26.20 OMDoc as a data format for √eriFun

Project Home	`http://www.verifun.de/`
Authors	Normen Müller School of Engineering and Science, International University Bremen

√eriFun (Verification of Functional programs) is a semi-automated system for the verification of programs written in a simple functional programming language $\mathcal{FP}$. The system has been developed since 1998 at the university of Darmstadt for use in education and research. The main design goals are a clearly structured, didactically suited system interface (Figure 26.20), an easily portable implementation (JAVA) and an easily but also powerful proof calculus [WS02]. The system's object language consists of a simple definition principle for free data structures, called *sorts* (see Chapter 16), a recursive definition principle for *functions*, and finally a definition principle for statements, called *lemmas*, about the data structures and the functions. To prove a statement √eriFun supports the user with a couple of inference rules aggregated in *tactics*. A collection of *sorts*, *functions*, *lemmas*, and *proofs* is called a √eriFun *program*. Common file commands, which are based on the JAVA binary serialization mechanism, are provided to save and reload intermediate work.

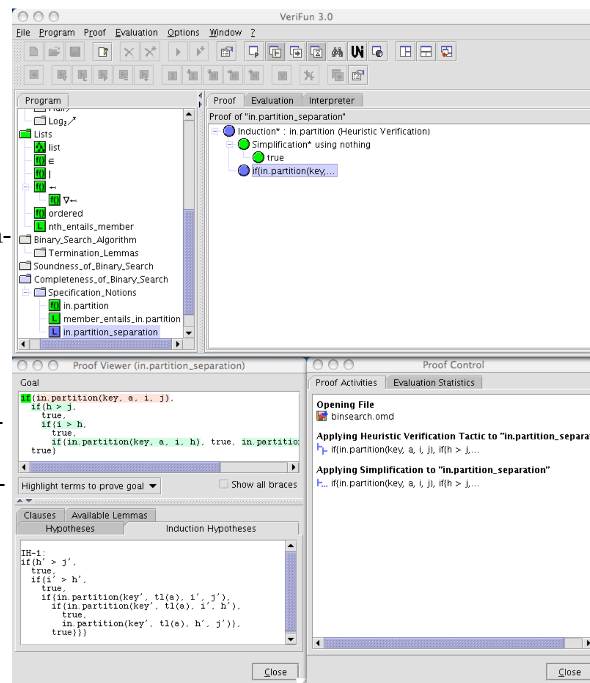

Fig. 26.27. A √eriFun session

The OMDoc interface for √eriFun described here (see [Mül05] for details) was introduced to alleviate the following drawbacks of the former I/O mechanism based on JAVA binary serialization:

– Files are only machine-readable. Thus, e.g. if the files became corrupted by any circumstance, there is no change of a manual repair.
– Files are strongly bound to the version of the system. Thus any internal system modifications make the files unreadable.
– Files are not interchangeable with other theorem provers or other mathematical software systems. Thus the information inside the files are only accessible by √eriFun.

√eriFun's interface to OMDoc can be divided into two parts: Encoding and decoding of √eriFun programs to and from OMDoc respectively.

Encoding. In a typical session with the system, a user defines a *program* by stipulating the *sorts* and the *functions* of the program, defines *lemmas* about the sorts and the functions of the program, and finally verifies these lemmas and the termination of the functions.

In general a *program* is mapped to two OMDoc files: The first one consists of the user-defined elements[21] and in the second one √eriFun's logic comprising the predefined symbols, the type system and the proof tactics is defined. At each case one √eriFun-generated-OMDoc file is composed of one `theory` element. The name of a user-defined theory can be set by the user, whereas the name of the theory √eriFun is based on is fixed to VAFP.

Functions are declared by `symbol` elements that also introduces the type of the function (Subsection 15.2.3). The body of a function is encoded as an OpenMath object inside a `definition` element. The corresponding `symbol` element is referenced by the `definition` element in the `for` and relating termination assertions in the `existence` attribute.

Note that instead of using `name` attributes, which only allow XML simple names, we generate a unique ID. The actual √eriFun names are represented in `presentation` elements or rather their `use` elements (Listing 26.30). By using this technique we can use any character string[22] for element names. To cover the whole set of √eriFun fixities (`prefix` (the default), `infix`, `postfix`, `infixl`, `infixr`, and `outfix`) we had to extend the OMDoc format by `infixl` and `infixr`. However, it was not necessary to also add the `outfix` value, but encoding of `outfix` functions is treated slightly different: The name of the function is encoded in the `lbrack` and `rbrack` attribute respectively of the relating `presentation` element and the `use` element is left empty[23].

Lemmata are mapped to `assertion` elements, the value "`lemma`" being assigned to the `type` attribute. The formula of a lemma, analogous to function bodies, is encoded as an OpenMath object inside an `assertion` element.

Particularly convenient is the direct mapping of √eriFun proofs to the OMDoc presentation of proofs. Verifications of lemmas and termination

[21] Actually there are also automatically system-generated elements included, but we may neglect those at this point.

[22] √eriFun has full Unicode [Inc03] support

[23] As a consequence the previous mentioned special encoding feature does not hold for outfix functions

analysis of functions are represented in `proof` elements. The assertion to be proven is referenced in the `for` attribute. VAFP-tactics used inside a proof to achieve the various proof steps (encoded in `derive` elements) are denoted by `method` elements. Parameters heuristically computed by the system or manually annotated by the user are encoded as OPENMATH objects and appended to each proof step. Furthermore each proof step in ✓eriFun is annotated with a sequence of the form $h_1, \ldots h_n, \forall \ldots ih_1, \ldots, \forall \ldots ih_l \vdash goal$ whereas the expressions h_i are the hypotheses, the expressions $\forall \ldots ih_k$ are the induction hypotheses, and the expression *goal* is the goal-term of the sequence. Such a sequent is represented by `assumption` and `conclusion` child-elements respectively of the relating `derive` element.

Listing 26.28. A polymorphic ✓eriFun sort

```
structure list [@value] <=
  ∅,
  [ infixr ,100] :: (hd : @value, tl : list [@value])
```

Sorts are wrapped inside `adt` elements. At this point this integration process provoked two further adaptations of the OMDOC standard. On the one hand, in contrast to OMDOC, sorts in ✓eriFun could be polymorphic (Listing 26.28). This led to the additional, optional `parameters` attribute of an `adt` element (Listing 26.29). Within this new attribute one can declare by a comma separated list the names of type variables of the abstract data type.

Listing 26.29. A polymorphic OMDOC ADT

```
<adt xml:id="vf7b9f3e59−e78e−4221−8064−7fa0c5689f5d.adt" parameters="value">
  <sortdef name="vf7b9f3e59−e78e−4221−8064−7fa0c5689f5d" type="free">
    <constructor name="vf8a6673ac−c1d9−4698−b6ee−90213539a984"/>
    <constructor name="vf38164505−4983−417f−8bdc−6a42b046e933">
5     <argument>
        <type system="simpletypes">
          <OMOBJ xmlns="http://www.openmath.org/OpenMath">
            <OMV name="value"/>
          </OMOBJ>
10      </type>
        <selector name="vf9fc4c672−207f−45c0−ae61−1f675fde7aed" total="yes"/>
      </argument>
      <argument>
        <type system="simpletypes">
15        <OMOBJ xmlns="http://www.openmath.org/OpenMath">
            <OMA>
              <OMS cd="VeriFun" name="vf7b9f3e59−e78e−4221−8064−7fa0c5689f5d"/>
              <OMV name="value"/>
            </OMA>
20        </OMOBJ>
        </type>
        <selector name="vf55767f3a−b019−4308−88f9−d68ee0db595e" total="yes"/>
      </argument>
    </constructor>
25 </sortdef>
</adt>
```

On the other hand, the child elements of a `constructor` element had to be expanded by an additional `type` element to specify the type of the formal

parameter of the parent `constructor` element. Listing 26.30 illustrates the corresponding `presentation` elements of the ADT in Listing 26.29.

Listing 26.30. Representation of √eriFun names to OMDoc

```
<presentation for="#vf7b9f3e59-e78e-4221-8064-7fa0c5689f5d" role="applied">
  <use format="VeriFun">list</use>
</presentation>
<presentation for="#vf8a6673ac-c1d9-4698-b6ee-90213539a984" role="applied"
              bracket-style="math" precedence="1" fixity="prefix" lbrack="(" rbrack=")">
  <use format="VeriFun">∅</use>
</presentation>
<presentation for="#vf38164505-4983-417f-8bdc-6a42b046e933" role="applied"
              bracket-style="math" precedence="100" fixity="infixr" lbrack="(" rbrack=")">
  <use format="VeriFun">::</use>
</presentation>
<presentation for="#vf9fc4c672-207f-45c0-ae61-1f675fde7aed" role="applied"
              bracket-style="math" precedence="1" fixity="prefix" lbrack="(" rbrack=")">
  <use format="VeriFun">hd</use>
</presentation>
<presentation for="#vf55767f3a-b019-4308-88f9-d68ee0db595e" role="applied"
              bracket-style="math" precedence="1" fixity="prefix" lbrack="(" rbrack=")">
  <use format="VeriFun">tl</use>
</presentation>
```

Decoding. The decoding of a √eriFun program represented in OMDoc is reverse to the encoding mechanism. First we create an empty program and then start the sequential decoding of each `adt`, `symbol` and its relating `definition`, and `assertion` element back into the $\mathcal{FP}$ syntax. After a successful reconstruction of an element it is appended to the current program. Right after such an insertion we check for a `proof` element containing a reference to this new program element. If a proof exists, we re-play all the proof steps and associate the recreated √eriFun proof to the corresponding program element.

One aspect of this decoding exercise is worth mentioning here. The √eriFun system also benefited by the development of the OMDoc standard: Revelation of bugs deep in the system! Especially $\mathcal{FP}$ parser errors and inconsistencies in proof tactics applications could be discovered. Maybe those errors would never have been detected, because in most cases the user is not able to produce them manually, but this errors are automatically generated by the system. So with the assistance of the strict encoding and decoding to and from OMDoc respectively we were able to achieve a much more robust verification system.

By the integration of the open content Markup language OMDoc into the semi-automated theorem prover √eriFun, we made the system more reliable and facilitate the participation in the mathematical network to serve as yet another service. Functional programs and especially proof of statements created in √eriFun are now open to the public. The data is human-readable, machine-understandable, no longer subjected to a particular version of the system. Thus, √eriFun generated knowledge became accessible, robust, interchangeable and transparent.