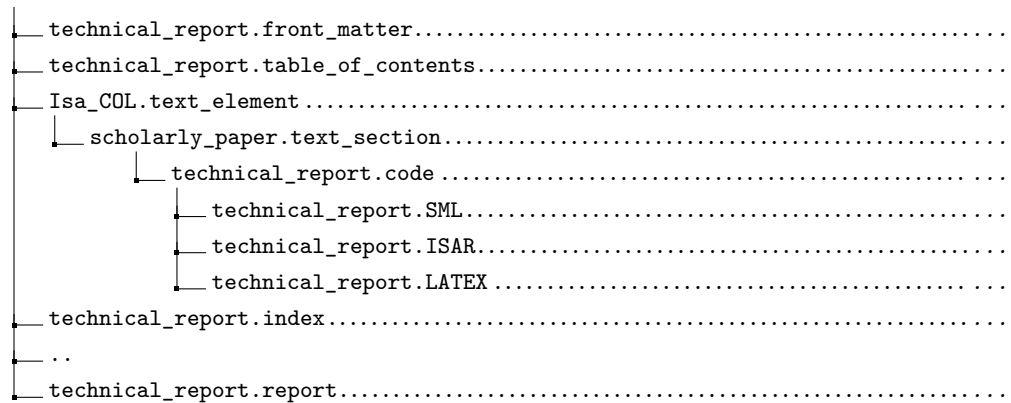
```
Theorem[T1, short_name=(DF definition captures deadlockfreeness)] ( )
```

Isar

5.3.3 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,



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:

```

theory
...
imports
...
keywords
  section
begin
...
end

```

Isar

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:

```

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} )));

```

SML

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:

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

 \LaTeX

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%
}
```

 \LaTeX

5.4 Advanced ODL Concepts

5.4.1 Example

We assume in this section the following local ontology:

```

doc_class title =
  short_title string option i= None
doc_class author =
  email string i=
datatype classification = SIL0 SIL1 SIL2 SIL3 SIL4
doc_class abstract =
  keywordlist string list i= []
  safety_level classification i= SIL3
doc_class text_section =
  authored_by author set i= -"
  level int option i= None
type_synonym notion = string
doc_class introduction = text_section +
  authored_by author set i= 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 i= []
datatype kind = expert_opinion argument proof
doc_class result = technical +
  evidence kind
  property thm list i= []
doc_class example = technical +
  referring_to (notion + definition) set i= -"
doc_class conclusion = text_section +
  establish (claim » result) set

```

5.4.2 Meta-types as Types

To express the dependencies between text elements to the formal entities, e. g., `term` (-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 istringi}`. 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. 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 where this theorem is established. Similarly, attribute values like `property = @-term ⟨A B⟩` require that the HOL-string `A B` is again type-checked and represents indeed a formula in `.`. Another instance

of this process, which we call *second-level type-checking*, are term-constants generated from the ontology such as `@{definition ;string}`.

5.4.3 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 are checked when an instance of the class is defined, and trigger warnings. The checking is enabled by default but can be disabled with the *invariants_checking* theory attribute:

```
declare[[invariants_checking = false]]
```

Isar

To enable the strict checking of the invariants, that is to trigger errors instead of warnings, the *invariants_strict_checking* theory attribute must be set:

```
declare[[invariants_strict_checking = true]]
```

Isar

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

```
doc_class class_inv1 =
  int1 int
  invariant inv1 int1 3

doc_class class_inv2 = class_inv1 +
  int2 int
  invariant inv2 int2 ; 2
```

Isar

The `int1` 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* 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:

```
text[testinv1class_inv1, int1=4]⟨lorem ipsum...⟩
text[testinv2class_inv2, int1=3, int2=1]⟨lorem ipsum...⟩
```

Isar

The value of each attribute defined for the instances is checked against their classes invariants. As the class *class_inv2* is a subclass 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 5.4.1. 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:

```

doc_class introduction = text_section +
  authored_by author set i= UNIV
  uses notion set
  invariant author_finite finite (authored_by )
doc_class result = technical +
  evidence kind
  property thm list i= []
  invariant has_property evidence = proof property []
doc_class example = technical +
  referring_to (notion + definition) set i= -"
doc_class conclusion = text_section +
  establish (claim » result) set
  invariant total_rel x. x Domain (establish )
    ( y Range (establish ). (x, y) establish )

```

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. The invariants *author_finite* and *establish_defined* can not be checked directly and need a little help. We can set the *invariants_checking_with_tactics* theory attribute to help the checking. It will enable a basic tactic, using unfold and auto:

```

declare[[invariants_checking_with_tactics = true]]

```

There are still some limitations with this high-level syntax. For now, the high-level syntax does not support the checking of specific monitor behaviors (see Section 5.4.5). 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 5.4.4). Also, for now, term-antiquotations can not be used in an invariant formula.

5.4.4 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 5.4.3, 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 =
      ISA_core.compute_attr_access ctxt "evidence" oid NONE @{here}
    val prop =
      ISA_core.compute_attr_access ctxt "property" oid NONE @{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")
    | _ => true
  end
val cid_long = DOF_core.get_onto_class_name_global "result" @{theory}
val bind = Binding.name "Check_Result"
val _ = Theory.setup (DOF_core.make_ml_invariant (check_result_inv, cid_long)
  |> DOF_core.add_ml_invariant bind)

```

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 `value_<evidence @-result <oid>`, since `oid` is bound to a variable here and can therefore not be statically expanded.

5.4.5 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      i= LNCS
  version (int » int » int) i= (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[thisarticle]
...
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 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 .

Should the specified ranges of admissible instances not be observed, warnings will be triggered. To forbid the violation of the specified ranges, one can enable the `strict_monitor_checking` theory attribute:

```
declare[[strict_monitor_checking = true]]
```

Isar

It is possible to enable the tracing of free classes occurring inside the scope of a monitor by enabling the `free_class_in_monitor_checking` theory attribute:

```
declare[[free_class_in_monitor_checking = true]]
```

Isar

Then a warning will be triggered when defining an instance of a free class inside the scope of a monitor. To forbid free classes inside the scope of a monitor, one can enable the `free_class_in_monitor_strict_checking` theory attribute:

```
declare[[free_class_in_monitor_strict_checking = true]]
```

Isar

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 does not support the checking of specific monitor behaviors like the one just described and you must use the low-level class invariants (see Section 5.4.4).

Low-level invariants checking can be set up to be triggered when opening a monitor, when closing a monitor, or both by using the `DOF_core.add_opening_ml_invariant`, `DOF_core.add_closing_ml_invariant`, or `DOF_core.add_ml_invariant` commands respectively, to add the invariants to the theory context (See Section 5.4.4 for an example).

5.4.6 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) @-instances_of ⟨result⟩⟩
value⟨map (text_section.authored_by) @-instances_of ⟨introduction⟩⟩
```

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 (λ. result.evidence = proof) @-instances_of ⟨result⟩⟩
```

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

```
value⟨filter (λ. the (text_section.level) > 1) @-instances_of ⟨introduction⟩⟩
```

5.5 Technical Infrastructure

5.5.1 The Previewer

A screenshot of the editing environment is shown in Figure 5.1. It supports incremental continuous PDF generation which improves usability. Currently, the granularity is restricted to entire theories (which have to be selected in a specific document pane). The response times can not (yet) compete with a Word- or Overleaf editor, though, which is mostly due to the checking and evaluation overhead (the turnaround of this section is about 30 s). However, we believe that better parallelization and evaluation techniques will decrease this gap substantially for the most common cases in future versions.

5.5.2 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.

5 Ontologies and their Development

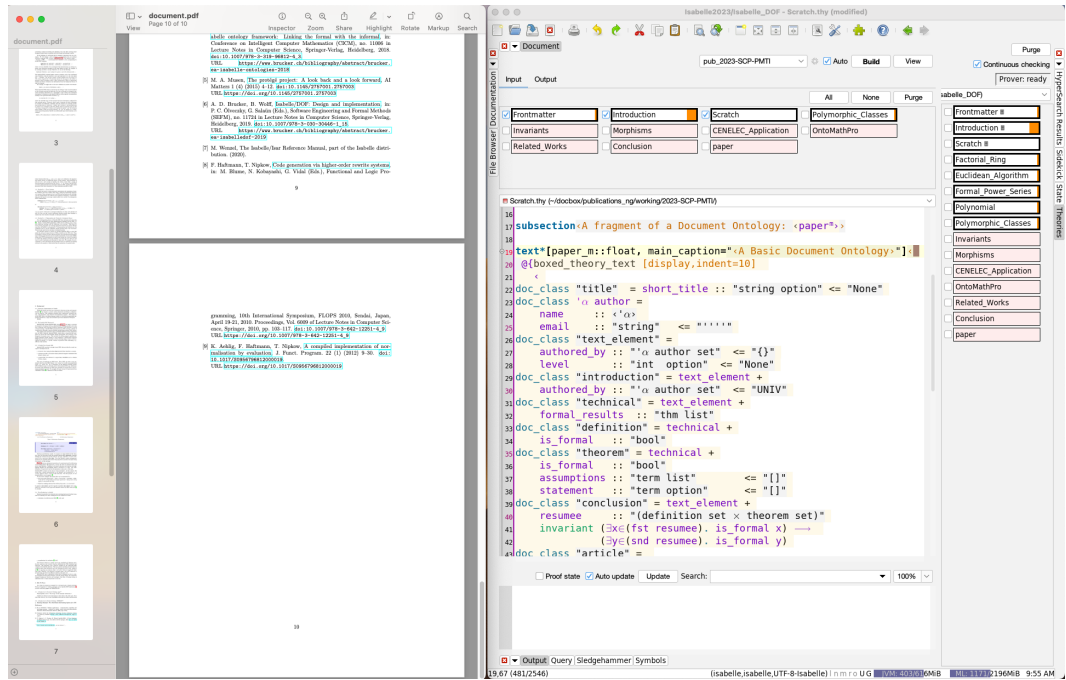


Figure 5.1: A Screenshot while editing this Paper in DOF with Preview.

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 ontologies and consist of an Isabelle theory file and a \LaTeX -style file:

```

├── 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

```

Developing a new ontology “foo” requires the following steps:

- definition of the ontological concepts, using Isabelle/DOF’s Ontology Definition Language (ODL), in a new theory file ontologies/foo/foo.thy.

- definition of the document representation for the ontological concepts in a \LaTeX -style stored in the same directory as the theory file containing the ODL definitions. The file name should start with the prefix “DOF-”. For instance: DOF-foo.sty
- registration of the \LaTeX -style by adding a suitable **define_ontology** command to the theory containing the ODL definitions.

5.5.3 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`:

```

└─ 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`
- add a suitable **define_template** command to a theory that is imported by the project that shall use the new document template.

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.

5.6 Defining Document Templates

5.6.1 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 `preamble.tex`. The file name of the new template should start with the prefix `root-` and need to be registered using the **define_template** command. a typical Isabelle/DOF document template looks as follows:

```

1 \documentclass{article}    % The LaTeX-class of your template
2 \usepackage{DOF-core}
3 \usepackage{subcaption}
4 \usepackage[size=footnotesize]{caption}
5 \usepackage{hyperref}
6
7 %% Main document, do not modify
8 \begin{document}
9 \maketitle
10 \IfFileExists{dof_session.tex}{\input{dof_session}}{\input{session}}
11 \IfFileExists{root.bib}{\bibliography{root}}{\}
12 \end{document}

```

5.6.2 Tips, Tricks, and Known Limitations

In this section, we will discuss several tips and tricks for developing new or adapting existing document templates or L^AT_EX-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 L^AT_EX-setup. In this directory, you can directly use L^AT_EX on the main file, called `root.tex`:

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

Bash

This allows you to develop and check your L^AT_EX-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, L^AT_EX cuts off 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 L^AT_EX in such a way that it prints long error or warning messages. This can easily be done for individual L^AT_EX invocations:

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

Bash

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. Examples of this can be found, e.g., in the ontology-style `../../ontologies/scholarly_paper/DOF-scholarly_paper.sty`. For details about the expansion mechanism in general, we refer the reader to the \LaTeX literature (e.g., [9, 13, 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 `../../ontologies/scholarly_paper/DOF-scholarly_paper.sty` contains an example that, firstly, shows how to write the author and affiliation information into the auxiliary file for re-use in the next \LaTeX -run and, secondly, shows how to collect the author and affiliation information into an `\author` and a `\institution` statement, each of which containing the information for all authors. The collection of the author information is provided by the following \LaTeX -code:

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

\LaTeX

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 (L^AT_EX-classes), authors of ontology representations might restrict their use to specific classes. This can, e.g., be done using the `\ifclassloaded{}` command:



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

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 L^AT_EX support provides support for the `scrrept`-class which is not supported by the L^AT_EX support for *scholarly_paper*.

6 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 to the possibility to modify Isabelle itself,
- Isabelle/DOF has been organized as an AFP entry and a form of an Isabelle component that is compatible with this goal,
- 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.

6.1 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 `Theory_Data`, and the following fairly typical code-fragment is drawn from Isabelle/DOF:

```
structure Onto_Classes = Theory_Data
(
  type T = onto_class Name_Space.table;
  val empty : T = Name_Space.empty_table onto_classN;
  fun merge data : T = Name_Space.merge_tables data;
);
```

SML

where the table `Name_Space.table` manages the environment for class definitions (`onto_class`), inducing the inheritance relation, using a `Name_Space` table. Other tables capture, e. g., the class instances, class invariants, inner-syntax antiquotations. Operations

follow the MVC-pattern, where Isabelle/Isar provides the controller part. A typical model operation has the type:

```
val opn :: <args_type> -> theory -> theory
```

SML

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

```
fun add_onto_class name onto_class thy =
  thy |> Onto_Classes.map
    (Name_Space.define (Context.Theory thy) true (name, onto_class) #> #2);
```

SML

This code fragment uses operations from the library structure `Name_Space` that were used to update the appropriate table for document objects in the plugin-local state. A name space manages a collection of long names, together with a mapping between partially qualified external names and fully qualified internal names (in both directions). It can also keep track of the declarations and updates position of objects, and then allows a simple markup-generation. Possible exceptions to the update operation are automatically triggered.

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

```
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
```

SML

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:

```
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.$$$ "]"
```

SML

The “model” `create_and_check_docitem` and “new” `ODL_Meta_Args.Parser.attributes` parts were combined via the piping operator

and registered in the Isar toplevel:

```
val _ =
  let fun create_and_check_docitem (((oid, pos), cid_pos), doc_attrs)
      = (Value_Command.Docitem_Parser.create_and_check_docitem
         {is_monitor = false} {is_inline=true}
         {define = false} oid pos (cid_pos) (doc_attrs))
  in Outer_Syntax.command @{command_keyword "declare_reference*"}
    "declare_document_reference"
    (ODL_Meta_Args_Parser.attributes
     >> (Toplevel.theory o create_and_check_docitem))
  end;
```

SML

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

```
declare_reference [!a!requirement, alpha=main, beta=42]
```

Isar

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.

6.2 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

```
val _ = Theory.setup
  (docitem_antiquotation @{binding "docitem"} DOF_core.default_cid #>

  ML_Antiquotation.inline @{binding "docitem_value"}
    ML_antiquotation_docitem_value)
```

SML

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

```
text⟨as defined in @-docitem ⟨d1⟩" ...⟩
```

Isar

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 `M_06_RefMan.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.

6.3 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 term-annotation (ISA’s), a function of type

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

SML

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**.

6.4 Programming Class Invariants

See Section 5.4.4.

6.5 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:

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

SML

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 5.4.5). 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 suffix 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].

6.6 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:

```
\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}%
```

 \LaTeX

The \LaTeX -generator of Isabelle/DOF maps each *doc_item* to an \LaTeX -environment (recall Section 5.3.2). As generic *doc_items* are derived from the text element, the environment `isamarkuptext*` builds the core of Isabelle/DOF’s \LaTeX implementation.

Bibliography

- [1] Y. A. Ameur, F. Besnard, P. Girard, G. Pierra, and J. Potier. Formal specification and metaprogramming in the EXPRESS language. In *The 7th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 181–188. Knowledge Systems Institute, 1995.
- [2] 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.
- [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] Eclipse Foundation. ATL – a model transformation technology. URL <https://www.eclipse.org/at1/>.
- [9] V. Eijkhout. *The Computer Science of TeX and LaTeX*. Texas Advanced Computing Center, 2012.
- [10] J. Euzenat and P. Shvaiko. *Ontology Matching, Second Edition*. Springer, 2013. ISBN 978-3-642-38720-3. doi: 10.1007/978-3-642-38721-0.

Bibliography

- [11] 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.
- [12] IBM. IBM engineering requirements management DOORS family, 2019. <https://www.ibm.com/us-en/marketplace/requirements-management>.
- [13] D. E. Knuth. *The TeXbook*. Addison-Wesley Professional, 1986. ISBN 0201134470.
- [14] A. Kraus. Defining recursive functions in isabelle/hol, 2020. <https://isabelle.in.tum.de/doc/functions.pdf>.
- [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. <https://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*, 2022. Part of the Isabelle distribution.
- [24] M. Wenzel and B. Wolff. Building formal method tools in the Isabelle/Isar framework. In K. Schneider and J. Brandt, editors, *TPHOLs 2007*, number 4732 in *LNCS*, pages 352–367. Springer, 2007. doi: 10.1007/978-3-540-74591-4_26.