Isabelle/DOF: Design and Implementation

Achim D. Brucker[1(✉)] and Burkhart Wolff[2]

[1] Department of Computer Science, University of Exeter, Exeter, UK
a.brucker@exeter.ac.uk
[2] LRI, CNRS, Université Paris-Saclay, Paris, France
wolff@lri.fr
http://www.brucker.ch/, http://www.lri.fr/~wolff

Abstract. DOF is a novel framework for *defining* ontologies and *enforcing* them during document development and document evolution. A major goal 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.

To support a consistent development of formal and informal parts of a document, we provide Isabelle/DOF, an implementation of DOF on top of Isabelle/HOL. 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.

In this paper, we give an in-depth presentation of the design concepts of DOF's Ontology Definition Language (ODL) and key aspects of the technology of its implementation. Isabelle/DOF is the first ontology language supporting machine-checked links between the formal and informal parts in an LCF-style interactive theorem proving environment.

Sufficiently annotated, large documents can easily be developed collaboratively, while *ensuring their consistency*, and the impact of changes (in the formal and the semi-formal content) is tracked automatically.

Keywords: Ontology · Formal document development · Certification · DOF · Isabelle/DOF

1 Introduction

With the maturation and growing power of interactive proof systems, the body of formalized mathematics and engineering is dramatically increasing. The Isabelle Archive of Formal Proof (AFP) [6], created in 2004, counted in 2015 a total of 215 articles, whereas the count stood at 413 only three years later. An in-depth empirical analysis shows that both complexity and size increased accordingly [11]. Together with the AFP, there is also a growing body on articles concerned with formal software engineering issues such as standardized language definitions (e. g., [15,21]), data-structures (e. g., [14,24]), hardware-models (e. g., [20]), security-related specifications (e. g., [13,26]), or operating systems (e. g., [22,27]).

P. C. Ölveczky and G. Salaün (Eds.): SEFM 2019, LNCS 11724, pp. 275–292, 2019.
https://doi.org/10.1007/978-3-030-30446-1_15

This development raises interest in at least two ways: First, there is a substantial potential of *retrieve* and *reuse* of formal developments, and second, formal techniques allow a deeper checking of documents containing formal specifications, proofs and tests. This paves the way for collaborative, continuously machine-checked developments of certification documents involving both formal as well of informal content evolution.

We are focusing in this paper on the latter aspect. Certification documents have to follow a structure which is relatively strictly defined in certification standards such as [16,17]. In practice, large groups of developers have to produce a substantial set of documents where the consistency is notoriously difficult to maintain. In particular, certifications are centered around the *traceability* of requirements throughout the entire set of documents. While technical solutions for the traceability problem exists (most notably: DOORS [7]), they are weak in the treatment of formal entities (such as formulas and their logical contexts).

Enforcing a document structure is done by *annotations* with meta-information; the language in which the latter is defined is widely called a *document ontology* (an equivalent term is *vocabulary*) in the semantic web community [3], i. e., a machine-readable form of the structure of a document and the document discourse. Let us consider a set of *text elements* available in a given corpus. These elements may be sentences or paragraphs, figures, tables, definitions or lemmas, code, and, for example, the results of test-executions. By annotation, we make links explicit that may exist between an ontology concept and a document element of the considered corpus. While ontologies as such can be used for a variety of applications, this paper is concerned with the representation of a mixture formal and semi-formal content (as it is, e. g., very common in documents within a software development process). Therefore, we also discuss the mapping to a concrete target document format (e. g., PDF) that, e. g., might be used within a traditional certification process.

In this paper, we present the concepts of our Document Ontology Framework (DOF) designed for building scalable and user-friendly tools on top of interactive theorem provers, and an implementation of DOF called Isabelle/DOF. Isabelle/DOF supports both defining ontologies and documents that conform to one or more ontologies. An example-driven introduction into Isabelle/DOF also presenting details of the user-interaction in the IDE can be found elsewhere [12]. In this paper, we are focusing on the fundamental concepts of its ontology definition language ODL and the more technical issues of its implementation. In particular, we present novel concepts such as *meta-types-as-types*, *class-invariants*, *monitors*, *inner-syntax antiquotations* as well as their interaction.

The rest of the paper is structured as follows: after explicating the underlying assumptions in a generic document model, we present the design of DOF as a language in Sect. 3. It follows a presentation of the implementation of Isabelle/DOF (Sect. 4) and a discussion on related and future work (Sect. 5).

2 Background: The Document Model

In this section, we introduce the assumed document model underlying DOF in general; in particular the concepts *integrated document*, *sub-document*, *text-*

element and *semantic macros* occurring inside text-elements. Furthermore, we assume two different levels of parsers (for *outer* and *inner syntax*) where the inner-syntax is basically a typed λ-calculus and some Higher-order Logic (HOL).

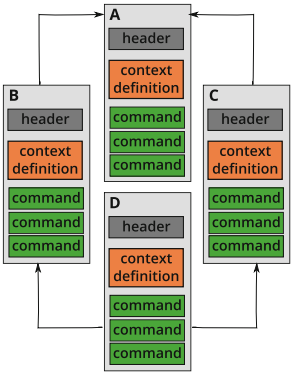

Fig. 1. A theory-graph in the document model.

We assume a hierarchical document model, i. e., an *integrated* document consist of a hierarchy *sub-documents* (files) that can depend acyclically on each other. Sub-documents can have different document types in order to capture documentations consisting of documentation, models, proofs, code of various forms and other technical artifacts. We call the main sub-document type, for historical reasons, *theory*-files. A theory file consists of a *header*, a *context definition*, and a body consisting of a sequence of *command*s (Fig. 1). 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 Example          (* Name of the "theory"                    *)
  imports               (* Declaration of "theory" dependencies     *)
    Main                (* Imports a library called "Main"          *)
  keywords              (* Registration of keywords defined locally *)
    requirement         (* A command for describing requirements    *)
```

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

We distinguish fundamentally two different syntactic levels:

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

On the semantic level, we assume a validation process for an integrated document, where the semantics of a command is a transformation $\theta \rightarrow \theta$ for some system state θ. This document model can be instantiated with outer-syntax commands for common text elements, e. g., **`section`**`‹...›` or **`text`**`‹...›`. Thus, users can add informal text to a sub-document using a text command:

```
text‹This is a description.›
```

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

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

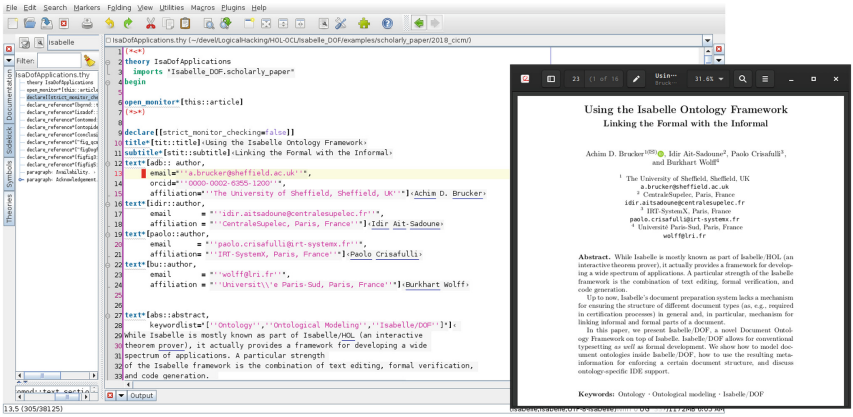

Fig. 2. The Isabelle/DOF IDE (left) and the corresponding PDF (right).

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

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

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

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

Implementability of the Assumed Document Model. Batch-mode checkers for DOF can be implemented in all systems of the LCF-style prover family, i. e., systems with a type-checked `term`, and abstract `thm`-type for theorems (protected by a kernel). This includes, e. g., ProofPower, HOL4, HOL-light, Isabelle, as well as 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 an IDE [10,18,28,29]. For our implementation of DOF, called Isabelle/DOF, we are using the Isabelle platform [25]. Figure 2 shows a screen-shot of an introductory paper on Isabelle/DOF [12] presenting a number of application scenarios and user-interface aspects. On the left, we represented the Isabelle/DOF IDE, while on the right, the generated presentation in PDF is shown.

Isabelle provides, beyond the features required for DOF, a lot of additional benefits. For example, it also allows the asynchronous evaluation and checking of the document content [10,28,29] and is dynamically extensible. Its IDE provides a *continuous build, continuous check* functionality, syntax highlighting, and IntelliSense-like auto-completion. It also provides infrastructure for displaying meta-information (e. g., binding and type annotation) as pop-ups, while hovering over sub-expressions. A fine-grained dependency analysis allows the processing

of individual parts of theory files asynchronously, allowing Isabelle to interactively process large (hundreds of theory files) documents. Isabelle can group sub-documents into sessions, i. e., sub-graphs of the document-structure that can be "pre-compiled" and loaded instantaneously, i. e., without re-processing.

3 The DOF Design

DOF consists basically of two parts: 1. the declaration of new keywords and new commands allowing for the specification of ontological concepts in our Ontology Definition Language (ODL), and 2. the definition of text-elements that are "ontology-aware," i. e., perform the necessary checks to ensure compliance to an imported ontology. This represents a partial instantiation of the underlying generic document model. The document language can be extended (recall the `keywords`-section) dynamically, i. e., new *user-defined* can be introduced at run-time. This is similar to the definition of new functions in an interpreter.

We illustrate the design of DOF by modeling a small ontology that can be used for writing formal specifications that, e. g., could build the basis for an ontology for certification documents used in processes such as Common Criteria [17] or CENELEC 50128 [16].[1] Moreover, in examples of certification documents, we refer to a controller of a steam boiler that is inspired by the famous steam boiler formalization challenge [9].

3.1 Ontology Modeling in ODL

Conceptually, ontologies specified in ODL consist of:

- *document classes* (syntactically marked by the `doc_class` keyword) that describe concepts;
- an optional document base class expressing single inheritance extensions;
- *attributes* specific to document classes, where
  - attributes are 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*);
- a special link, the reference to a super-class, establishes an *is-a* relation between classes;
- classes may refer to other classes via a regular expression in a *where* clause (classes with a where clauses are called *monitor classes*);
- attributes may have default values in order to facilitate notation.

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

[1] The Isabelle/DOF distribution contains an ontology for writing documents for a certification according to CENELEC 50128.

Listing 1.1. An example ontology modeling simple certification documents, including scientific papers such as [12]; also recall Fig. 2.

```
doc_class title  = short_title :: "string option" <= "None"
doc_class author = email      :: "string" <= "''''"

datatype classification = SIL0 | SIL1 | SIL2 | SIL3 | SIL4

doc_class abstract =
    keywordlist :: "string list"      <= []
    safety_level :: "classification"  <= "SIL3"
doc_class text_section =
    authored_by :: "author set"       <= "{}"
    level       :: "int  option"      <= "None"

type_synonym notion = string

doc_class introduction = text_section +
    authored_by :: "author set"       <= "UNIV"
    uses :: "notion set"
doc_class claim = introduction +
    based_on :: "notion list"
doc_class technical = text_section +
    formal_results  :: "thm list"
doc_class "definition" = technical +
    is_formal :: "bool"
    property  :: "term list"          <= "[]"

datatype kind = expert_opinion | argument | proof

doc_class result = technical +
    evidence  :: kind
    property  :: "thm list"           <= "[]"
doc_class example = technical +
    referring_to :: "(notion + definition) set" <=  "{}"
doc_class "conclusion" = text_section +
    establish   :: "(claim × result) set"
```

model, ODL definitions may be arbitrarily intertwined with standard HOL type definitions. Finally, document class definitions result in themselves in a HOL-types in order to allow *links* to and between ontological concepts.

Listing 1.1 shows an example ontology for mathematical papers (an extended version of this ontology was used for writing [12], also recall Fig. 2). The commands **datatype** (modeling fixed enumerations) and **type_synonym** (defining type synonyms) are standard mechanisms in HOL systems. Since ODL is an add-on, we have to quote sometimes constant symbols (e. g., `"proof"`) to avoid confusion with predefined keywords. ODL admits overriding (such as `authored_by` in the document class `introduction`), where it is set to another library con-

stant, but no overloading. All `text_section` elements have an optional `level` attribute, which will be used in the output generation for the decision if the context is a section header and its level (e. g., chapter, section, subsection). While *within* an inheritance hierarchy overloading is prohibited, attributes may be redeclared freely in independent parts (as is the case for `property`).

3.2 Meta-Types as Types

To express the dependencies between text elements to the formal entities, e. g., `term` (λ-term), `typ`, or `thm`, we represent the types of the implementation language *inside* the HOL type system. We do, however, not reflect the data of these types. They are just declared abstract types, "inhabited" by special constant symbols carrying strings, for example of the format `@{thm <string>}`. When HOL expressions were used to denote values of `doc_class` instance attributes, this requires additional checks after conventional type-checking that this string represents actually a defined entity in the context of the system state θ. For example, the `establish` attribute in the previous section is the power of the ODL: here, we model a relation between *claims* and *results* which may be a formal, machine-check theorem of type `thm` denoted by, for example: `property ="[@{thm ''system_is_safe''}]"` in a system context θ 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 θ. Another instance of this process, which we call *second-level type-checking*, are term-constants generated from the ontology such as `@{definition <string>}`. For the latter, the argument string must be checked that it represents a reference to a text-element having the type `definition` according to the ontology in Listing 1.1.

3.3 Annotating with Ontology Meta-Data: Outer Syntax

DOF introduces its own family of text-commands, which allows having side effects of the global context θ and thus to store and manage own meta-information. Among others, DOF provides the commands `section*[<meta-args>]‹...›`, `subsection*[<meta-args>]‹...›`, or `text*[<meta-args>]‹...›`. Here, the argument `<meta-args>` is a syntax for declaring instance, class and attributes for this text element, following the scheme

```
<ref> :: <class_id>, attr_1 = <expr>, ..., attr_n = <expr>
```

The `<class_id>` can be omitted, which represents the implicit superclass `text`, where `attr_i` must be declared attributes in the class and where the HOL `<expr>` must have the corresponding HOL type. Attributes from a class definition may be left undefined; definitions of attribute values *override* default values or values of super-classes. Overloading of attributes is not permitted in DOF.

We can now annotate a text as follows. First, we have to place a particular document into the context of our conceptual example ontology (Listing 1.1):

```
theory Steam_Boiler
  imports
    tiny_cert  (* The ontology defined in Listing 1.1. *)
begin
```

This opens a new document (theory), called `Steam_Boiler` that imports our conceptual example ontology "`tiny_cert`" (stored in a file `tiny_cert.thy`).[2] Now we can continue to annotate our text as follows:

```
title*[a] ‹The Steam Boiler Controller›
abstract*[abs, safety_level="SIL4", keywordlist = "[''controller'']"]‹
  We present a formalization of a program which serves to control the
  level of water in a steam boiler.
›

section*[intro::introduction]‹Introduction›
text‹We present ...›

section*[T1::technical]‹Physical Environment›
text‹
  The system comprises the following units
  • the steam-boiler
  • a device to measure the quantity of water in the steam-boiler
  • ...
›
```

Where `title*[a ...]` is a predefined macro for `text*[a::title, ...]‹...›` (similarly `abstract*`). The macro `section*` assumes a class-id referring to a class that has a `level` attribute. We continue our example text:

```
text*[c1::contrib_claim, based_on="[''pumps'',''steam boiler'']" ]‹
  As indicated in @{introduction "intro"}, we the water level of the
  boiler is always between the minimum and the maximum allowed level.
›
```

The first text element in this example fragment *defines* the text entity `c1` and also references the formerly defined text element `intro` (which will be represented in the PDF output, for example, by a text anchor "Section 1" and a hyperlink to its beginning). The antiquotation `@{introduction ...}`, which is automatically generated from the ontology, is immediately validated (the link to `intro` is defined) and type-checked (it is indeed a link to an `introduction` text-element). Moreover, the IDE automatically provides editing and development support such as auto-completion or the possibility to "jump" to its definition by clicking on the antiquotation. The consistency checking ensures, among others,

[2] The usual import-mechanisms of the Isabelle document model applies also to ODL: ontologies can be extended, several ontologies may be imported, a document can validate several ontologies.

that the final document will not contain any "dangling references" or references to entities of another type.

DOF as such does not require a particular evaluation strategy; however, if the underlying implementation is based on a declaration-before-use strategy, a mechanism for forward declarations of references is necessary:

```
declare_reference* [<meta-args>]
```

This command declares the existence of a text-element and allows for referencing it, although the actual text-element will occur later in the document.

3.4 Editing Documents with Ontology Meta-Data: Inner Syntax

We continue our running example as follows:

```
text*[d1::definition]‹
  We define the water level @{term "level"} of a system state
  @{term "σ"} of the steam boiler as follows:
›
definition level :: "state → real" where
          "level σ =  level₀ + ..."
update_instance*[d1::definition,
                 property += "[@{term ''level σ =  level₀ + ...''}]"]

text*[only::result,evidence="proof"]‹
  The water level is never lower than @{term "level₀"}:
›
theorem level_always_above_level_0: "∀ σ. level σ ≥ level₀"
  unfolding level_def
  by auto
update_instance*[only::result,
                 property += "[@{thm ''level_always_above_level_0''}]"]
```

As mentioned earlier, instances of document classes are mutable. We use this feature to modify meta-data of these text-elements and "assign" them to the property-list afterwards and add results from Isabelle definitions and proofs. The notation `A+=X` stands for `A := A + X`. This mechanism can also be used to define the required relation between *claims* and *results* required in the `establish`-relation required in a `summary`.

3.5 ODL Class Invariants

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

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

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

where `result`, `conclusion`, and `introduction` are the set of all possible instances of these document classes. All specified constraints are already checked in the IDE of DOF while editing; it is however possible to delay a final error message till the closing of a monitor (see next section). The third constraint enforces that the user sets the `authored_by` set, otherwise an error will be reported.

3.6 ODL Monitors

We call a document class with an accept-clause a *monitor*. Syntactically, an accept-clause contains a regular expression over class identifiers. We can extend our `tiny_cert` ontology with the following example:

```
doc_class article =
    style_id :: string   <= "''CENELEC50128''"
    accepts "(title ~~ ⦃author⦄+ ~~ abstract ~~ ⦃introduction⦄+  ~~
             ⦃technical || example⦄+ ~~ ⦃conclusion⦄+)"
```

Semantically, monitors introduce a behavioral element into ODL:

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

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

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

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

Monitors manage an implicit attribute `trace` containing the list of "observed" text element instances belonging to the accept-set. Together with the concept of ODL class invariants, it is possible to specify properties of a sequence of instances occurring in the document section. For example, it is possible to express that in

the sub-list of `introduction`-elements, the first has an `introduction` element with a `level` strictly smaller than the others. Thus, an introduction is forced to have a header delimiting the borders of its representation. Class invariants on monitors allow for specifying structural properties on document sections.

3.7 Document Representation

Up to now, we discussed the support of ontological concepts in the context of an IDE, i. e., a rather dynamic environment that, e. g., allows for interactive querying and displaying of information. Certification processes often require "static" documents, e. g., in a format such as PDF/A that are designed for archiving and long-term preservation of electronic documents, are required.

While many concepts of ODL can easily be mapped to such static formats, more dynamic features (e. g., references) requires additional considerations such as ensuring that references point to text elements that have a unique identifier that is visible in the actual document representation. Currently, the definition of a static document representation is not part of DOF itself and, thus, depends on the underlying implementation. We refer the reader to Sect. 4.6 for details.

4 The Isabelle/DOF Implementation

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

- the entire Isabelle/DOF is a "pure add-on," i. e., we deliberately resign on the possibility to modify Isabelle itself.
- we made a small exception to this rule: the Isabelle/DOF package modifies in its installation about 10 lines in the LaTeX generator `thy_output.ML` which greatly simplifies the architecture.[3]
- 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 [30]. 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.

[3] Earlier versions of Isabelle/DOF used an additional LaTeX-to-LaTeX translator that needed to be integrated into the document build process.

4.1 Writing Isabelle/DOF as User-Defined Plugin in Isabelle/Isar

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

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

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

All the text samples shown here have to be in the context of an SML file or in an `ML‹...›` command inside a theory file.

Operations follow the model-view-controller paradigm, where Isabelle/Isar provides the controller part. A typical model operation has the type:

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

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

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

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

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

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

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

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

The "model" `declare_reference_opn` and "new" `attributes` parts were combined via the piping operator and registered in the Isar toplevel:

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

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

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

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

4.2 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(
          Thy_Output.antiquotation @{binding docitem}
                                   docitem_antiq_parser
                                   (docitem_antiq_gen default_cid) #>
          ML_Antiquotation.inline  @{binding docitem_value}
                                   ML_antiq_docitem_value)
```

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

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

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

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

4.3 Implementing Second-Level Type-Checking

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

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

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

4.4 Programming Class Invariants

For the moment, there is no high-level syntax for the definition of class invariants. A formulation, in SML, of the first class-invariant in Sect. 3.5 is straight-forward:

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

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

Isabelle's code generator can in principle generate class invariant code from a high-level syntax. Since class-invariant checking can result in an efficiency problem—they are checked on any edit—and since invariant programming involves a deeper understanding of ontology modeling and the Isabelle/DOF implementation, we backed off from using this technique so far.

4.5 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
```

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 Sect. 3.6). During top-down document validation, whenever a text-element is encountered, it is checked if a monitor is *enabled* for this class; in this case, the `next`-operation is executed. The transformed automaton recognizing the rest-language is stored in `docobj_tab` if possible; otherwise, if `next` fails, an error is reported. The automata implementation is, in large parts, generated from a formalization of functional automata [23].

4.6 Document Representation

Isabelle/DOF can generate PDF documents, using a LaTeX-backend (for end users there is no need to edit LaTeX-code manually). For PDF documents, a specific representation, including a specific layout or formatting of certain text types (e. g., title, abstract, theorems, examples) is required: for each ontological concept (using the `doc_class`-command), a representation for the PDF output needs to be defined. The LaTeX-setup of Isabelle/DOF provides the `\newisadof` -command and an inheritance-based dispatcher, i. e., if for a concept no LaTeX-representation is defined, the representation of its super-concept is used.

Recall the document class `abstract` from our example ontology (Listing 1.1). The following LaTeX-code (defined in a file `tiny_cert.sty`) defines the representation for abstracts, re-using the the standard `abstract`-environment:

```
\newisadof{tiny_cert.abstract}[reference=,class_id=%
        ,keywordlist=,safety_level=][1]{%
  \begin{isamarkuptext}%
    \begin{abstract}\label{\commandkey{reference}}%
      #1\\ % this is the main text of the abstract
      \ifthenelse{\equal{\commandkey{safety_level}}{}}{}{%
        \medskip\noindent{Safety Level:} \commandkey{safety_level}\\%
      }
      \ifthenelse{\equal{\commandkey{keywordlist}}{}}{}{%
        \medskip\noindent{\textbf{Keywords:}} \commandkey{keywordlist}%
      }
    \end{abstract}%
  \end{isamarkuptext}%
}
```

The `\newisadof` takes the name of the concept as first argument, followed by a list of parameters that is the same as the parameters used in defining the concept with `doc_class`. Within the definition section of the command, the main argument (written in the actual document within ‹ ... ›) is accessed using `#1`. The parameters can be accessed using the `\commandkey`-command. In our example, we print the abstract within `abstract`-environment of LaTeX. Moreover, we test if the parameters `safety_level` and `keywordlist` are non-empty and, if yes, print them as part of the abstract.

5 Conclusion and Related Work

5.1 Related Work

Our work shares similarities with existing ontology editors such as Protégé [5], Fluent Editor [1], NeOn [2], or OWLGrEd [4]. These editors allow for defining ontologies and also provide certain editing features such as auto-completion. In contrast, Isabelle/DOF does not only allow for defining ontologies, directly after defining an ontological concept, they can also be instantiated and their correct use is checked immediately. The document model of Jupyter Notebooks [8] comes probably closest to our ideal of a "living document."

Finally, the LaTeX that is generated as intermediate step in our PDF generation is conceptually very close to SALT [19], with the difference that instead of writing LaTeX manually it is automatically generated and its consistency is guaranteed by the document checking of Isabelle/DOF.

5.2 Conclusion

We presented the design of DOF, an ontology framework designed for formal documents developed by interactive proof systems. It foresees a number of specific features—such as monitors, meta-types as-types or semantic macros generated from a typed ontology specified in ODL—that support the specifics of such documents linking formal and informal content. As validation of these concepts,

we present Isabelle/DOF, an implementation of DOF based on Isabelle/HOL. Isabelle/DOF is unique in at least one aspect: it is an integrated environment that allows both defining ontologies and writing documents that conform to a set of ontological rules, and both are supported by editing and query features that one expects from a modern IDE.

While the batch-mode part of DOF can, in principle, be re-implemented in any LCF-style prover, Isabelle/DOF is designed for fast interaction in an IDE. It is this feature that allows for a seamless development of ontologies together with validation tests checking the impact of ontology changes on document instances. We expect this to be a valuable tool for communities that still have to develop their domain specific ontologies, be it in mathematical papers, formal theories, formal certifications or other documents where the consistency of formal and informal content has to be maintained under document evolution. Today, in some areas such as medicine and biology, ontologies play a vital role for the retrieval of scientific information; we believe that leveraging these ontology-based techniques to the field of formal software engineering can represent a game changer.

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

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".

References

1. Fluent editor (2018). http://www.cognitum.eu/Semantics/FluentEditor/
2. The neon toolkit (2018). http://neon-toolkit.org
3. Ontologies (2018). https://www.w3.org/standards/semanticweb/ontology
4. Owlgred (2018). http://owlgred.lumii.lv/
5. Protégé (2018). https://protege.stanford.edu
6. Archive of formal proofs (2019). https://afp-isa.org
7. Ibm engineering requirements management doors family (2019). https://www.ibm.com/us-en/marketplace/requirements-management
8. Jupyter (2019). https://jupyter.org/
9. Abrial, J.-R.: Steam-boiler control specification problem. In: Abrial, J.-R., Börger, E., Langmaack, H. (eds.) Formal Methods for Industrial Applications. LNCS, vol. 1165, pp. 500–509. Springer, Heidelberg (1996). https://doi.org/10.1007/BFb0027252
10. Barras, B., et al.: Pervasive parallelism in highly-trustable interactive theorem proving systems. In: Carette, J., Aspinall, D., Lange, C., Sojka, P., Windsteiger, W. (eds.) CICM 2013. LNCS (LNAI), vol. 7961, pp. 359–363. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39320-4_29
11. Blanchette, J.C., Haslbeck, M., Matichuk, D., Nipkow, T.: Mining the archive of formal proofs. In: Kerber, M., Carette, J., Kaliszyk, C., Rabe, F., Sorge, V. (eds.) CICM 2015. LNCS (LNAI), vol. 9150, pp. 3–17. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-20615-8_1

12. Brucker, A.D., Ait-Sadoune, I., Crisafulli, P., Wolff, B.: Using the isabelle ontology framework. In: Rabe, F., Farmer, W.M., Passmore, G.O., Youssef, A. (eds.) CICM 2018. LNCS (LNAI), vol. 11006, pp. 23–38. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96812-4_3
13. Brucker, A.D., Brügger, L., Wolff, B.: Formal network models and their application to firewall policies. Archive of Formal Proofs (2017). http://www.isa-afp.org/entries/UPF_Firewall.shtml
14. Brucker, A.D., Herzberg, M.: The Core DOM. Archive of Formal Proofs (2018). http://www.isa-afp.org/entries/Core_DOM.html
15. Brucker, A.D., Tuong, F., Wolff, B.: Featherweight OCL: a proposal for a machine-checked formal semantics for OCL 2.5. Archive of Formal Proofs (2014). http://www.isa-afp.org/entries/Featherweight_OCL.shtml
16. BS EN 50128:2011: Bs en 50128:2011: Railway applications - communication, signalling and processing systems - software for railway control and protecting systems. Standard, Britisch Standards Institute (BSI) (2014)
17. Common Criteria: Common criteria for information technology security evaluation (version 3.1), Part 3: Security assurance components (2006)
18. Faithfull, A., Bengtson, J., Tassi, E., Tankink, C.: Coqoon - an IDE for interactive-proof development in coq. STTT **20**(2), 125–137 (2018). https://doi.org/10.1007/s10009-017-0457-2
19. Groza, T., Handschuh, S., Möller, K., Decker, S.: SALT - semantically annotated LaTeX for scientific publications. In: Franconi, E., Kifer, M., May, W. (eds.) ESWC 2007. LNCS, vol. 4519, pp. 518–532. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72667-8_37
20. Hou, Z., Sanan, D., Tiu, A., Liu, Y.: A formal model for the SPARCv8 ISA and a proof of non-interference for the LEON3 processor. Archive of Formal Proofs (2016). http://isa-afp.org/entries/SPARCv8.html
21. Hupel, L., Zhang, Y.: CakeML. Archive of Formal Proofs (2018). http://isa-afp.org/entries/CakeML.html
22. Klein, G., et al.: Comprehensive formal verification of an OS microkernel. ACM Trans. Comput. Syst. **32**(1), 2:1–2:70 (2014). https://doi.org/10.1145/2560537
23. Nipkow, T.: Functional automata. Archive of Formal Proofs (2004). http://isa-afp.org/entries/Functional-Automata.html. Formal proof development
24. Nipkow, T.: Splay tree. Archive of Formal Proofs (2014). http://isa-afp.org/entries/Splay_Tree.html. Formal proof development
25. Nipkow, T., Wenzel, M., Paulson, L.C. (eds.): Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
26. Sprenger, C., Somaini, I.: Developing security protocols by refinement. Archive of Formal Proofs (2017). http://isa-afp.org/entries/Security_Protocol_Refinement.html. Formal proof development
27. Verbeek, F., et al.: Formal specification of a generic separation kernel. Archive of Formal Proofs (2014). http://isa-afp.org/entries/CISC-Kernel.html. Formal proof development
28. Wenzel, M.: Asynchronous user interaction and tool integration in Isabelle/PIDE. In: Klein, G., Gamboa, R. (eds.) ITP 2014. LNCS, vol. 8558, pp. 515–530. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08970-6_33
29. Wenzel, M.: System description: Isabelle/jEdit in 2014. In: Proceedings Eleventh Workshop on User Interfaces for Theorem Provers, UITP 2014, Vienna, Austria, 17th July 2014, pp. 84–94 (2014). https://doi.org/10.4204/EPTCS.167.10
30. Wenzel, M.: The Isabelle/Isar Reference Manual (2017). Part of the Isabelle distribution