A report on an experiment in porting formal theories from Isabelle/HOL to Ecore and ACL2*

J. Aransay, J. Divasón, J. Heras, L. Lambán,
V. Pascual, A. L. Rubio and J. Rubio

Departamento de Matemáticas y Computación, Universidad de La Rioja,
Edif. Luis Vives, c. Luis de Ulloa s/n. 26004. Spain
{jesus-maria.aransay,jose.divasonm,jonathan.heras,
lalamban,mvico,arubio,julio.rubio}@unirioja.es

Abstract. In this report we present a tool that enables to automatically transport specifications in the Isabelle/HOL theorem prover to some other systems. In particular, for a given theory, including type definitions, definitions and theorems, an ACL2 theory with similar constructs will be generated. Additionally, from the Isabelle/HOL theory an specification in the Ecore language will be generated, by means of a UML class diagram and OCL restrictions. The transformations are done by means of XSLT, taking advantage of the fact that some of the intermediary languages are based on XML schemas, and some Java programs for additional transformations.

1 Introduction

A meaningful and aspiring goal in the field of formalization of Mathematics with theorem provers is the capacity to transfer results proven with one concrete tool to different tools. The idea is natural, since from a naive point of view one would expect that once a proving effort has been carried out to formalize a result in a given system, the result should be available (and trusted) in any other system.

Unfortunately, this possibility is not supported by the current technology because of various reasons. First, every system implements its own logic and even has a particular "proving style"; thus, expressivity among logics is different. For instance, ACL2 (which we will describe in detail later) is a subset of first-order logic, whereas Isabelle/HOL is an implementation of a higher-order logic (with type classes and polymorphism), or Coq is based on a (higher-order) dependent type theory. Even worse, proofs are not carried out in a similar way; ACL2 has a great degree of automation, based on inductive tactics provided by the user, whereas Isabelle/HOL encourages the use of the Isar (intelligible semi-automated reasoning) language in proofs, a human readable format, where

* This work has been supported by the Spanish Government (project MTM2009-13842-C02-01 of MEC), by FORMATH project, nr. 243847, of the FET program within the FP7 European Commission and by Universidad de La Rioja, research grant FPI-UR-12.

proofs can be carried out in a backward or forward manner and with different degrees of automation and detail. Then, Coq proofs are also carried out by means of tactics or tacticals in the Gallina language, but the range of tactics and automation is different from the one in Isabelle/HOL. Every single system has its particularities derived from its implementation, logical foundations, goals or purposes, as programming languages do (or even more, since theorem provers perform more complex tasks).

Despite of this, one could think of sharing between different systems at least specifications, definitions, statements or proofs that rely on the common places of each system' logic. Unfortunately, after decades of tools' development, every system has its own syntax and libraries (which are worth to be used in new developments), which makes complicated to directly move proofs from one system to the other. Even if we are dealing with systems based on the same logical foundation, such as HOL4 and Isabelle/HOL, each one of them includes its own *definitional extensions* (as for instance type classes and ad-hoc polymorphism, packages for recursive functions' definition and so on) that cannot be directly transferred from one system to the other, but somehow emulated.

Nevertheless, relevant efforts have been carried out to fill the gaps that we have enumerated, even getting the direct translation of specifications and proofs among systems. Usually they are based on the idea of considering one system as a subset of another, and then mapping elements of that system to this one. A different approach consists of defining a "meta-language" in which various systems can be translated (ideally in both directions, to the "meta-language" and from the "meta-language"). We will present some of these tools in detail in Section 5.1.

In this work, we propose a different approach. From a given development in Isabelle/HOL [5] (where a formalization of an algorithm computing a particular form of matrices is presented), we define a set of transformations that enable us to transfer the datatypes, specification of definitions and theorems' statements to both ACL2 and to an Ecore model (given by UML class definitions and OCL restrictions). The translations to Ecore and ACL2 are both based on an intermediary XML document, automatically generated from the Isabelle sources, and compliant with an schema that we have also developed, XLL (standing for Xmall Logical Language). XLL includes elements for specifying datatypes, operations and logical statements relating them; its syntax can be divided into two substantial parts; one of them is used to state the definitions of data types and its operations appearing in the input Isabelle theory, whereas the second part contains the statements of the properties included in the same theory; we postpone the XLL syntax details to its description in Section 3. The XML schema of XLL performs additional operations to ensure that the statements are related to datatypes and operations that have been previously introduced (*i.e.*, that the specification of properties corresponds with the elements already introduced in a theory).

The XLL language is used as a common basis to perform translations to both Ecore and ACL2 of the original Isabelle developments (or *theories*, in the Isabelle jargon, even when an Isabelle theory can include additional constructs).

The translation to Ecore, where OCL is also a first order language, serves as a general purpose formal specification of the theory we are carrying out. The translation includes the definition of data types and the specification of properties; both are made starting from the XLL file of each theory; the translation of the data types in XLL gives place to the Ecore classes (or the "context") over which the OCL properties have to be stated; the translation of the statements in XLL produces OCL properties, which make reference to the generated Ecore classes and its operations. Additionally, this specification of properties and data types could be used to build (automatically or provided by the user, depending on the underlying technologies) instances of the Isabelle specifications, where the properties (the theorems' statements) can be thoroughly checked or refuted, (once the Isabelle specifications have been proved, and considering the translation as correct, faults could be only originated in the representation chosen for the instances; if the Isabelle properties have not been proved, the Ecore models could be considered as a testing tool for the properties).

Then, the translation from XLL to ACL2 shows that the formal specification of the theory can be ported to some other theorem proving tools. The reason why we have chosen ACL2 is that we are familiar with its syntax, its simplicity (apart of the particular treatment applied to existential and universal quantifiers, that we introduce later) and also our personal interest in transporting the theory about matrices that we have completed in Isabelle to it. Nevertheless, there is nothing inherent to ACL2 that is needed to carry out the translation, so probably a similar transformation could be carried out to some other theorem provers without a substantial effort. As we have said before, the expressivity of HOL is larger than the one of ACL2, which means that our use cases will restrict to their common subset. These tools (ACL2, in our case) now could use the statements transferred from the Isabelle formal development as a guideline to achieve a similar formalization to the original one.

In this report, we will present and translate two different case uses; a first one based on lists, which we will use to introduce the architecture and its different languages and formalisms, and then a complex development (with more than 6.000 lines of Isabelle code), the Isabelle/HOL formalization of an algorithm computing the diagonal form a matrix by only performing elementary operations, a previous step to the computation of its Smith normal form.

It is worth noting that the concrete representation of data types (in our case, lists, represented in our Isabelle sources by an inductive datatype, or matrices, represented in our Isabelle sources by finite functions over pairs of naturals) is not transferred from one setting (Isabelle/HOL) into the others (XLL, ACL2 or Ecore), since we prefer to leave to the user the task of choosing a suitable representation of the data types in each setting or language; the object oriented setting in Ecore probably demands a different definition of the matrix data type than Isabelle or ACL2.

In order to build the different translations, one could think of using ad-hoc programs, in the form of code snippets in different programming languages, such as ML (the underlying language in which Isabelle is implemented) or Common Lisp (the language in which ACL2 is developed, and from which is a subset). At this point, we decided to make use primarily of *XML* technology to produce the different translations involved. Isabelle already offers support for generating and reading XML files *wrt* an *schema*. Ecore also has its own *schema* and good support for *XML*, even if we did not find an approved schema for OCL, the restriction language. The generic XLL schema served us as a source for both generating ACL2 and Ecore + OCL. We created these schemas and translations for our main use case, the formalization of a diagonal form of matrices, but they can be applied to a broader range of use cases (for instance, our introductory example based on lists); even more, we claim that the XLL schema used for the generation of Ecore, OCL and ACL2 could be also used to generate specifications in other theorem provers (as, for instance, Coq).

This report is divided in five sections. In Section 2 we introduce the tools we are using in our later development (i.e., Isabelle/HOL, Ecore and ACL2). Then, in Section 3 we describe the different transformations and intermediary languages that we have introduced to port theories among the different proving and specification tools. An introductory use case based on lists will be presented. Then, in Section 4 we present the Isabelle/HOL formalization of the diagonal form of matrices, and how it is transferred to both Ecore and ACL2. We pay special attention to the intermediary steps in the setting since they should provide useful information on the applicability of the tool.We also present the resulting specification in Ecore, as well as the ACL2 theory obtained. Finally, in Section 5 we present some works in the field and some proposed enhancements of the technology presented in the paper.

2 Tools description

2.1 Isabelle/HOL

Isabelle [5] is a generic theorem prover (in the sense that different logics can be implemented on top of it). It is programmed in *ML*, a well-known funcional programming language. The Isabelle core (known as Isabelle/Pure, o metalogic) is composed by basic inference rules that represent a fragment of higher-order logic, in the spirit developed by Alonzo Church [2], also known as *simple type* theory.

A detailed description of the metalogic can be found in [7]. We will not get here into the details, but point out that it is based in two main components:

- A *type system*, based on non-empty types, and function types. One of these types is a type called *prop*, which contains the propositions that can be expressed in the system. In particular, it contains two constants, true ($\top$) and false ($\bot$).

– A set of *inference rules* which act over terms of type *prop*, and that express the properties of the connectors of the metalogic. These are $\phi \Longrightarrow \psi$ (which is the equivalent to 'ϕ implies ψ'), the universal quantifier $\bigwedge$, such that $\bigwedge x.\phi$ is the equivalent to 'for all x, ϕ is true' and the equality $a \equiv b$.[1]

The way to define functions in the system is as follows. If for every constant x of a given type σ we assign a value $b(x)$ of a type τ, the λ-abstraction $\lambda x{:}\,\sigma.b(x)$ denotes the function of type $\sigma \to \tau$, which maps each given x to $b(x)$.

On top of this metalogic different logics can be implemented. For instance, the Isabelle standard distribution contains implementations of first-order logic, Zermelo-Fraenkel set theory or logic of computable functions. It also contains an implementation of higher-order logic, in which we will focus, since our posterior developments will be mainly carried out on top of it (at some point we will also revisit the metalogic). HOL is the most widely used logical setting by the Isabelle community, so that usually Isabelle/HOL is commonly referred as Isabelle. Its expressiveness has been helpful to formalize relevant results in diverse fields, from software and hardware verification (for instance, the seL4 project on the formal verification of an operating system kernel [4]) to mathematical foundations (as, for instance, the Basic Perturbation Lemma, an intricate result in Homological Algebra [1]).

In order to define HOL over the metalogic, a *type system*, capturing the properties of the HOL (also known as simple type theory) type system, and a set of axioms (or rules) which define our logical system, are defined.[2]

With respect to the type system, and following the notion of types in the metalogic, new types can be defined as long as they are *not empty* (the user has to prove that new defined types are inhabited). New types can be also defined as *subsets* of existing types. In this way, the *product* of two given types (allowing thus to work with tuples), the addition of two defined types (giving place to the set of direct sums of elements of both types) or types defined by induction can be defined. The system itself includes facilities which ease the definition of new types. In the Isabelle distribution library types representing the natural numbers, integers, reals, complex numbers, polynomials, matrices, rings, vector spaces and almost every mathematical structure in a standard text book of the field are available. One relevant fact for our further development is that these representations are not unique; despite the rudimentary type system behind HOL, different representations (type definitions) can be proposed, for instance, for matrices; a matrix can be considered as a finite function from its coordinates (pairs of naturals) to the matrix underlying type (integers, reals, complex numbers) or as a list of lists of elements of the underlying type. These design choices have a deep influence in the proofs of the results that have to be carried out over the type.

[1] The symbols $\Longrightarrow$, $\bigwedge$ and $\equiv$ are chosen for the metalogic, thus leaving available for the logics implemented on top of the metalogic the most common ones $\longrightarrow$, $\forall$, $=$.

[2] In general, when dealing with theorem provers, the inclusion of *axioms* in the system demands extreme attention, since wrong axioms could lead the system to an inconsistent state where everything could be provable.

The set of rules which HOL incorporates is rather sort; take into account that some of them simply state equivalences between operators in HOL and the corresponding operators in the metalogic:

refl: $t = t$
subst: $[\![s = t ; P\ s]\!] \Longrightarrow P\ t$
ext: $(\bigwedge x.\ f\ x = g\ x) \Longrightarrow (\lambda x.\ f\ x) = (\lambda x.\ g\ x)$
impI: $(P \Longrightarrow Q) \Longrightarrow P \longrightarrow Q$
mp: $[\![P \longrightarrow Q ; P]\!] \Longrightarrow Q$
iff: $(P \longrightarrow Q) \longrightarrow (Q \longrightarrow P) \longrightarrow (Q = P)$
someI: $P\ x \Longrightarrow P\ (\varepsilon x.\ P\ x)$
True_or_False: $(P = True) \vee (P = False)$

From the previous axioms, **ext** expresses extensionality of functions (wrt the universal quantifier $\bigwedge$ of the metalogic). The rule **iff** expresses that formulae logically equivalent are equal. The rule **True_or_False** (also known as law of excluded middle) makes the implemented logic classic. The rule **impI** relates the element of the logic $\longrightarrow$ with the element of the metalogic $\Longrightarrow$. The remaining elements in the logic (the constants *True* and *False*, the connectors $\neg$, $\forall$, $\wedge$, $\vee$, the unique existential $\exists_1, \ldots$) can be defined (without the need of axiomatically including them) from the introduced connectives. For instance, *True* is equal to $(\lambda x.x = x) = (\lambda x.x = x)$ and $\neg$ is defined as $\neg P = (P \longrightarrow \mathit{False})$.

New statements can be introduced in the system by means of idioms such as *lemma* or *theorem*, which admit premises and hypothesis in the form of boolean expressions; their proofs can be carried out in different ways; the traditional style consisted in the successive application of tactics or tacticals (functions mapping the statement to one or various easier statements) that must led to the original premises or to a trivial statement (*True*); thus, proofs were developed in a backward style. Then, the Isar language allows the construction of proofs in a backward or forward style, endorsing the use of an almost natural language, where proofs should remain *human-readable*. Nevertheless, in the rest of the paper we will focus on statements and specifications more than in proofs.

Another tool of the system that will be used in our experiment is the facility to *export* Isabelle files (usually called *theories*) to XML. Along the years, the theorem proving community has become more sensible to the necessity of interaction among theorem proving tools, as we have already highlighted in the introduction, but also among different external tools such as PIDE (Isabelle communicates with JEdit as external editor) or tools for generating documentation (such as facilities to generate Latex and html sources from Isabelle theories). Interoperability requires widely used standards, apart of the traditional functional programming languages and dialects used internally in the theorem provers. To fill this gap, Wenzel developed a tool integrated in Isabelle allowing to generate the XML code of any language primitives (such as definitions, type definitions, types, theorems, lemmas, syntax annotations and so on). This XML is compliant to a certain schema (distributed with Isabelle in a file"isabelle.xsd"). This XML code can be used (and will be used in our experiments) as starting point

for further translations. There is another ongoing project to enhance this tool by means of YXML syntax, where XML trees are untyped (without a compliant DTD) and the user must decide the structure in which they are encoded, but our application did not require that much adaptability. Additional information on these tools, already existing applications and experiments can be found in [31]. The paper itself serves also as a useful introduction to our problem on knowledge transference among different tools, by means of an XML infrastructure.

2.2 Ecore and OCL

Eclipse is an open source software project, for the purpose of providing a highly integrated tool platform. This project supports the development of a platform, or framework, for the implementation of integrated development environments (IDEs) and other applications. The Eclipse framework itself is implemented using Java, but is used also to implement development tools for other languages as well (e.g.,C++, XML...).

The Eclipse Modeling Framework (EMF) is a framework for describing a model and then generating other models or code from it. In fact, with EMF modeling and programming can be considered the same thing. It brings them together as two well-integrated parts of the same job, because EMF unifies three important technologies: Java, XML and UML. Using EMF, one can model an application in a UML class diagram, press a button to obtain an XML Schema with that representation or press another button to generate the Java implementation of the interfaces. An EMF model integrates the three technologies and can be defined using either of them.

The model used to represent models in EMF is called Ecore. Ecore is itself an EMF model, and thus is its own metamodel. It is the center of the EMF world and an Ecore model can be created from any of the three technologies: a UML model, an XML Schema, or Java interfaces. In addition, the reverse transformation is possible: from an Ecore model one can generate a UML model, an XML Schema, Java implementation code and, optionally, other forms of the model.

As we have said before, the "conceptual" model of an application could be represented using Java code, XML Schema or a UML diagram. EMF unifies them using a canonical representation: XMI (XML Metadata Interchange). We have to remark that Java code, XML Schema and UML all carry additional information beyond what is captured in an Ecore model; in fact, Ecore is a small and simplified subset of full UML (full UML supports much more ambitious modeling than the core support in EMF; some of these limitations, as the lack of static methods in Ecore, will be noticed later in our technology).

Thus EMF allows us to have interoperability among those three technologies in an XMI file:

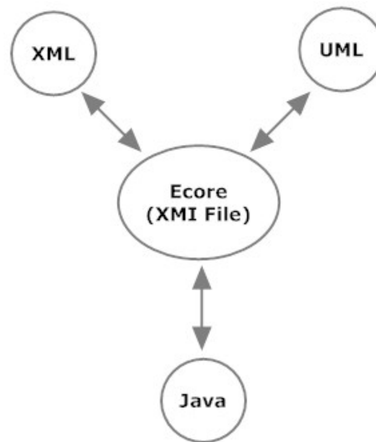


Fig. 1. EMF unifies Java, XML and UML in Ecore (XMI file)

More information about Eclipse project, EMF and Ecore can be found in [6].

We illustrate how Ecore works with an example: given a simple UML class diagram (see figure 2) with two classes, we can obtain the corresponding Ecore model, see its diagram (figure 3) and export it as an Ecore/XMI representation (an XML file, see figure 4).

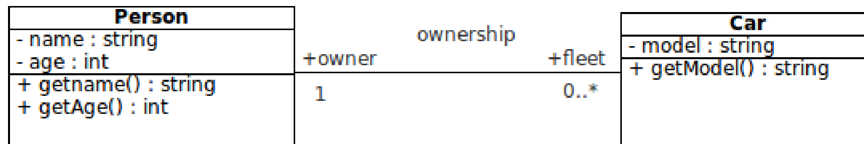


Fig. 2. UML Class Diagram

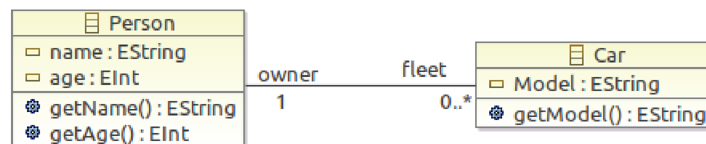


Fig. 3. Ecore Diagram

```
 <?xml version="1.0" encoding="UTF-8"?>
<Ecore:EPackage xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:Ecore="http://www.eclipse.org/emf/2002/Ecore" name="example"
    nsURI="http://example/1.0" nsPrefix="example">
  <eClassifiers xsi:type="Ecore:EClass" name="Person">
    <eOperations name="getName"
          eType="Ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eOperations name="getAge"
          eType="Ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
    <eStructuralFeatures xsi:type="Ecore:EAttribute" name="name"
          eType="Ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="Ecore:EAttribute" name="age"
          eType="Ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
    <eStructuralFeatures xsi:type="Ecore:EReference" name="fleet" upperBound="-1"
          eType="#//Car" eOpposite="#//Car/owner"/>
  </eClassifiers>
  <eClassifiers xsi:type="Ecore:EClass" name="Car">
    <eOperations name="getModel"
          eType="Ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="Ecore:EAttribute" name="Model"
eType="Ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="Ecore:EReference" name="owner" lowerBound="1"
        eType="#//Person" eOpposite="#//Person/fleet"/>
  </eClassifiers>
</Ecore:EPackage>
```

Fig. 4. Ecore XMI

The Object Constraint Language (OCL) is a declarative language for describing rules that apply to a UML model. OCL supplements UML by providing expressions that have neither the ambiguities of natural language nor the inherent difficulty of using complex mathematics, thus OCL is a formal specification language with precise semantics. More information about this technology can be found in [24].

In some situations, UML is not expressive enough, and OCL can be used to state additional properties of UML diagrams. For example, in the figure 2, how can we specify that “a car owner must be at least 18 years old”? The answer is using an OCL invariant:

```
context Car
inv: self.owner.age >= 18
```

Fortunately, an Ecore model can be enriched with OCL restrictions. Even, one can create a dynamic instance of that Ecore model and validate this instance with respect to the model and the OCL restrictions. A tutorial about this possibility (and in general, on working with OCL in Ecore) is found in [8].

2.3 An ACL2 Overview

In this section we present a brief introduction to the ACL2 system. ACL2 stands for “A Computational Logic for an Applicative Common Lisp.” Roughly speaking, ACL2 is a programming language, a logic and a theorem prover. Its programming language is an extension of an applicative subset of Common Lisp [32].

The ACL2 logic describes the programming language, with a formal syntax, axioms and rules of inference: the applicative subset of Common Lisp is a model of the ACL2 logic. Finally, the theorem prover provides support for mechanized reasoning in the logic. Thus, the system constitutes an environment in which programs can be defined and executed, and their properties can be formally specified and proved with the assistance of a theorem prover.

The logic is a first-order logic with equality. The syntax of its terms is that of Common Lisp and therefore uses prefix notation. Formulas are quantifier-free and their variables are considered to be universally quantified. For example, the following formula may be read as "for all natural numbers n and x, with x even and $n > 0$, x^n is even":

```
(defthm evenp-expt
    (implies (and  (natp n) (> n 0) (natp x) (evenp x))
             (evenp (expt x n))))
```

The logic includes axioms for propositional logic and for a number of primitive Common Lisp functions and data types. Rules of inference include those for propositional calculus, equality, instantiation and a principle of proof by induction.

By the *principle of definition*, new function definitions (using `defun`) are admitted as axioms only if there exists an ordinal measure in which the arguments of each recursive call (if any) decrease, thus proving its termination. This ensures that no inconsistencies are introduced by new definitions.

The ACL2 theorem prover is an integrated system of ad hoc proof techniques, including simplification and induction among them. Simplification is a process combining term rewriting with some decision procedures (linear arithmetic, type set reasoner, etc.) Sophisticated heuristics for discovering an (often suitable) induction scheme is one of the key features in ACL2. The command `defthm` starts a proof attempt, and, if it succeeds, the theorem is stored as a rule (in most cases, a conditional rewriting rule). For example the above theorem `evenp-expt`, once proved, would allow the prover to rewrite an instance of the term `(evenp (expt x n))` to the boolean constant `t` (true), provided that the corresponding instantiated conditions of the rule can be established.

The theorem prover is automatic in the sense that, once `defthm` is submitted, the user can no longer interact with the system. However, in some sense, it is interactive. Often, non-trivial results cannot be proved on a first attempt, and then the role of the user is important: she has to guide the prover by a suitable collection of definitions and lemmas, used in subsequent proofs as rewriting rules. These lemmas are suggested by a preconceived hand proof (at a higher level) or by inspection of failed proofs (at a lower level). This kind of interaction is called "The Method" by the authors of the system [33].

A relevant feature of ACL2 is executability: since its axioms and rules of inference describe a subset of Common Lisp, most ground expressions in the logic are directly executable in the host Lisp (as opposed to deducing their values via the axioms). Nevertheless, this simple relationship is complicated by the fact

that not all Common Lisp functions are defined on all inputs: the Common Lisp standard introduces the notion of "intended domain" of a primitive function. Outside this intended domain the behavior of a function is not specified. In contrast, in the ACL2 logic functions are total: that is, every application of a function defined has a completely specified result.

For more information on ACL2, the best reference is [33]. For a detailed and updated description of all the system details, we also recommend visiting the ACL2 home page [34] and the user's manual in it.

3 Translating formal specifications

First of all, we are going to introduce a diagram showing the different ingredients in the development.

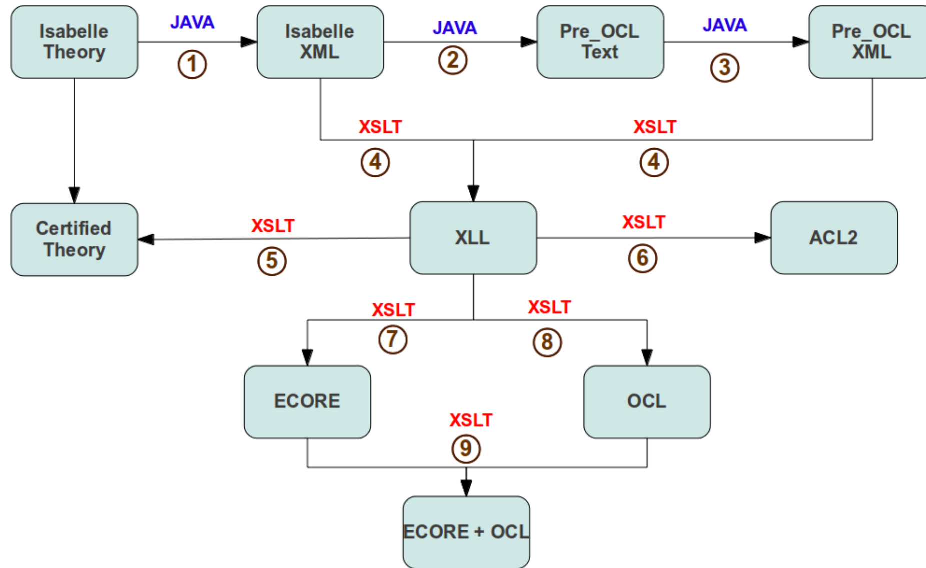


Fig. 5. Architecture of our development

There exist several intermediary steps in the process of translating the theorems' statements from Isabelle to ACL2 and Ecore with OCL restrictions. These steps are exclusively developed using three technologies: Java, XSLT and XProc.

XSLT (Extensible Stylesheet Language Transformations) is a declarative language for transforming XML documents into other XML documents, or other objects such as HTML for web pages, plain text... To introduce it briefly: a XSLT processor takes one or more XML sources, plus one XSLT stylesheet, and processes them with the XSLT template-processing engine (the processor) to

produce an output document. More information on XSLT can be found in [9]. Due to topics as reusability and maintenance, it is a "good practice" to have various small XSLT files instead of one big transformation. For this reason, another technology is used in our development: XProc, which is designed to address the common problem of how to compose XML processes. Many document processing scenarios involve some combination of XML technologies; canonical examples include XInclude, schema validation or transformations. XProc has been specifically designed to allow authors to compose XML processes and share these compositions in a standard way. In our case, we use XProc to make consecutive XSLT transformations[3]. A very nice tutorial of the use of this tool can be found in [11]. Most XProc processors can be downloaded from [23].

Figure 5 shows the necessary steps to transform a suitable Isabelle input to an XLL file (through the steps 1 to 4); in these steps, an intermediary XML language called *Pre_OCL* is created to express the statements of properties; in order to formally prove that this transformation can be reversed, step 5 proves that the statements presented in the XLL document generated through the steps 1 to 4 are equivalent to the original Isabelle statements (inside of Isabelle). Step 6 translates the theorems' statements presented in the XLL to ACL2, obtaining a guideline to formalize the original Isabelle theory in ACL2. Steps 7, 8 (which could be understood as a single step) and 9 translate the XLL specification of types and properties to the Ecore + OCL environment.

3.1 Architecture overview

As we have said in the introduction, the main objective of this work is to port the theorems' statements from a given Isabelle theory to ACL2 and Ecore + OCL restrictions. Our main case study proves that an integer matrix can be diagonalized by elemental operations, as a previous step to obtain its Smith Normal Form.

Nevertheless, our translation process works reasonably well for other theories which do not make use of higher-order statements. For instance, we will present an additional example of theorems about lists. It seems rather possible that, for a given Isabelle theory (restricted to first-order constructions), our XSLT transformations could be minimally adapted in order to properly translate that theory to ACL2 and Ecore with OCL restrictions through an XLL document. In any case, since our starting point are Isabelle theories, whose type system is rather simple (functions and product types, as we presented in Section 2.1) and the output of our technology is Ecore, which is a modeling language with a richer type system, delicate design decisions should be made. Contrarily, ACL2 and OCL are based on first-order predicate logic, whereas Isabelle is based on HOL (Higher-Order Logic). Thus, ACL2 and OCL are less expressive and the translation of statements could give place to some ill-formed statements in the target languages.

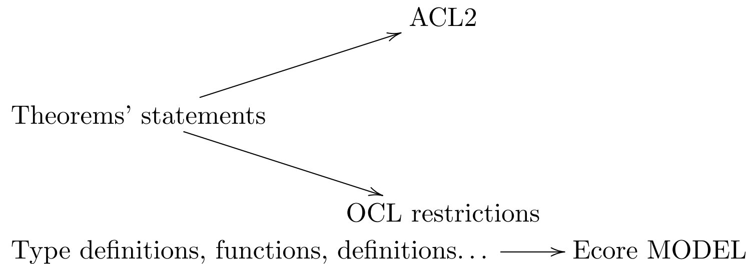
[3] A pipeline of transformations, where the output of the k-th transformation is input of the (k+1)-th one.

In order to get the ACL2 output, we will be mainly interested in the theorems' statements appearing in the XLL file. ACL2 is untyped, thus to introduce operations in ACL2 we only need to know its arity, which can be directly inferred from the statements. Nevertheless, ACL2 usually introduces predicates to state that free variables or constants appearing in operations and formulas belong to the appropriate sets (for instance, that an index for a list or a matrix position is a natural number); these premises and belonging predicates will be also generated by our automatic translation.

We insist on the idea that data types representation is not translated from Isabelle to ACL2 since we prefer to preserve the internal design decisions of ACL2 to this language (for instance, our matrix representation in this Isabelle development, by means of finite functions over pairs of naturals would not be natural in ACL2, where a representation by lists, or lists of lists seems much more appropriate).

On the other hand, modeling in Ecore requires information about data types (in order to convert each type to a class) and about the types and arity of functions/definitions used in Isabelle (in order to convert them to class methods). Let us remark that we obtain an Ecore **model**, so the information about implementation or representation from Isabelle is neither required for this transformation (nor for ACL2). To complete the Ecore model, we also need theorems' statements, since they will be translated as OCL restrictions in that model.

To sum up:



Starting from an Isabelle theory we will translate it to ACL2 and Ecore + OCL by means of a common XML language (XLL), trying to minimize the "ad-hoc" translations (step 6 or steps 7 to 9). Firstly, some XML files are generated with the necessary information, that is, information about types (necessary for modeling in Ecore) and information about theorems' statements (necessary for translating them to ACL2 and OCL). Once all this information is translated into XML files, the Isabelle theory is no longer necessary; we just keep it to complete some further tests about the relationship between the generted statements in some of the intermediary languages developed (Pre_OCL and XLL) and the original theorems. One of the key products in the development process is the Pre_OCL file (step 3). It contains an XML dialect with the theorems' statements from the input Isabelle theory, but now written in a "raw" Isabelle style, where operations appear in prefix notation, most of the abundant Isabelle syntax translations are avoided (for instance, Isabelle pretty syntax for lists $[x, y]$ would appear in terms of the primitive Isabelle list constructors), and additionally fol-

lowing the XML structure presented in Appendix 6.2. This Pre_OCL XML file is completed with the specification of the datatypes used in the Isabelle theory (typedefs, definitions, functions. . .) to obtain the XLL document, which is the keystone of our development.

We will explain the previous steps with the help of a simple example on lists.

3.2 Step 1: From Isabelle Theory to Isabelle XML

The first step that we apply consists in generating XML files containing all the information that we need in order to be exported to the other systems in later stages. These XML files are generated from Isabelle standard elements (statements of lemmas, definitions of types, functions, operations. . .) and follow the XSD Schema presented in Appendix 6.1. From here on, we will refer to them as Isabelle XML files. The fact of having the information in XML trees provides great advantages, for example:

- Two theorems which are equal (except for Isabelle syntactic translations or pretty syntax artifacts), give place to the same XML file. For example, "$\forall n.n > 1 \longrightarrow n > 0$" and "$\forall n > 1.\ n > 0$".
- The XML files generated can be processed using XLST.
- With the tree structure of an XML file, it is easier to see the dependencies that there exist among, for example, a function and its input arguments. The ideal would be to have an XML tree in which functions have as children their input parameters. Unfortunately, the XML files generated from Isabelle using ML functions don't keep this property, although it will be achieved in following steps. The reason why the XML files don't preserve this property is that, internally, Isabelle transforms multi-argumental functions to a composition of several single-argumental functions. For instance, the XML file generated from Isabelle user input $a + b = (c :: int)$ is as follows:

```
<App>
  <App>
    <Const name="HOL.eq">
      <Type name="fun">
        <Type name="Int.int"/>
        <Type name="fun">
          <Type name="Int.int"/>
          <Type name="HOL.bool"/>
        </Type>
      </Type>
    </Const>
    <App>
      <App>
        <Const name="Groups.plus_class.plus">
          <Type name="fun"/>
        </Const>
        <Var name="a"/>
      </App>
```

```
    <Var name="b"/>
  </App>
 </App>
 <Var name="c"/>
</App>
```

The previous tree is obtained using ML functions already available in the Isabelle distribution (we need to write three ML lines for each statement that we want to convert to one XML file). Every single theorem, datatype definition, or operation in Isabelle gives place to a different XML file. In general, it will be necessary to convert a multitude of theorems' statements, data type information, definitions... to XML, so we automated this process making a Java program to perform this task. This program gets as input a text file which contains the path of the Isabelle theory that has to be processed and the list with the names of the Isabelle elements that have to be translated to XML.

The program works as follows:

1. First, it reads from the text file the path of the Isabelle theory and a list of names of Isabelle locales, classes, typedefs, datatypes, partial functions, functions, definitions and theorems to process.
2. A backup of the Isabelle theory file is made in a temp folder.
3. The program inserts at the end of the theory file the necessary ML code to generate the XML files.
4. It makes a new directory called XML in the theory path. Moreover, this directory contains two folders: the first one is named CLASS and the second one THEOREMS. In the folder CLASS the algorithm generates the XML files obtained from locales, classes, typedefs, datatypes, partial functions, functions and definitions; i.e., the information about the data types and their operations. As its own name shows, in the folder THEOREMS the XML files containing theorems' statements will be placed.
5. Once we have the theory file with the ML code inserted, we need to process it in Isabelle to generate the XML files. The program invokes to the Isabelle process over the file with the ML commands, by means of the following expect script [12].
6. Finally, the input theory file is kept in its initial state, i.e., without the ML code added at the end of it. This is possible thanks to the temporary copy created before.

The input text file, where the path of the Isabelle theory and the list of elements to be processed can be found, has the following structure:

```
/directory/theory_name.thy
use_locale:
use_class:
use_typedef:
use_datatype:
use_partial_function:
```

```
use_fun:
use_definition:
use_thm:
```

We also have to remark a detail about Isabelle's syntax. In Isabelle, theorems' statements can include both free variables and quantified variables. For example:

```
lemma free_variables: "(x::int)+y=y+x"
```

In this theorem, the variables x and y are free, i.e., they can be later instantiated with any value of the appropriate type (provided that there are no local variables created previously in our theory with that names). Therefore, the theorem is equivalent to:

```
lemma free_variables2: "(z::int)+t=t+z"
```

A "similar" theorem can be stated quantifying variables x and y:

```
lemma quantified_variables: "∀x::int.∀y.x+y=y+x"
```

In addition, the equivalence between both theorems can be proved:

```
lemma eq_free_quantified:
  "((z::int)+t = t+z) ⟷ (∀x y::int. x+y=y+x)"
  by auto
```

It is important to stress that a free variable is not a quantified variable. This is a sensitive topic; for further discussion on it you can visit the following thread in the Isabelle mailing list [14] and [15].

OCL does not allow the use of free variables. Therefore, we will (universally) quantify free variables appearing in the Isabelle theorems' statements that are to process. In order to do that, we make use of a ML function of the Isabelle library: *forall_intr_vars*. This function quantifies free variables with the Universal Quantifier of the Isabelle metalogic ($\bigwedge$). As we are working in HOL, we define an additional function which translates theorems from the metalogic (or Pure) to HOL. This function will be in charge of replacing $\bigwedge$ by the HOL equivalent $\forall$ (among other changes). The previous changes will turn an Isbelle statement like "$x + y = y + x$" into its equivalent "$\forall x.\forall y.x + y = y + x$". The ML function performing such conversion is the following:

```
ML{*fun atomize_thm thm =
        Thm.equal_elim (Object_Logic.atomize (cprop_of thm)) thm*}
```

We present an example of the ML code which is inserted into the original Isabelle file in orer to generate the XML file corresponding to a theorem (datatype definitions or operations are generated with a similar code script):

```
ML{*val xml_tree = XML_Syntax.xml_of_term
      (prop_of (atomize_thm (forall_intr_vars @{thm "theorem_name"})))*}
ML{*val output_path = Path.explode "directory/XML/Theorems/theorem.xml"*}
ML{*XML_Syntax.write_to_file output_path "term" xml_tree*}
```

The folder CLASS with the previously generated XML files will be used to generate the part of the XLL file including the information about data types and their operations. The folder THEOREMS will be used to generate the Pre_OCL XML file, which is later processed to produce the part of the XLL file including the properties of data types and their operations. Every Isabelle element (theorems, data types, operations) is generated to a different XML file, so we are losing relevant information about the possible dependencies among them. In order to store the order in which theorems have been proved in the original Isabelle theory (which will be later useful for translating them to XLL in the very same order) a file "index_theorems.txt" that contains the theorems' names (separated by blank spaces) in a similar order as they were introduced and proved in the Isabelle input theory is automatically generated. In addition, an XML file "index_class.xml" is also created with the names of the XML files placed in the folder CLASS (*i.e.*, a list of the data types and operations that have been generated). This file is later processed by an XSLT transformation to produce the XLL part corresponding to data types' and operation's declarations in the very same order as they were introduced in the Isabelle theory (respecting the possible dependencies among them).

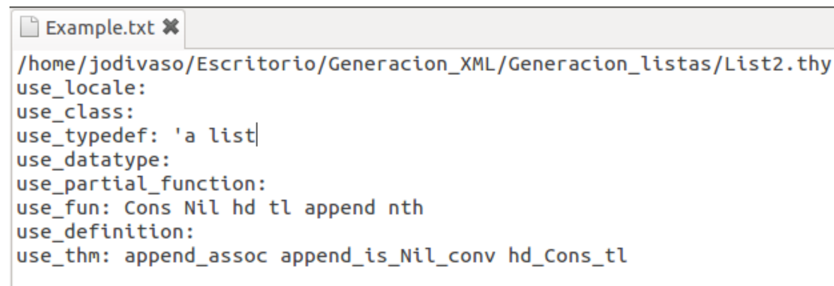
Example on lists We start from a very simple Isabelle theory file about lists. The results have been directly obtained from the List theory in the Isabelle library and the code of this simple theory is the following:

```
theory List2
imports Presburger
begin

datatype 'a list =
    Nil    ("[]")
  | Cons 'a  "'a list"    (infixr "#" 65)

primrec
  hd :: "'a list ⇒ 'a" where
  "hd (x # xs) = x"

primrec
  tl :: "'a list ⇒ 'a list" where
    "tl [] = []"
  | "tl (x # xs) = xs"

primrec
  append :: "'a list ⇒ 'a list ⇒'a list" (infixr "@" 65) where
    append_Nil:"[] @ ys = ys"
  | append_Cons: "(x#xs) @ ys = x # xs @ ys"

primrec
  nth :: "'a list => nat => 'a" (infixl "!" 100) where
  nth_Cons: "(x # xs) ! n = (case n of 0 ⇒ x | Suc k ⇒ xs ! k)"
```

```
lemma append_assoc [simp]: "(xs @ ys) @ zs = xs @ (ys @ zs)"
by (induct xs) auto

lemma append_is_Nil_conv [iff]: "(xs @ ys = []) = (xs = [] ∧ ys = [])"
by (induct xs) auto

lemma hd_Cons_tl [simp,no_atp]: "xs ≠ [] ==> hd xs # tl xs = xs"
by (induct xs) auto

end
```

We create a text file with the structure presented previously selecting what information we want to process, that is, the data types, statements, functions or definitions from which we will generate XML files.

In our example we make use of the type definition *'a list*, of functions (or constants, which are treated as functions with 0 arguments) *Cons, Nil, hd, tl, append* and *nth*, and of three theorems named *append_assoc, append_is_Nil_conv* and *hd_Cons_tl*. The input text file created to process them is shown in Figure 6.



```
Example.txt
/home/jodivaso/Escritorio/Generacion_XML/Generacion_listas/List2.thy
use_locale:
use_class:
use_typedef: 'a list
use_datatype:
use_partial_function:
use_fun: Cons Nil hd tl append nth
use_definition:
use_thm: append_assoc append_is_Nil_conv hd_Cons_tl
```

Fig. 6. Example of text file used to generate XML files from Isabelle

The first line of the file contains the Isabelle path of the theory file. Once this file has been created, by executing the Java program the following process starts: First, the program demands from the user the text file with the enumeration of Isabelle elements that are to be generated (see Figure 7).

Then, the program automatically creates two folders (Class and Theorems) where the XML files are saved (see Figure 8).

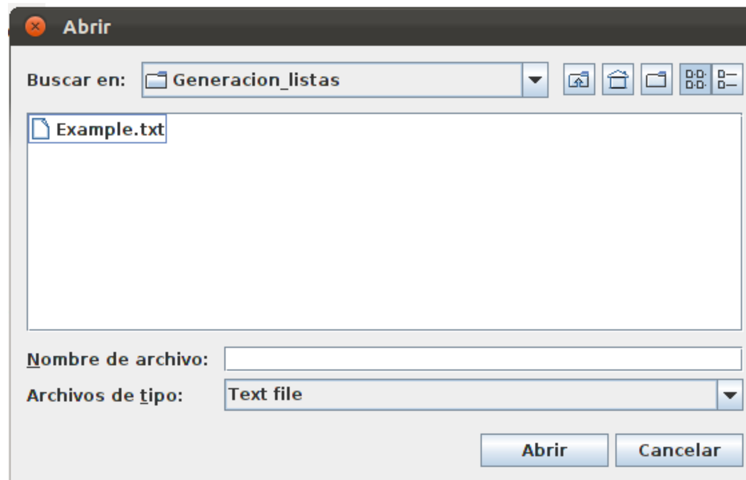


Fig. 7. We have to select the text file

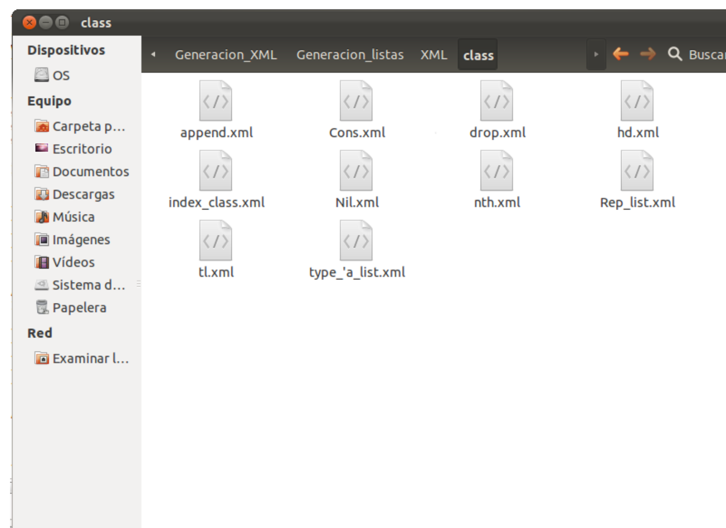


Fig. 8. Example of folder Class obtained with generated XML files

In these folders there can be found, respectively, the files *index_class.xml* and *index_theorems.txt* explained before. An example of the contents of these files (concretely, about the XML generated from theorem named *hd_Cons_tl*), is given in appendix 6.5.

3.3 Step 2: From Isabelle XML to Pre_OCL text file

The collection of Isabelle XML files generated from the statements of properties (data types and definitions are treated in a different way) in the previous step (one file for each statement), is processed in order to obtain a version which is easier to translate to ACL2 and OCL (that will be previously fitted in the XLL file).

In this step a text file is generated: the *Pre_OCL text file*. In this file, the Isabelle theorems' statements, written in prefix notation, are stored (in the Isabelle input files, operations with their prefix and infix notations can be found simultaneously). This conversion is facilitated thanks to the XML files generated in the previous step, in which operations already appear in prefix notation. Nevertheless, we have to pay attention to some additional details, for example, the presence of λ-abstractions and bound variables in universal and existential quantifiers (see [16] for a detailed explanation of bound variables).

In the XML files obtained in step 1 each (universally or existentially) quantified variables are abstracted using a λ-abstraction and from then on, these variables are referred to with a De Bruijn index. Each De Bruijn index is a natural number that represents an occurrence of a variable in a λ-term, and denotes the number of binders that are in scope between that occurrence and its corresponding binder (see [25] for more details). In Isabelle, the notation "Bound n" is used to refer to the De Bruijn index "n". For instance, given the following Isabelle user input:

```
∀x. x>0 ⟶ (∃y. y<0 ∧ x+y=0)
```

Internally, Isabelle transforms the user input into an expression which is similar (except for some details which have been omitted to clarify the example) to the following one:

```
HOL.All (λx. (Bound 0)>0 ⟶ (HOL.Ex (λy. (Bound 0)<0
                              ∧ (Bound 1)+(Bound 0)=0)))
```

The use and detection of these bound variables in the XML files generated from Isabelle is the main reason to make this step using Java instead of XSLT (the translation that retrieves the variable names from the bounded ones is easier using Java).

This *Pre_OCL text file* is an intermediary step to obtain the *Pre_OCL XML file* which will be part of the XLL document.

Example on lists From the XML files generated in step 1 by using a Java program, the text file Pre_OCL about lists is obtained.

When the program is executed, it will demand from the user the location of the Isabelle theory file[4] (see Figure 9).

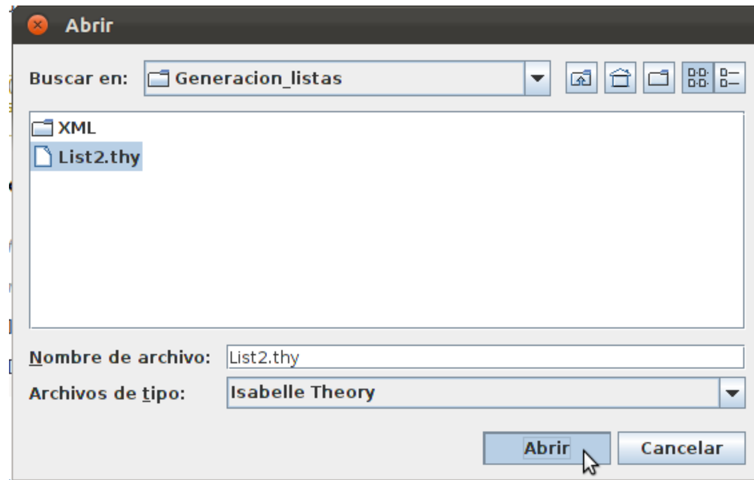


Fig. 9. We select the theory file to obtain the prefix notation of theorems in a text file

Internally, the program will open the file "index_theorems.txt" to know the correct order in which statements have to be processed.

The result will be stored in a text file named *"List2.txt"*, created in the same path of the original theory. It can be found in Appendix 6.9. Tt contains the statements of the three theorems selected in the previous step, but now written in prefix notation. For example, the theorem *append_assoc* in the original input Isabelle file had the following statement (and proof):

```
lemma append_assoc [simp]: "(xs @ ys) @ zs = xs @ (ys @ zs)"
by (induct xs) auto
```

Its statement, in prefix version, generated after step 2 in file *List2.txt* looks as follows:

```
append_assoc (HOL.All (\<lambda>xs::'a List2.list. (HOL.All
(\<lambda>ys::'a List2.list. (HOL.All (\<lambda>zs::'a List2.list.
(HOL.eq (List2.append (List2.append xs ys) zs) (List2.append xs
(List2.append ys zs)))))))))
```

In the next step, this prefix version will be stored in a XML file.

[4] The transformations are made from the XML files generated in step 1. For that, the program achieves the path of the XML files easily: they will be placed in a subfolder "THEOREMS" in the path of the original theory.

3.4 Step 3: From Pre_OCL text file to Pre_OCL XML

Once we have the theorems' statements of the Isabelle theory in a prefix notation, we translate them to a rather simple XML file. The XSD Schema of this XML is found in Appendix 6.2. The original XML files generated in step 1 have several redundant information for our purposes (for example, the type of every operation, function, constant or parameter appears explicitly for each of their occurrences, even in the same statement, even though they can be inferred or stored in their first occurrence in a statement) and they are difficult to be processed (for example, functions, in the XML trees do not have their parameters as branches, because of the definition of the Schema used in step 1). With the XML Schema introduced in this step, we, among other things, simplify these XML files, keeping the indispensable data for the translation to XLL (as a previous step to the transformation to OCL and ACL2). In general terms, each theorem is presented in this XML following a tree structure, separating the quantifiers (universal and existential) from its statement, operations (whose children are their parameters, which can be variables, other quantifiers or other operations) and variables.

The process is done with a Java program which takes the Pre_OCL text file generated in step 2 as input. The output will be an XML file (from here on, we will refer to it as Pre_OCL XML file) compliant with the schema presented in Appendix 6.2. This Pre_OCL XML file has the same theorems that the original Isabelle input file (and that the Pre_OCL text file) but following the prefix notation presented in the Pre_OCL text file and with a tree structure, as presented in the Schema in Appendix 6.2. It is worth noting that there are no syntactic differences between the statements in the Pre_OCL text file and the ones in the Pre_OCL XML file, but just an arrangement of the statements with respect to the Schema in Appendix 6.2.

The XML file generated in this step is a key element of the architecture, since we will use it (almost without changes) to obtain the part of the XLL file presenting theorems' statements, and then the OCL restrictions and ACL2 statements.

Example on lists Following with the example about lists, we use a Java program to generate the XML file that follows our schema. When we execute it, we have to select the *List2.txt* file created before. After that, we select the location where the Pre_OCL XML file will be saved (see Figure 10).

Then, a Pre_OCL XML file is obtained where the statements of *append_assoc* *append_is_Nil_conv* and *hd_Cons_tl* are presented in an XML structure following our XSD Schema. This file is shown in Appendix 6.6.

3.5 Step 4: From Isabelle XML and Pre_OCL XML to XLL

This step is labeled as step 4 in Figure 5. In this step we introduce a new XML Schema, named XLL. This XML file can be understood as the keystone for the rest of the translation process; from here we will be able to define direct

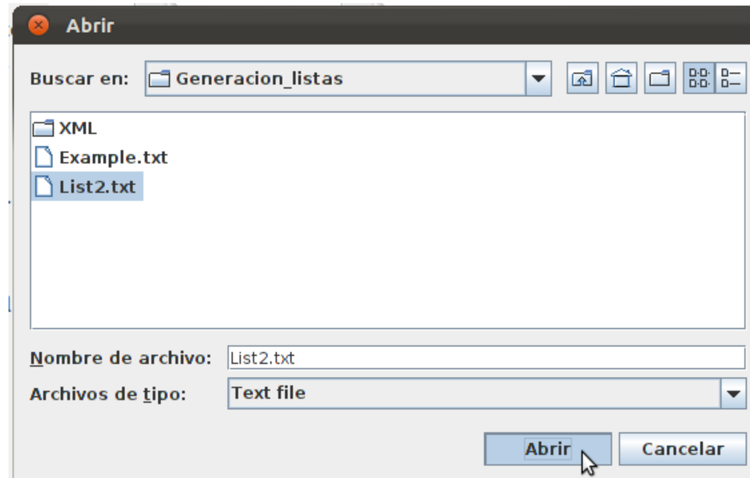


Fig. 10. We have to select the text file created in step 2

translations, by means of exclusively XSLT technology, to ACL2, Ecore plus OCL, and Isabelle.

This schema (and the XML files complaint to it) is also the most complex piece of our work; it contains information referred to the data types involved in the use cases, the operations and definitions associated to them (these two pieces of information are obtained from the Isabelle XML files) and the statements of the properties (or theorems), as processed in the Pre_OCL XML file. The XLL schema is shown in Appendix 6.7.

To sum up, this XLL schema consists of two different but associated parts.

1. A specification of datatypes (or classes), including for each datatype a name plus a family of operators (or methods) in an object oriented style; this part is essentially imported from the Isabelle XML files obtainend in step 1.
2. A set of logical statements, expressing properties of the datatypes involved. This part is directly obtained importing the file Pre_OCL generated in step 3.

In fact, the XLL schema is a combination of the Pre_OCL schema completed with information about data types (Pre_OCL is only obtained as a previous step to generate this XLL file). Additionally, the schema performs operations checking that the statements of the properties contain operations that exclusively appear on the XLL file itself (in the part about specification); our intention is to ensure that the properties stated in the file are referred to a certain context (a set of data types and operations). This idea has been obtained from the notion of "context" in OCL, where properties must be always stated with respect to some set of classes (and their operations), which provides a necessary context in which the statement holds (or not, depending on the underlying model). Actually, a similar

behavior is obtained in Isabelle, where it is always checked that the statements of lemmas or theorems are always performed over types and operations previously introduced, and respect the arities and types of their parameters.

These additional cross-checks, which are natural for compilers and syntax checkers, introduce additional complexities in the XLL schema, since the schema itself needs to know the name of the operations that it contains in the specification of datatypes, and then must be capable of verifying that the only operations that appear in the statements are these ones. This way, the two parts of the XLL are connected.

From a strict point of view, a XLL file is built exclusively from Isabelle XML files (the part about types and operations' specifications is obtained directly from the input XML files, and the set of logical statements is the Pre_OCL file which has been generated from Isabelle XML files too). Nevertheless, the link between the methods' names in the datatypes part and operations' names in the logical expressions can't be checked using the Isabelle XML files due to two reasons:

1. For each Isabelle element (theorem, class, datatype, definition...) a different Isabelle XML file has been created in step 1, so we have to unify all information in a single file (the XLL file) to check the coherence of methods' names.
2. From a technical point of view, recovering methods' names from the Isabelle XML files requires complex XPath commands; standard XSD schemas only allow a simple subset of XPath commands to be used in [35]. Therefore, the restriction about the coherence of methods' names couldn't be checked using the original Isabelle XML files. However, our XLL schema is simpler than the Isabelle one (used in step 1) and obtaining methods' names requires only basic Xpath expressions which are allowed in the standard XSD (in other words, restrictions about the correctness of operations' names can be checked).

XLL does no aim at creating a new description of a logical language, but at being a simple language in which types, operations and logical statements over them (in a typed first-order logical language) can be expressed. We have been capable of translating its information to ACL2, Ecore plus OCL, and also to Isabelle. Probably, other language targets (supporting at least this logic) could be also reached without an extraordinary effort.

The four XSLT templates which transform the Isabelle XML files to an XLL document are the following. The first three ones construct the part about specification and the last one joins the generated specification with the Pre_OCL XML file (which contains the statements).

1. **Concatenate.xsl:** We start from the file index_class.xml generated in the step 1. This file is placed in the folder Class created in that step and it contains the list with all necessary elements to generate the specification (datatype information, functions, definitions...). It is, an XML file in order to be processed with XSLT. This file has the following simple structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<files>
  <file name="datatype_name.xml"/>
  <file name="function_name.xml"/>
  <file name="definition_name.xml"/>
  ...
</files>
```

What this template do, is to concatenate the files pointed in *index_class.xml*, creating a single XML file with all information.

2. **Create_xll.xsl:** From the XML file with all information this transformation generates the specification part of the XLL (a model with the classes and the methods with their input and output parameters). As we have explained before, the specification part of XLL allows us to create an object oriented model which will be later converted to an Ecore model, which is also an object oriented model. For that reason, for each function or definition that we have to process, we have to choose what class must belong to. The universal decision that we have made is that each method belongs to the class that represents the datatype of its first parameter. By this way, an operation of type *int matrix* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *int matrix* in Isabelle, e.g., *interchange_columns A m n* will be converted in our XLL (and then, in the Ecore model too) to a method belonging to the class "matrix" named *interchange_columns* with two input parameters of type nat which return will be a matrix. That is, the method will be (in UML notation) *interchange_columns(nat,nat):matrix*.
   We also have to remark the importance of keeping the return type. In a pure object oriented modeling, it looks natural to choose the return type of the function *interchange_columns* as void. By this way, *A.interchange_columns(1,2)* would interchange the columns 1 and 2 of matrix A. However, in order to keep the coherence with the Isabelle functions and, above all, by simplicity when we translate the theorems to OCL, we keep the same return type in Isabelle and in XLL (and then, in Ecore too). So, the expected behaviour of *A.interchange_columns(1,2)* would be to return a new matrix equal to the A matrix but with columns 1 and 2 interchanged. It can be noted that A matrix would not be modified. Behavior is similar, for example, to the substring method of the Java class String (it is not static but it doesn't modify the object).
   Furthermore, if this transformation finds an Isabelle function whose first parameter belongs to a type that doesn't match with a class in the XLL specification, then the corresponding class will be created and the function would be added to it.
   This XSLT transformation also introduces two classes into the specification:
   - **Program Logic:** This class contains the basic logical operations of our setting (HOL), such as conjunction, disjunction, implication and so on.
   - **Arith:** This class contains the basic arithmetic operations, such as multiplication, plus, less or equal...

   These two classes are created to guarantee the required coherence between functions that appears in the theorems and methods' names in the part

about specification. Nevertheless, these classes won't be translated to Ecore, because their methods will correspond to OCL constants (for example, the conjunction will be converted in OCL syntax to "and"...).

3. **Grouping.xsl:** This is a technical transformation. After applying the previous XSLT transformation, it is possible that there exist several class with the same name. This transformation groups them into a unique class, and then the part of specification of XLL will be completed.
4. **Join.xsl:** This transformacion is in charge of generating the final XLL file which will contain the specification and the theorems. For that, the output of the previous transformation (the specification) is joined to the Pre_OCL XML file obtained in step 4 (which contains the theorems), giving rise to the XLL document.

Example on lists Applying the XSLT transformations over the *index_class.xml* file generated in step 1, the XLL document of our example on lists is obtained. This document takes up about 350 code lines and it is shown in Appendix 6.8.

3.6 Step 5: From XLL to the original Isabelle theory

This step is labeled as step 5 in Figure 5. The intention is to show that the statements of the theorems presented in the XLL document generated in previous step can be moved backwards to statements in Isabelle that are (automatically) provable equal to the original ones (to the statements of the original Isabelle theory file). Basically, this step serves for the purpose of proving that the statements that we have processed in steps 1 through 4 are only syntactic translations of the Isabelle original ones (these translations will be shown useful to ease further translations to ACL2 and OCL, since they are closer to the syntax of these target languages).

This step also shows that the possibility of translating the XLL language to different target languages (in this case, Isabelle), as we had already announced before. The result of this step is an Isabelle theory, named *Certified Theory*.

As we have already said, this file is an Isabelle theory itself, automatically generated from the XLL document, which checks that the translation from each Isabelle statement to prefix notation has been done properly (*i.e.*, the obtained statement is provable equal, by Isabelle, to the original one in the input file). This theory imports the original theory and proves that theorems (the original theorems and their transformations in prefix notation stored in the XLL document) are equivalent. Because of the introduction of universally quantified variables in statements in step 1, we translate the original statements (which usually are expressed within HOL syntax) and the ones obtained in this step (where metalogic connectives have been used) to the same logical setting (HOL or Isabelle/Pure) before comparing them, to ease the proofs of their equivalence. To carry out the translation of theorems from HOL to the metalogic, we make use of some ML functions of the Isabelle ML layer: *Thm.eq_thm_prop* (to compare if two theorems are equivalent), *Object_Logic.rulify* (to translate the theorem connectives

to the ones of the metalogic) and *atomize_thm* (to translate the theorem connectives to the ones of HOL). Last two functions are necessary because the function *Thm.eq_thm_prop* compares theorems in the same logic (metalogic or HOL)[5].

Therefore, for each theorem statement two ML sentences will be generated (actually, both of them perform the same comparison, one in the metalogic, the other one in HOL, so one of them would be enough to prove that the results are equivalent):

```
ML{*Thm.eq_thm_prop (Object_Logic.rulify @{thm length_append_prefix},
    Object_Logic.rulify @{thm length_append})*}
ML{*Thm.eq_thm_prop (@{thm length_append_prefix},
    atomize_thm (forall_intr_vars @{thm length_append}))*}
```

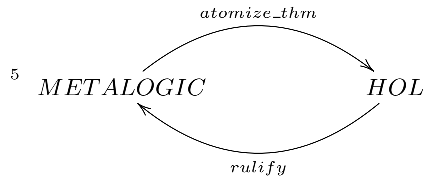
When the Isabelle file automatically generated (that we have named Certified Theory) is processed, those ML sentences return the value *True*. An additional validation is performed in the file Certified Theory, checking that every theorem obtained in prefix notation can be trivially proved using the input theorem[6]. For example, the following Isabelle statement, automatically generated in the file Certified Theory, proves that the prefix notation of lemma *length_rev_prefix* is provable from the input result (*length_rev_prefix*):

```
lemma length_rev_prefix:
shows "(HOL.All (\<lambda>xs::'a List2.list. (HOL.eq
(Nat.size_class.size (List2.rev xs)) (Nat.size_class.size xs))))"
    using length_rev by fast
```

To sum up, this step ensures the correctness of the transformations made to obtain the XLL document (steps 1 through 4), in the sense of that we are able to demonstrate (using Isabelle) the equivalence between the statements presented in the XLL document and the original ones presented in the Isabelle theory.

Example on lists Following our example about lists, we show the result that has been produced by the XSLT transformations labeled as step 5 in Figure 5.

We obtain a new Isabelle theory file (the Certified Theory), named in this case as "List2_certified.thy'" and it can be seen in Appendix 6.10. If it is processed with Isabelle one could check that theorems *append_assoc append_is_Nil_conv* and *hd_Cons_tl* of the original theory are proved to be equivalent to the new ones generated in prefix notation (see Figure 11); we consider this as a certificate of the preservation of the original statements by our Java and XSLT transformations so far.

5 

6 This is not necessary because the equivalence between both theorems has been already proved

Fig. 11. Certified Theory List processed

3.7 Step 6: From XLL to ACL2

This step is labeled as step 6 in Figure 5. In this step we already reach one of the target languages of our architecture, ACL2. By means of a XSLT translation, we obtain a list of ACL2 elements (including terms and theorems' statements); this collection of statements should serve as a guideline to achieve a similar formalization to the original Isabelle theory in ACL2. The starting point is our language XLL, and by means of a XSLT transformation, ACL2 code is obtained.

ACL2 syntax uses prefix notation for operations, constants or functions. For this reason, we have converted (in steps 1 to 3) the Isabelle notation to prefix one; additionally, we have organized the statements in an XML tree (already introduce in Pre_OCL XML, also used in a very similar way in XLL) in order ease its processing using XSLT templates. In our files Pre_OCL XML and XLL, each function has as children its own arguments (this fact doesn't occur in the XML files following the schema "Isabelle.xsd"), and thus information is organized following a prefix pattern. From it, we can achieve prefix notation in ACL2 (or also in OCL) almost directly.

A pair of syntactic details are relevant for our transformations:

1. ACL2 gives a special meaning to the apostrophe (') character, and therefore it cannot be included in theorems' names. In order to remove it, we rename each appearance of the apostrophe to *_bis*.

2. In ACL2, the identifier *t* stands for the constant *true*. On the contrary, in Isabelle, *t* does not stand for any particular constant. Thus, we had to rename appearance of variables labeled as *t* in the XLL language to a fresh variable name, as *t1*.

For each function that appears in any theorems statement, the XSLT transformation automatically generates an (empty) definition of function in ACL2, with similar name (except for the previous syntactic preventions) and arity to the one appearing in XLL, following the next structure:

```
(defun function_name (x1 x2 x3 )
    (declare (ignore x1 x2 x3 ))
    nil)
```

Our intention is that the ACL2 code automatically generated by the XSLT transformation, can be syntactically checked by ACL2. As far as we do not pretend to translate the specification of the behavior of operators from Isabelle to the target languages (ACL2 in this case), we simply assign to the function a *nil* definition, ignoring its parameters. When the ACL2 obtained code be used as a guideline for a formalization, each function will have to be suitably implemented.

In the ACL2 code generated in this step, an example of which we will show later, definitions appear first, and then theorems are introduced. The order of theorems is obtained from the XLL file, and thus possible dependencies among them is preserved.

The ACL2 theorems obtained follow a full prefix notation, that is, for example $a + b$ is written in ACL2 as $+ \, a \, b$. In order to introduce new statements of properties in ACL2, the command *def-thm* has to be used. For example:

```
(defthm assoc-of-app
        (equal (app (app a b) c)
               (app a (app b c))))
```

There is one special case that has to be treated separately: the case where quantified variables appear in statements. In order to define a function whose body has an outermost quantifier, we have to make use of the command *defun-sk* (see [26] for a detailed explanation). For example:

```
(defun-sk exists-x-p0-and-q0 (y z)
  (exists x
          (and (p0 x y z)
               (q0 x y z))))
```

We also have to remark that, in Isabelle notation, quantifiers can bind various variables[7], for instance:

[7] Really, this is only syntactic sugar that the system offers to bind more than one variable with a single quantifier; internally, each existencial quantifier binds a single variable. Nevertheless, this syntactic artifact needs to be taken into account when we parse the Isabelle input.

```
∃ a b. a + b = 2
```

Nevertheless, when the Pre_OCL XML tree of this theorem is generated following steps 1, 2 and 3, the single quantifier is converted to two different ones, each of them binding a single variable (and since XLL is constructed from the Pre_OCL XML file, the same will occur in it). That is, the statement of the theorem that we already obtain in step 2 is:

```
∃ a. ∃ b. a + b = 2
```

Of course, this theorem can be proved equivalent to the original one. In ACL2, several quantifiers can appear in a single *defun-sk* (otherwise, we would obtain several nested *defun-sk*, which would make the code harder to read). The next example shows a universal quantifier in which two variables are bound simultaneously:

```
(defun-sk forall-x-y-p0-and-q0 (z)
  (forall (x y)
          (and (p0 x y z)
               (q0 x y z))))
```

Example on lists Applying the XSLT transformation over the XLL file, ACL2 code of our example on lists is obtained. The result is a single ACL2 file in which the theorems' statements selected in step 1 and the definitions of functions appear.

For instance, the Isabelle theorem *hd_Cons_tl*

```
lemma hd_Cons_tl [simp,no_atp]: "xs ≠ [] ==> hd xs # tl xs = xs"
by (induct xs) auto
```

has been translated to the following ACL2 statement:

```
(defthm hd_Cons_tl
       (implies (and (List2.listp xs)
                     (not (equal xs List2.list.Nil)))
                (equal (List2.list.Cons (List2.hd xs) (List2.tl xs)) xs)
```

The Isabelle notion of a variable being of a certain type is translated in ACL2 to an ad-hoc predicate resembling Isabelle types. For example, in the theorem presented above, the premise `List2.listp xs` has been introduced to assure that the variable xs is a list.

If the ACL2 file obtained is processed, it can be seen that it is syntactically correct (see Figure 12); the "similarity" between Isabelle and ACL2 statements can be also observed from the previous example, and also in the example on matrices that we will introduce later.

We have obtained a guideline (with the collection of functions involved in theorems, as well as the theorems' statements) to prove the statements of an Isabelle theory given as input in ACL2. Now definitions must be implemented in ACL2 to help the system to demonstrate the theorems.

```
*resultado_listas.lisp   *resultado_matrices.lisp
Compatible Mode
(defun List2.list.Cons (x1 x2 )
    (declare (ignore x1 x2 ))
    nil)

(defun List2.hd (x1 )
    (declare (ignore x1 ))
    nil)

(defun List2.tl (x1 )
    (declare (ignore x1 ))
    nil)

(defthm append_assoc
      (implies (and (listp xs )
                    (listp ys )
                    (listp zs )
                    )
          (equal (List2.append (List2.append xs ys ) zs ) (List2.append xs (List2.append ys zs ) ) )
))
```

```
*resultado_listas.lisp.a2s
Ready for command input  Compatible Mode
Form:  ( DEFTHM APPEND_ASSOC ...)
Rules: ((:FAKE-RUNE-FOR-TYPE-SET NIL)
        (:TYPE-PRESCRIPTION LIST2.APPEND))
Warnings:  Subsume and Non-rec
Time:  0.00 seconds (prove: 0.00, print: 0.00, proof tree: 0.00, other: 0.00)
 APPEND_ASSOC
ACL2 !>
```

Fig. 12. ACL2 processing the code

3.8 Step 7: From XLL to Ecore

This step is labeled in Figure 5 as step 7. Now we aim at generating a model in Ecore of an Isabelle theory. Ecore was conceived to create object oriented models; indeed, it is close to Java, and a subset of UML. One of the features of Java and UML that is not included in Ecore is the possibility to define static methods.

The translation of an XLL file to an Ecore model (an XMI file) consists of six simple consecutive XSLT transformations:

1. **Xll_to_ecore.xsl:** This template takes as input the XLL document obtained in step 4 and makes the main necessary syntax translations to transform the part about specification of datatypes of the XLL document to the corresponding Ecore model.
2. **Remove_dots.xsl:** Since operations appear with their long identifiers (for example "Theory_name.operation_name") in Isabelle XML files (and then, in the XLL document too), the dots and the theory names are removed in order to obtain exclusively the operations' identifiers.
3. **Translate_product_type.xsl:** We have to translate the data types of the part about specification in the XLL document (in our case of study, the datatypes presented in the XLL document are Isabelle datatypes) to Ecore proper types. A special case is the product type which comes from the Isabelle product type $(a \times b)$ and it is a parametrized class in XLL. This XLL class is transformed in an Ecore parameterized class $Pair < a, b >$ using this

template. We have considered this case separately because is a type which is parameterized by two different parameters.

4. **Translate_types.xsl:** Using this transformation we process the remaining data types in the XLL file, mapping them to classes that we have created *ad hoc* in Ecore, or to Ecore data types. For instance, the Isabelle type "bool" will be converted to EBoolean in Ecore (internally, we have to specify its long identifier, which is *"ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EBoolean"*). The Isabelle type matrix is mapped to the class *matrix* that we have specifically created (and we have to refer to it as *"#//matrix"*).
5. **Plus_and_Times.xsl:** Ecore, as Java does, uses infix syntax for some special operations, as, for instance, arithmetic operators "+" and "∗". If we want to make use of them, we need to process the XLL input (now in prefix notation), detect these special operators and translate them to OCL predefined operations. That process is made using this XSLT transformation.
6. **Remote_apos.xsl:** The apostrophe (') is an invalid character in OCL. For that reason, we must avoid it in our model (in names and methods). Using this XSLT, we transform all occurrences of ' to _bis (for example, an operation named append' will be renamed to append_bis).

In order to avoid to make these consecutive transformations one by one, we use an XProc template which applies then incrementally. The code of this template is presented in Appendix 6.3.

Example on lists Following the example on lists, we start from the XLL file obtained in step 4. Now we only have to process the Xproc template to obtain as a result a "List.Ecore" file. This Ecore model produces the diagram shown in Figure 13.

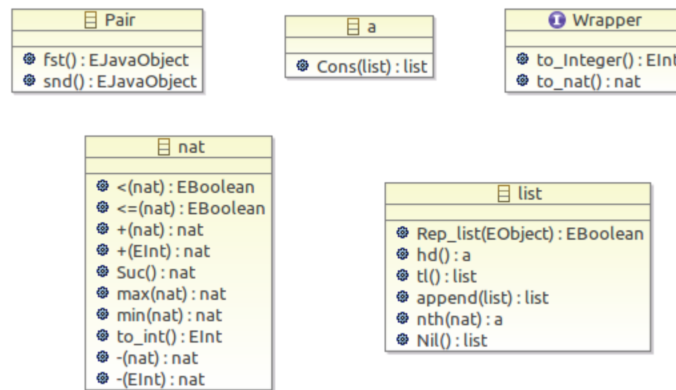


Fig. 13. Ecore Diagram obtained from List Theory using XSLT transformations

As it can be observed, five different classes have been created:

1. **Pair:** This class models the concept of parameterized "Tuple" and its basic operations. It will be crucial in our example on matrices, because a pair of natural numbers will be used to represent the position of each element of a matrix in Isabelle.
2. **Nat:** Ecore has the integers ("int") as one of its basic types, but the natural numbers ("nat") are not so. Therefore, this class is created to model the natural numbers and its operations (we introduce standard methods like $+, -, <, \leq,\ \max \ldots$).
3. **Wrapper:** This interface is necessary to assign to each positive number its correct type in each case. OCL interprets any number greater than zero as an integer (type "Integer"). Nevertheless, sometimes positive numbers are originated from elements of type "nat" in Isabelle, and therefore must be typed in OCL as naturals (variables which belong to the class "Nat" introduced previously). This interface allows the type conversions between "Int" and "Nat".
4. **List:** This is the most important class in this example. As we start from a *typedef 'a list*, this class is parameterized by a type parameter "*a*". That is, we obtain an Ecore $List < a >$ class (the class a, since it includes some particular operations, will be also generated to an Ecore class; we describe it in the next paragraph). The apostrophe of the type identifier *'a* has been automatically removed since it is an invalid character in OCL. We can see that almost every function selected in step 1 belongs to this class, now in the form of a class method.
5. **a:** This class has been automatically generated to model the Isabelle free type *'a*. This class has been also used to introduce some other additional methods that could have well also been placed in some other different classes; for instance, the function *Cons(list):list* has been added in class "a", even if it would also fit (even more properly) in class "List". The reason for taking this design decision is to get a more generic processing of operations. In Isabelle, the function *Cons(list):list* has the following type (pay special attention to the type of its first parameter):

```
"Cons" :: "'a ⇒ 'a list ⇒ 'a list"
```

As long as we are moving the Isabelle definitions to an object oriented setting, we must choose in which class we have to introduce each Isabelle function. As we have already explained, our universal decision is that every method will belong to the Ecore class that models the datatype of its first parameter. In the case of function *Cons(list):list*, its first parameter has type *'a*, and thus a method named "Const" has been generated in the generated Ecore class "a" automatically. We could have also processed some Isabelle functions in a differentiated way to introduce them in the Ecore classes of their second or third parameters, or even leave it to the user's choice, but this would complicate the processing steps and the technology without improving the final

product (the user only needs to know where the method is, only conceptually its placement in one class or another may be of some relevance).

3.9 Step 8: From XLL to OCL

This step is labeled as step 8 in Figure 5. Actually, both steps 7 and 8 could be thought of together; one of them (step 7) generates the Ecore model, starting from data types declarations and operations in the XLL file, the other one gives place to the associated restrictions (in OCL). We just split them into two separate steps in order to obtain XSLT transformations of a smaller complexity (actually, step 9 will be in charge of joining the resulting XML files of steps 7 and 8 into a single XMl file, by almost simply appending the original files).

Once we have the Ecore model with the classes and their methods, we are to add the theorems' statements as OCL restrictions. We make use of five XSLT transformations:

1. **Remove_apos.xsl:** As we already did in the previous step, we have to remove the apostrophe character from the theorems, replacing it by _bis.
2. **Remove_dots_from_names.xsl:** Operations in the Pre_OCL XML file are written with long identifiers (including their file name). Using this XLST template we obtain the short identifiers (removing file names).
3. **Remove_dots_from_types.xsl:** We have to make a similar process than in the previous template but with type names.
4. **Translate_operations.xsl:** Using this template we map the operations of Isabelle that can be mapped to already existing operations in OCL. For instance, the Isabelle constant "HOL.True" is mapped to the OCL constant "true", ' 'HOL.conj" to "and"... . In general, we have tried to use the Ecore and OCL already existing operations as far as possible, adapting us to the tool constructs, instead of developing an "ad-hoc" translation (for instance, this ad-hoc translation could be implemented creating an Ecore class named "HOL" with methods "and", "True", "implies"...). Nevertheless, as we have said before, in the XLL document (but not in the Ecore generated model) there exists a class named "Program Logic" which contains the main logic operators (in order to make possible the checking of the coherence between theorems and methods' names in XLL).
5. **Convert_to_OCL_object_oriented_version.xsl:** Applying this XSLT template we translate the theorems' statements which are in tree structure in the XLL file to strings representing OCL restrictions. For this, we have to take into account that in the source XML file, theorems appear in prefix Isabelle notation and we have to translate them to object oriented notation (usual operations have to be mapped to classes plus methods, there could appear an active object named "self" in certain restrictions...). Some details are also considered:
   - Not every operation in Ecore (neither in Java) is introduced in the form of a method; operations over basic types are sometimes introduced in an infix notation, and without any associated class: $+, -, *, <$, $\wedge$, $\vee$, *implies* (implication is denoted as "A *implies* B" instead of "$A.implies(B)$").

– The minus operation $-$ has at least two different meanings: the binary subtraction $(a - b)$ or the unary change of sign $(-a)$.
– In order to avoid the definition of sets, we translate the Isabelle interval set "$i \in \{k.. < n\}$" to a pair of predicates "$(k <= i)$ *and* $(i < n)$".
– We have introduced a class named Nat which models the natural numbers and some additional operations. A problem appears here: Isabelle allows overloading of constants, and thus the number 0 (and any other positive integer) belongs as much to naturals as to the integers. By default, when OCL processes the constant 0, its inferred type is "int", instead of "nat" and this may cause type inference problems in the OCL automatically generated expressions. Therefore, when we find in XLL a constant 0 with assigned type "nat", we invoke the class Wrapper adding at the beginning of the theorem:
```
Wrapper.allInstances->forAll(zero|zero.to_integer()=0 implies...)
```
This XSLT transformation looks for the occurrences of the natural number 0 (taking are of the type it appears with in the context) and translates them to "zero.to_nat()" in the corresponding OCL restrictions (if we don't make this conversion, by default, OCL considers 0 as a "int"), being "zero" an instance of the class Wrapper and such that *"zero.to_integer()=0"*[8]. For example, the Isabelle statement in the original input file "$(0 :: nat) + 0 = 0$" would be translated to the OCL restriction:
```
Wrapper.allInstances->forAll(zero|zero.to_integer()=0
implies zero.to_nat() + zero.to_nat() = zero.to_nat())
```

As in previous steps, we make use of an XProc template to apply the enumerated XLST transformations incrementally. This XProc template is presented in Appendix 6.4. The output is an XML file containing the OCL restrictions generated from the statements in the XLL file, which will be now finally merged with the Ecore model generated in step 7.

Example on lists In our example on lists, we generate the OCL restrictions of the three theorems *append_assoc, append_is_Nil_conv* and *hd_Cons_tl* (these theorems were presented in Isabelle standard notation in Section 3.2). For converting them to OCL restrictions, we apply the XProc template to the file XLL obtained after step 4. The resulting OCL is shown below:

```
<eAnnotations source="http://www.xocl.org/NAMED_OCL">
  <details key="append_assoc" value="list.allInstances()->
  forAll (xs|list.allInstances()->forAll (ys|list.allInstances()->
  forAll (zs|(xs.append(ys).append(zs) = xs.append(ys.append(zs))))))"/>

  <details key="append_is_Nil_conv" value="list.allInstances()->
  forAll (lista |list.allInstances()->forAll (xs|list.allInstances()->
  forAll (ys|((xs.append(ys) = lista.Nil()) = ((xs = lista.Nil())
```

[8] In fact, we have to write the implication ("zero.to_integer() = 0" implies ...) because *zero* is simply a name that we have assigned to the variable of type Wrapper.

```
  and (ys = lista.Nil())))))"/>

  <details key="hd_Cons_tl" value="list.allInstances()->forAll (lista
  |list.allInstances()->forAll (xs|(not((xs = lista.Nil())) implies
  (xs.hd().Cons(xs.tl()) = xs))))"/>
</eAnnotations>
```

Note that each tag *details* corresponds to an OCL restriction. The theorem's identifier is stored in the attribute *key*, and the OCL restriction in the attribute *value*.

3.10 Step 9: Joining Ecore and OCL

Finally, we bring together the OCL restrictions (obtained after step 8) and the Ecore model (obtained after step 7) using another XSLT transformation. Its template works as follows:

1. The OCL XML file (obtained in step 8) must be stored in a file "OCL.xml" which has to be placed in the same path as the XSLT template.
2. The XSLT template is first applied to the Ecore model file generated in step 7.
3. Internally, the transformation copies the whole Ecore model to an XML file, looks for the OCL file in the same folder and copies it in the proper location; the transformation generates a new XML file from the Ecore model file where the restrictions in the OCL xml file are properly inserted, giving rise to an Ecore model that has been enriched with OCL restrictions.

It can be noted that the template is applied to the Ecore model file, but the content of a second file (the file that contains the OCL restrictions) is also incorporated. This can be done by using the XSLT function *document()*, which needs to know the complete path of the second file as input. This was our motivation to fix the name of the OCL restrictions file as "OCL.xml" and place it in the same path as the template.

The OCL restrictions generated are syntactically correct with respect to the Ecore model.

Example on lists In our example on lists, once we have applied the XSLT transformation presented in this step we obtain an Ecore model together with a set of OCL restrictions generated from the original Isabelle theory like:

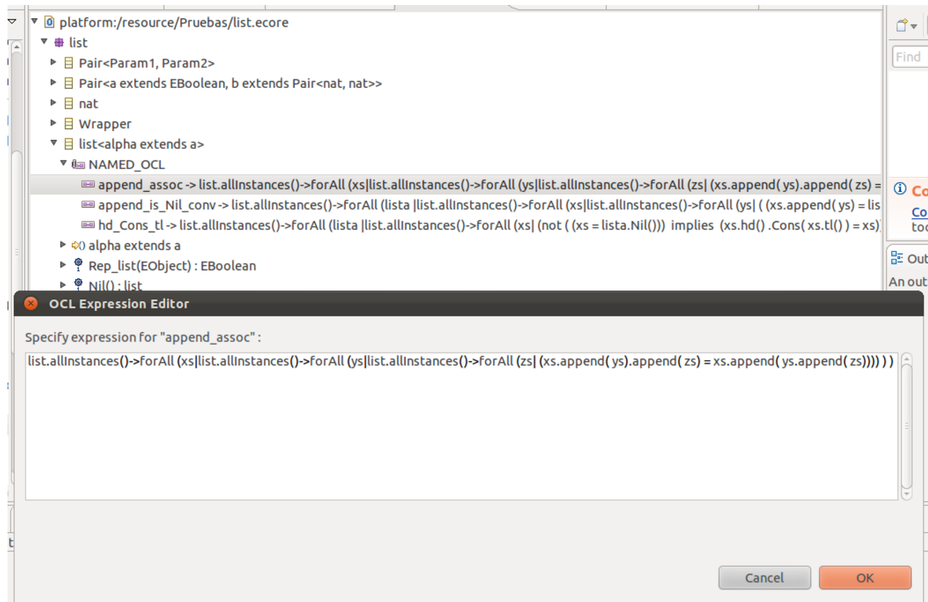


Fig. 14. Ecore model with OCL restrictions.

4 An example on matrices: diagonalizing an integer matrix

As we said in the introduction of this paper, the previous technology was originally designed to port an Isabelle/HOL development about matrices to ACL2 and Ecore + OCL, even if we have shown by means of the previous example that the tool is generic enough to be applied to another Isabelle developments with minor modifications. The development on matrices that we are using as input for our technology is available from the following link [22]. In the website can be also found the XLST transformations and Java programs that we have used in each step (as well as the intermediary XML and text files obtained after each step).

The fundamental theorem proved in our theory about matrices, claims that every integer matrix can be diagonalized using elementary transformations (row and column operations); the theorem additionally proves that there exist two invertible matrices P and Q such that the original matrix A and the diagonalized matrix B satisfy $B = PAQ$. This result is important to achieve the Smith normal form of a matrix; most algorithms presented in the literature to obtain the Smith normal form consists of two differentiated parts (for example [19,20]); in the first part, the matrix is diagonalized exclusively applying elementary transformations (*i.e.*, they are equivalent to the algorithm that we present here).

This Isabelle theory is lengthy (ca. 6000 lines, 30 definitions and 220 lemmas). It relies on a previous development of matrices in the Isabelle library (which

was successfully applied to prove a relevant part of the Kepler conjeture [28]), but we had to introduce ourselves the notion of elementary operations and the formalization of the algorithm, as presented in [27]. In the following sections, we illustrate the stepwise transformations applied to achieve the ACL2 and Ecore + OCL codes.

4.1 Step 1

Firstly, a selection of the important elements of the input theory (definitions, functions, theorems…) is performed in a text file (as presented in 3.2).

The text file for this theory is also presented in [22]; it contains:

- 2 typedef
- 2 partial functions
- 1 fun
- 29 definitions
- 119 theorems

Once the file is processed with the Java program (presented in Section 3.2), two folders *Class* and *Theorems* are obtained with the information presented in an XML tree structure following the XSD Schema presented in appendix 6.1. Concretely, there are 35 XML files in the folder Class (2 created from the typedef, 2 partial functions, 1 fun, 29 definitions, and the index_class.xml file) and 120 XML files in the folder Theorems (119 theorems and the index_theorems.txt).

4.2 Step 2

From the XML files obtained after step 1, the translation to a prefix notation in a text file named *Diagonal_form.txt* is carried out. The procedure has been presented in Section 3.3. It is performed by means of a Java program which transforms the Isabelle XML files obtained in the step 1 to a text file where the theorems' statements are written in a prefix notation (we named this format Pre_OCL text file). As we have said before, this transformation will be an intermediary step to achieve an XML tree where the Isabelle statements are stored in a prefix notation and de Bruijn indexes are avoided.

The result of this step is a rather large file (about 600 lines) due to the numerous theorems that have been processed. This file can be found in [22].

4.3 Step 3

This step performs the conversion of the text file obtained after the previous step to a XML file which follows the simple XSD Schema presented in Appendix 6.2. In our case, the output is a huge file of 11775 lines, in which the 119 theorems selected in step 1 appear, structured in XML trees compliant to the Pre_OCL XML xsd; this intermediary dialect is intended to simplify the later translations to ACL2 and OCL; it corresponds almost literally to the XLL part of our theory which includes information about the theorem's statements.

4.4 Step 4

In this step, the keystone of our development, the XLL document, is generated. For that, the XLST transformations presented in Section 3.5 are applied to the file named index_class.xml, which is placed in the folder Class created in step 1. As we explained before, the 34 additional files situated in folder Class will be appended in a single file, and from them the information to construct the XLL part corresponding to specification of datatypes (and their operations). This part about the specification of data types is then joined to the Pre_OCL XML file obtained after step 3, giving rise to the XLL document.

This XML file follows the XLL schema presented in Appendix 6.7; in this example, it is file which takes up *ca.* 12200 code lines.

It is worth noting that this schema brings together the information about data types and their operations, as well as the lemmas in a prefix notation and with suitable binders, in a form that can be easily mapped to ACL2 and OCL restrictions (and, in general, to first-order logic settings).

4.5 Step 5

We can translate the statements presented in the XLL document created in step 4 to Isabelle following the XSLT transformation presented in Section 3.6. The result is an Isabelle theory (named Certified Theory), which guarantees the correctness of the statements as introduced in the XLL document.

Applying this XSLT transformation to our XLL file about matrices, the XLL document obtained in step 1 and the original Isabelle theory are coupled (the statement of each input theorem is brought together with the statement generated after steps 1 to 4). Then, each copule of statements are proved equivalent by Isabelle. The Isabelle "Certified Theory" file obtained in this example takes up 2350 lines (see Figure 15):

4.6 Step 6

Finally, starting from the XLL file about matrices obtained after step 4, the ACL2 code can be generated. In this example, the result is a file that takes up to 1400 lines.

As we already noticed in Section 3.7, Isabelle is a typed language, whereas ACL2 is untyped. In order to fill this gap between both logical settings, we add for each ACL2 theorem suitable premises about each variable: the Isabelle notion of variables being of a certain type is translated in ACL2 to "ad-hoc" predicates resembling the Isabelle types.

These predicates are standard practice in ACL2, and are called *recognizers*. There already exist some recognizers in the ACL2 library, for instance there is a predicate *natp* which allows to know if a variable is a natural number (see [21] for more information). In our case study on matrices, a predicate *matrixp* is automatically generated in order to check if a given variable is an integer matrix (the definition of integer matrices in ACL2 must be incorporated by the user,

Fig. 15. Certified theory on matrices processed.

following our original idea of letting the representation of data types to the users choice in each target system, as well as a sound definition for the following predicate):

```
(defun matrix_integerp (x) (declare (ignore x)) nil)
```

For example, in the following Isabelle theorem, variables A, n and m have types “matrix”, “nat” and “nat” respectively.

```
lemma interchange_columns_matrix_id:
shows "interchange_columns_matrix (interchange_columns_matrix A n
       m) n m = A"
```

The following ACL2 code is obtained after applying to the previous statement the steps 1 to 6:

```
(defthm interchange_columns_matrix_id
      (implies (and (matrix_integerp A) (natp n) (natp m))
      (equal (interchange_columns_matrix (interchange_columns_matrix
      A n m) n m) A)))
```

As it can be seen, an additional implication is added to the original lemma in order to check (using the functions *matrix_integerp* and *natp*) that variables A, n and m must have the appropriate representation.

In Section 3.7 we already presented the special function *defun-sk*, which is used to define a function whose body has an outermost quantifier. Nevertheless,

it could arise the case of a theorem in which quantifiers appear but not in the outermost position. For example, in the following Isabelle theorem of our theory about matrices:

```
lemma interchange_rows_nrows:
assumes n:"n<nrows A"
and m:"m<nrows A"
and i: "\<exists>i. Rep_matrix A (min n m) i \<noteq> 0"
shows "nrows (interchange_rows_matrix (A::int matrix) n m) = nrows A"
```

The existencial quantifier appears in one of the premises. In this case, the theorem is automatically converted to both, a *defun-sk* which represents the premise, and also a *defthm*.

```
(defun-sk exists_interchange_rows_nrows (A n m)
   (exists (i)
   (not (equal (Matrix.Rep_matrix A (Orderings.ord_class.min n m) i) 0))))

(defthm interchange_rows_nrows
    (implies (and (natp n)
                  (matrix_integerp A)
                  (natp m)
                  (< n (Matrix.nrows A))
                  (< m (Matrix.nrows A))
                  (exists_interchange_rows_nrows A n m))
              (equal (Matrix.nrows
                          (Diagonal_form.interchange_rows_matrix A n m))
                     (Matrix.nrows A))))
```

If the previous ACL2 code is evaluated, it can be observed that it is syntactically correct (see Figure 16). Now, we have achieved a guideline in ACL2 to formalize the same theory than we formalized in Isabelle. In this example, the result is an ACL2 file which takes up 1400 code lines, contains 38 definitions *defun*, 19 *defun-sk* and 119 theorems *defthm*.

4.7 Step 7

Now we use the tool to generate a model in Ecore. For that, the XLST transformations presented in Section 3.8 are applied to the XLL document generated in step 4. The result is an Ecore model which has to be imported in Eclipse, and then a class diagram with the information that the model contains is shown (we present it in Figure 17).

In our example, most of the methods obtained from definitions, functions and partial functions belong to a class named Matrix. We must note that the parameterized class Pair is very relevant to our example, since we use it to represent the coordinates or positions of matrices elements (a pair of “nats”); there also exists a function named *minNonzero_nc* which returns this data type (concretely, *Pair<Boolean, Pair<Nat,Nat>>*).

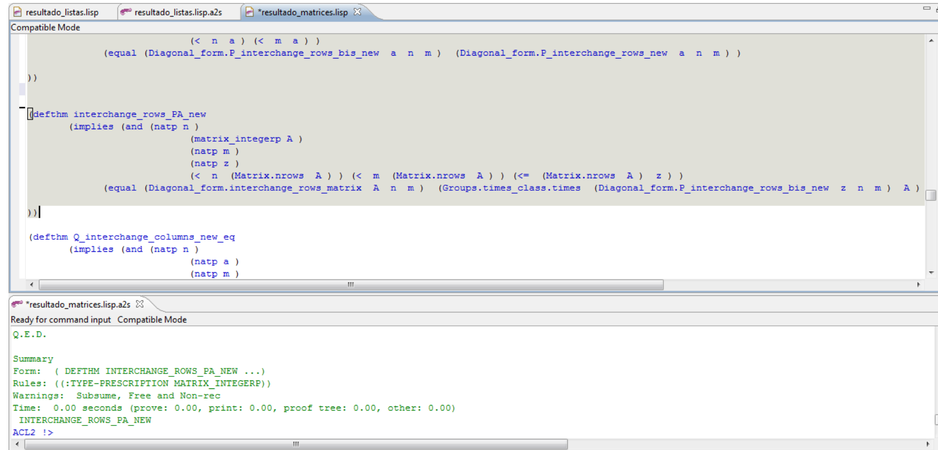


Fig. 16. ACL2 processing the code

4.8 Step 8

Now the theorems' statements are converted to OCL restrictions. By following the transformations explained in Section 3.9, a file named "OCL.xml" is obtained in which appear the 119 theorems written in OCL, in a single XML file.

For instance, the theorem of the input Isabelle theory about matrices which claims that:

```
lemma Diagonalize_theorem:
shows "∃P Q B. is_invertible P ∧ is_invertible Q ∧ B = P*A*Q
  ∧ is_square P (nrows (A::int matrix)) ∧ is_square Q (ncols A)
  ∧ Diagonalize_p B (max (nrows A) (ncols A))"
```

It has been translated to the following OCL restriction, over the model obtained in step 7:

```
matrix.allInstances()->forAll(A|matrix.allInstances()->exists(P|
matrix.allInstances()->exists(Q|matrix.allInstances()->exists(B|
 (P.is_invertible() and (Q.is_invertible() and ((B=((P*A)*Q))
  and (P.is_square(A.nrows()) and (Q.is_square(A.ncols())
  and B.Diagonalize_p(A.nrows().max(A.ncols()))))))))))
```

And this restriction is stored in an xml file as follows:

```
<details key="Diagonalize_theorem" value="matrix.allInstances()->
   forAll(A|matrix.allInstances()->exists(P|matrix.allInstances()->
   exists(Q|matrix.allInstances()->exists(B|(P.is_invertible()
      and (Q.is_invertible() and ((B=((P*A)*Q))
      and (P.is_square(A.nrows()) and (Q.is_square(A.ncols())
      and B.Diagonalize_p(A.nrows().max(A.ncols()))))))))))"/>
```

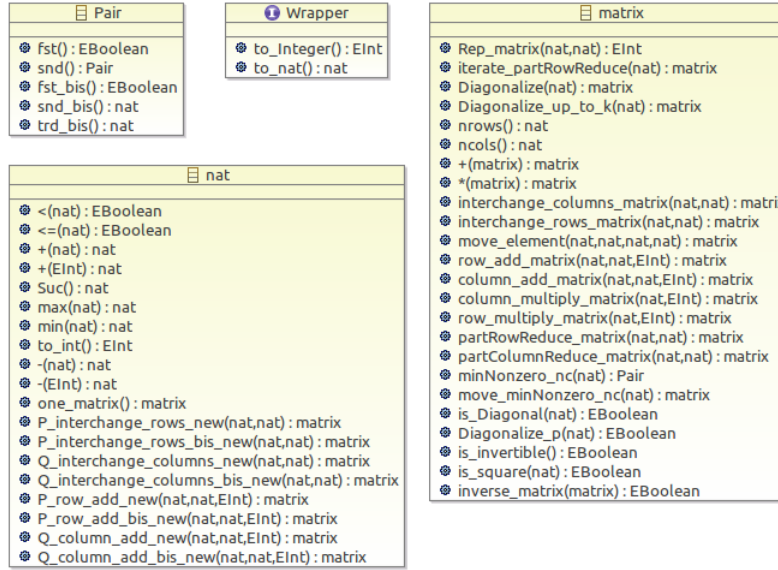


Fig. 17. Our Isabelle theory about matrices modeled in Ecore

4.9 Step 9

Finally, the OCL restrictions obtained in step 8 (the file "OCL.xml") are joined with to the Ecore model obtained in step 7.

Making use of the XLST transformation available in our website, the final file with the theorems included as OCL annotations in the Ecore model is obtained (see Figure 18). The result is a lengthy XML file, in which the OCL can be verified (thanks to appropriate tools and plugins of Eclipse). The generated OCL is syntactically valid with respect to the Ecore model obtained in step 7.

5 Related work and further work

5.1 Related work

As we have already pointed out in the introduction, different attempts of transferring formalization efforts from one theorem proving assistant to some others have been accomplished. We do not pretend to carry out an exhaustive classification, but at least two different approaches can be noted. First, translations which make use of some (ad-hoc or widely used) meta-language; the technology that we have presented could be included in this category. Second, direct translations (usually as embedding of one language into another). The technology presented in this work is closer to the first family; we have been using generic languages (in the form of XML *schemas*) as far as it has been possible; moreover, the same XML schema has been used to produce an Isabelle output, the Ecore + OCL

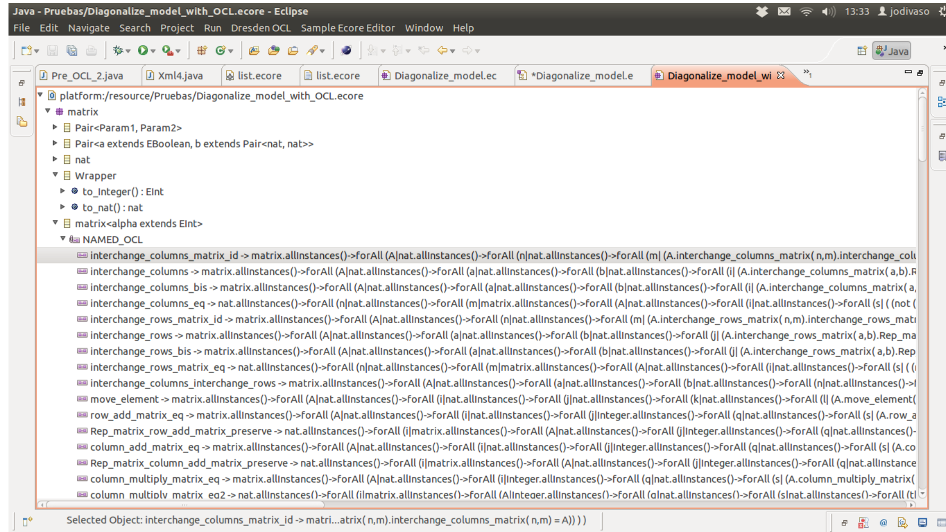


Fig. 18. The Isabelle theory about matrices modeled in Ecore with OCL restrictions

information and the ACL2 statements; this "meta-language" could be indeed origin for further translations to other theorem provers, such as for instance Coq, or first-order systems.

One of the most popular tools in the first approach presented is the TPTP [29] (thousands of problems for theorem provers) project. This set of test problems is thought to challenge and prospect the capacities of automated theorem provers (ATPs); as a by-product of the library of problems, a set of languages (untyped clause normal form, or CNF, first-order form, or FOF,...) has been proposed, which are now used by several reasoning tools. Nevertheless, the provers and the languages are mainly focused on untyped first-order logics, which makes them unsuitable for our case of study. There is a facility which communicates Isabelle/HOL with automated theorem provers (ATPs) by first translating the HOL problems to first-order logic ones and then transferring the problems to specialized ATPs [17,18]. The approach has been widely acclaimed, specially by the users; a sensible idea would be to communicate ACL and Isabelle through TPTP formalisms, being also possible to take advantage of the other tools available in this infrastructure, but we would have also lost part of the generality reached through the XML languages presented, which have been proved useful to reach also the Ecore + OCL (and eventually more theorem proving languages, and not only the ones based on first-order logics, with a feasible effort).

Some families of theorem provers share also logical foundations. For instance, HOL4, HOL-Light, ProofPower/HOL and Isabelle/HOL share a similar logic (higher-order logic), are mainly interactive (even if nowadays all of them have powerful automation tools) and follow the LCF approach of defining theorems as a datatype, with inference rules operating on this data type to reach new

theorems (thus, theorems will be valid as far as the inference rules are correctly implemented). The OpenTheory [30] project aims at the definition of a common language (or format) to describe statements and proofs, in a way that enables back and forth communication among each one of the HOL theorem provers and the OpenTheory format. Up to date, HOL-Light and HOL4 can both read and generate proofs to the OpenTheory format, whereas Isabelle can only read proofs in the format. It is worth remarking that our approach tries to respect the conventionalism and particular representation of each particular system (ranging from object oriented theory, UML + OCL, to typed and untyped logics, as the ones in ACL2 and HOL), and that we do not pretend to translate proofs among systems. Thus, the OpenTheory format does not seem a suitable tool for our development.

Apart from the use of meta-languages or formalisms to communicate different theorem provers, there have been also several ad-hoc tools to port developments (or the whole system) from one theorem prover to a different one. Traditionally, the idea has been always to embed a simpler logical framework into a more complex one (such as the different attempts of deep and shallow embedding HOL-Light into Coq), but some attempts to move from a more intricate formalism to a simpler one have also succeeded (as the ones usually applied to communicate theorem provers with higher-order logics to first-order logics, to increase the degree of automation). We do not aim at doing a survey here of the different attempts, but simply to situate our work in this setting. We have moved from a more expressive formalism, such as HOL, to less expressive ones (OCL and ACL2) with two different intentions; first, the Isabelle development should be useful to create a new ACL2 development in which automation would play a crucial role. The ALC2 development could also offer feedback for Isabelle in the form of new induction schemas. Second, the Ecore model generated from Isabelle will have also different utilities: first, as a communication tool; being UML and OCL two widely known languages in the field of formal methods, the possibility to create an Ecore model from an Isabelle development eases its dissemination through the community. Second, Ecore offers interesting tools for the automatic creation and testing of instances of a model, that could give valuable information on the way to define structures from Ecore tested models of information systems.

5.2 Further work

We have already commented on some possible future research lines that originate from this work. We sum up them here and present additional ideas.

First, the enhancement of our technology for covering some other theorem provers could be considered. Taking into account that the Pre OCL and XLL schemas that we have proposed as a source for the generation of Ecore, OCL restrictions and ACL2 statements are rather simple, containing exclusively data type definitions, operations and theorem statements (including quantifiers), information about variables, constants, operations and their arity, it seems feasible to translate these basic constructs to some other theorem provers, almost independently of the formalism in which they are based upon. A different matter

would be to include more expressive constructions from within the Isabelle environment, such as type classes, lambda expressions, or higher-order propositions, that would demand additional features on our Pre OCL and XLL schemas, and consequently on the languages targeted. These limitations on our technology have not proved relevant for our case studies over lists and matrices (which had a considerable number of results and definitions), and probably a wide range of results from the library could be covered with the actual technology; a deeper study of the real limitations that could emerge in the intermediary steps would be also meaningful.

Second, one of the goals of the work was to provide hints or guidelines based on a development carried out in Isabelle for a different theorem prover (ACL2, in our case), but without compromising the representation of data types and operations in this particular system. We have obtained a collection of ACL2 statements based on the Isabelle/HOL ones, and that are compliant with the ACL2 syntax (moreover, they were generated from an intermediary language that we proved that preserved the Isabelle statements), but have not completed the experiment of writing down the ACL2 proofs of the obtained statements. This development should be completed in ACL2, and then conclusions drawn on the utility of the Isabelle statements; one would expect the statements to be useful, since they usually express properties that are intermediary steps for the final result (in our case, the correctness of the diagonalization algorithm), and they will hold accordingly in any other system (for instance, ACL2). The question remains open if these properties are really useful in the search of an ACL2 proof.

Third, the possibilities of the Ecore environment should be explored. We have translated the Isabelle theory (data types, operations and properties or statements) into the UML and OCL languages; this framework has well-known capacities for the generation of models of specifications, and the automatic verification of the OCL restrictions. We trust the translation proposed from Isabelle to Ecore (it is almost literal, except for the particularities of each language), but the generation of models in UML could be a useful tool to provide ideas on the representation of Isabelle data types (if a UML generated model can not satisfy certain restrictions, the representation should be discarded), or also as an oracle for Isabelle statements; if a given OCL restriction is not satisfied, its originating Isabelle property would not hold (at least, under that UML representation).

References

1. J. Aransay A mechanized proof of the Basic Perturbation Lemma. *Journal of Automated Reasoning* **40** (4) (2008), 271–292.
2. A. Church, A formulation of the simple theory of types, *The Journal of Symbolic Logic* **5** (2) (1940), 56–68.
3. Formath Project: Formalisation of Mathematics. `http://wiki.portal.chalmers.se/cse/pmwiki.php/ForMath`.
4. G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch

and Simon Winwood seL4: Formal Verification of an Operating System Kernel. *Communications of the ACM* **53** (6) (2010), 107 – 115.
5. T. Nipkow, L. Paulson and M. Wenzel. Isabelle/HOL: A proof assistant for Higher-Order Logic. Springer, 2002.
6. D. Steinberg, F. Budinsky, M. Paternostro and Ed Merks. EMF: Eclipse Modeling Framework. Addison-Wesley, 2009.
7. L. Paulson, The foundation of a generic theorem prover, *Journal of Automated Reasoning* **5** (3) (1989), 363–397.
8. OCLinEcore Tutorial. `http://help.eclipse.org/indigo/index.jsp?topic=%2Forg.eclipse.ocl.doc%2Fhelp%2FTutorials.html`.
9. XSLT Tutorial. `http://www.w3schools.com/xsl/default.asp`.
10. David Flanagan. Java Examples in a Nutshell. O'Reilly, 2004.
11. XProc Tutorial. `http://www.xfront.com/xproc/`.
12. Expect Script for processing a theory in Isabelle console. `https://lists.cam.ac.uk/pipermail/cl-isabelle-users/2011-April/msg00022.html`.
13. Isabelle XSD Schema. `http://isabelle.in.tum.de/repos/isabelle/raw-file/035b2afbeb2e/src/Pure/Tools/isabelle.xsd`.
14. Isabelle Mailing List. Thread about free and quantified variables. `https://lists.cam.ac.uk/pipermail/cl-isabelle-users/2012-September/msg00096.html`.
15. Free variables and bound variables. `http://en.wikipedia.org/wiki/Free_variables_and_bound_variables`.
16. Lambda Calculus `http://en.wikipedia.org/wiki/Lambda_calculus#Free_and_bound_variables`.
17. J. Meng and L. C. Paulson, Translating higher-order clauses to first-order clauses. *Journal of Automated Reasoning* **40** (1) (2008), 35-60.
18. L. C. Paulson and K. W. Susanto, Source-level proof reconstruction for interactive theorem proving. *Theorem Proving in Higher Order Logics, volume 4732 of LNCS* 232-245, 2007.
19. Gordon H. Bradley. Algorithms for Hermite and Smith Normal Matrices and Linear Diophantine Equations. Mathematics of Computation, volume 25, number 116. 1971.
20. Henri Cohen. A Course in Computational Algebraic Number Theory. Springer, 1995.
21. `http://www.cs.utexas.edu/users/moore/acl2/v5-0/NATP.html`
22. Website with the XSLT transformations and Java programs of the development. `http://www.unirioja.es/cu/jodivaso/isabelle_ecore_acl2/`
23. Xproc implementations. `http://xproc.org/implementations/`
24. Object Constraint Language version 2.2 `http://www.omg.org/spec/OCL/2.2/PDF/`
25. De Bruijn Index. `http://en.wikipedia.org/wiki/De_Bruijn_index`
26. ACL2 Documentation: Defun-sk. `http://www.cs.utexas.edu/users/moore/acl2/v3-2/DEFUN-SK.html`
27. Formalization of the algorithm that diagonalizes a matrix using Isabelle/HOL. `http://www.unirioja.es/cu/jodivaso/diagonal/`
28. S. Obua and T. Nipkow. Flyspeck II: the basic linear programs. `http://www21.in.tum.de/~nipkow/pubs/flyspeckII.html`
29. G. Sutcliffe, The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, **43**(4): 337–362. 2009.
30. J. Hurd, The OpenTheory standard theory library. Third International Symposium on NASA Formal Methods (NFM 2011), *LNCS* **6617** 177–191. 2011.

31. M. Wenzel, Isabelle as Document-oriented Proof Assistant. Conference on Intelligent Computer Mathematics / Mathematical Knowledge Management (CICM/MKM 2011), *LNAI* **6824** 244–259. 2011.
32. G. L. Steele, Common Lisp the Language, 2nd edition, Digital Press, 1990.
33. M. Kaufmann, P. Manolios, J S. Moore, Computer-Aided Reasoning: An Approach, Kluwer, 2010.
34. `http://www.cs.utexas.edu/~moore/acl2/`
35. XML Path Language. W3C Recommendation. `http://www.w3.org/TR/xmlschema-1`

6 APPENDIX

6.1 Isabelle XSD Schema

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<xsd:element name="class">
  <xsd:complexType>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
  </xsd:complexType>
</xsd:element>

<xsd:element name="type">
  <xsd:complexType>
    <xsd:group ref="typeGroup"/>
  </xsd:complexType>
</xsd:element>

<xsd:element name="types">
  <xsd:complexType>
    <xsd:group ref="typeGroup" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:complexType>
</xsd:element>

<xsd:group name="typeGroup">
  <xsd:choice>
    <xsd:element name="TVar">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element ref="class" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:string" use="required"/>
        <xsd:attribute name="index" type="xsd:integer"/>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="TFree">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element ref="class" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:string" use="required"/>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="Type">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:group ref="typeGroup" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:string" use="required"/>
      </xsd:complexType>
    </xsd:element>
  </xsd:choice>
</xsd:group>

<xsd:element name="term">
  <xsd:complexType>
    <xsd:group ref="termGroup"/>
  </xsd:complexType>
</xsd:element>

<xsd:group name="termGroup">
  <xsd:choice>
    <xsd:element name="Var">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:group ref="typeGroup"/>
        </xsd:sequence>
```

```
      <xsd:attribute name="name" type="xsd:string" use="required"/>
      <xsd:attribute name="index" type="xsd:integer"/>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="Free">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:group ref="typeGroup"/>
      </xsd:sequence>
      <xsd:attribute name="name" type="xsd:string" use="required"/>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="Const">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:group ref="typeGroup"/>
      </xsd:sequence>
      <xsd:attribute name="name" type="xsd:string" use="required"/>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="Bound">
    <xsd:complexType>
      <xsd:attribute name="index" type="xsd:integer" use="required"/>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="App">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:group ref="termGroup"/>
        <xsd:group ref="termGroup"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="Abs">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:group ref="typeGroup"/>
        <xsd:group ref="termGroup"/>
      </xsd:sequence>
      <xsd:attribute name="vname" type="xsd:string" use="required"/>
    </xsd:complexType>
  </xsd:element>
 </xsd:choice>
</xsd:group>

<xsd:element name="proof">
  <xsd:complexType>
    <xsd:group ref="proofGroup"/>
  </xsd:complexType>
</xsd:element>

<xsd:group name="proofGroup">
  <xsd:choice>
    <xsd:element name="PThm">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:group ref="termGroup" minOccurs="0"/>
          <xsd:element ref="types" minOccurs="0"/>
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:string" use="required"/>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="PAxm">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:group ref="termGroup" minOccurs="0"/>
          <xsd:element ref="types" minOccurs="0"/>
        </xsd:sequence>
```

```
        <xsd:attribute name="name" type="xsd:string" use="required"/>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="Oracle">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:group ref="termGroup"/>
          <xsd:element ref="types" minOccurs="0"/>
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:string" use="required"/>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="PBound">
      <xsd:complexType>
        <xsd:attribute name="index" type="xsd:integer" use="required"/>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="Appt">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:group ref="proofGroup"/>
          <xsd:group ref="termGroup" minOccurs="0"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="AppP">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:group ref="proofGroup"/>
          <xsd:group ref="proofGroup"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="Abst">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:group ref="typeGroup" minOccurs="0"/>
          <xsd:group ref="proofGroup"/>
        </xsd:sequence>
        <xsd:attribute name="vname" type="xsd:string" use="required"/>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="AbsP">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:group ref="termGroup" minOccurs="0"/>
          <xsd:group ref="proofGroup"/>
        </xsd:sequence>
        <xsd:attribute name="vname" type="xsd:string" use="required"/>
      </xsd:complexType>
    </xsd:element>
  </xsd:choice>
</xsd:group>

</xsd:schema>
```

6.2 Pre_OCL XSD Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Theorems">
    <xs:complexType>
      <xs:sequence minOccurs="1" maxOccurs="unbounded">
        <xs:element name="Theorem">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="name" type="xs:string"/>
```

```
              <xs:choice minOccurs="1" maxOccurs="1">
                <xs:element ref="forall"/>
                <xs:element ref="exists"/>
                <xs:element ref="operation"/>
              </xs:choice>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
 <xs:attribute name="theory" type="xs:string" use="required"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="forall">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="param">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="name" type="xs:string"/>
              <xs:element name="type" type="xs:string"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="body">
          <xs:complexType>
            <xs:choice minOccurs="1" maxOccurs="1">
              <xs:element ref="forall"/>
              <xs:element ref="exists"/>
              <xs:element ref="operation"/>
            </xs:choice>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="exists">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="param">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="name" type="xs:string"/>
              <xs:element name="type" type="xs:string"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="body">
          <xs:complexType>
            <xs:choice>
              <xs:element ref="forall"/>
              <xs:element ref="exists"/>
              <xs:element ref="operation"/>
            </xs:choice>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="constant">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name"/>
        <xs:element name="type" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="operation">
    <xs:complexType>
```

```
      <xs:sequence>
        <xs:element name="name">
                </xs:element>
        <xs:choice minOccurs="1" maxOccurs="unbounded">
          <xs:element ref="constant"/>
          <xs:element ref="forall"/>
          <xs:element ref="exists"/>
          <xs:element ref="operation"/>
        </xs:choice>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

6.3 Xproc template to generate Ecore model

```
<?xml version="1.0"?>
<p:pipeline
    version="1.0"
    xmlns:p="http://www.w3.org/ns/xproc"
    xmlns:ex="http://example.com">

    <p:declare-step type="ex:xslt" name="xslt">
        <p:input port="source" sequence="true" primary="true"/>
        <p:input port="parameters" kind="parameter"/>
        <p:output port="result" primary="true"/>
        <p:option name="stylesheet" required="true"/>

        <p:load name="load-stylesheet">
            <p:with-option name="href" select="$stylesheet"/>
        </p:load>

        <p:xslt>
            <p:input port="stylesheet">
                <p:pipe port="result" step="load-stylesheet"/>
            </p:input>
            <p:input port="source">
                <p:pipe port="source" step="xslt"/>
            </p:input>
        </p:xslt>
    </p:declare-step>

    <ex:xslt stylesheet="xll_to_ecore.xsl"/>
    <ex:xslt stylesheet="remove_dots.xsl"/>
    <ex:xslt stylesheet="remove_dots.xsl"/>
    <ex:xslt stylesheet="translate_product_type.xsl"/>
    <ex:xslt stylesheet="translate_types.xsl"/>
    <ex:xslt stylesheet="Plus_and_Times.xsl"/>
    <ex:xslt stylesheet="remote_apos.xsl"/>
</p:pipeline>
```

6.4 Xproc template to generate OCL

```
<?xml version="1.0"?>
<p:pipeline
    version="1.0"
    xmlns:p="http://www.w3.org/ns/xproc"
    xmlns:ex="http://example.com">

    <p:declare-step type="ex:xslt" name="xslt">
        <p:input port="source" sequence="true" primary="true"/>
        <p:input port="parameters" kind="parameter"/>
        <p:output port="result" primary="true"/>
        <p:option name="stylesheet" required="true"/>
```

```
        <p:load name="load-stylesheet">
            <p:with-option name="href" select="$stylesheet"/>
        </p:load>

        <p:xslt>
            <p:input port="stylesheet">
                <p:pipe port="result" step="load-stylesheet"/>
            </p:input>
            <p:input port="source">
                <p:pipe port="source" step="xslt"/>
            </p:input>
        </p:xslt>
    </p:declare-step>

    <ex:xslt stylesheet="remove_apos.xsl"/>
    <ex:xslt stylesheet="remove_dots_from_names.xsl"/>
    <ex:xslt stylesheet="remove_dots_from_names.xsl"/>
    <ex:xslt stylesheet="remove_dots_from_types.xsl"/>
    <ex:xslt stylesheet="translate_operations.xsl"/>
    <ex:xslt stylesheet="Convert_to_OCL_object_oriented_version.xsl"/>
</p:pipeline>
```

6.5 Example of XML file generated from Isabelle in step 1

From theorem *hd_Cons_tl*:

```
lemma hd_Cons_tl [simp,no_atp]: "xs \<noteq> [] ==> hd xs # tl xs = xs"
by (induct xs) auto
```

We obtain the next XML file:

```
 <?xml version="1.0" encoding="UTF-8"?>
<term xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="isabelle.xsd">
  <App>
    <Const name="HOL.Trueprop">
      <Type name="fun">
        <Type name="HOL.bool"/>
        <Type name="prop"/>
      </Type>
    </Const>
    <App>
      <Const name="HOL.All">
        <Type name="fun">
          <Type name="fun">
            <Type name="List2.list">
              <TVar name="'a">
                <class name="HOL.type"/>
              </TVar>
            </Type>
            <Type name="HOL.bool"/>
          </Type>
          <Type name="HOL.bool"/>
        </Type>
      </Const>
      <Abs vname="xs">
        <Type name="List2.list">
          <TVar name="'a">
            <class name="HOL.type"/>
          </TVar>
        </Type>
        <App>
          <App>
            <Const name="HOL.implies">
              <Type name="fun">
                <Type name="HOL.bool"/>
                <Type name="fun">
```

```
        <Type name="HOL.bool"/>
        <Type name="HOL.bool"/>
      </Type>
    </Type>
  </Const>
  <App>
    <Const name="HOL.Not">
      <Type name="fun">
        <Type name="HOL.bool"/>
        <Type name="HOL.bool"/>
      </Type>
    </Const>
    <App>
      <App>
        <Const name="HOL.eq">
          <Type name="fun">
            <Type name="List2.list">
              <TVar name="’a">
                <class name="HOL.type"/>
              </TVar>
            </Type>
            <Type name="fun">
              <Type name="List2.list">
                <TVar name="’a">
                  <class name="HOL.type"/>
                </TVar>
              </Type>
              <Type name="HOL.bool"/>
            </Type>
          </Type>
        </Const>
        <Bound index="0"/>
      </App>
      <Const name="List2.list.Nil">
        <Type name="List2.list">
          <TVar name="’a">
            <class name="HOL.type"/>
          </TVar>
        </Type>
      </Const>
    </App>
  </App>
</App>
<App>
  <App>
    <Const name="HOL.eq">
      <Type name="fun">
        <Type name="List2.list">
          <TVar name="’a">
            <class name="HOL.type"/>
          </TVar>
        </Type>
        <Type name="fun">
          <Type name="List2.list">
            <TVar name="’a">
              <class name="HOL.type"/>
            </TVar>
          </Type>
          <Type name="HOL.bool"/>
        </Type>
      </Type>
    </Const>
    <App>
      <App>
        <Const name="List2.list.Cons">
          <Type name="fun">
            <TVar name="’a">
              <class name="HOL.type"/>
```

```
                      </TVar>
                      <Type name="fun">
                        <Type name="List2.list">
                          <TVar name="’a">
                            <class name="HOL.type"/>
                          </TVar>
                        </Type>
                        <Type name="List2.list">
                          <TVar name="’a">
                            <class name="HOL.type"/>
                          </TVar>
                        </Type>
                      </Type>
                    </Type>
                  </Const>
                  <App>
                    <Const name="List2.hd">
                      <Type name="fun">
                        <Type name="List2.list">
                          <TVar name="’a">
                            <class name="HOL.type"/>
                          </TVar>
                        </Type>
                        <TVar name="’a">
                          <class name="HOL.type"/>
                        </TVar>
                      </Type>
                    </Const>
                    <Bound index="0"/>
                  </App>
                </App>
                <App>
                  <Const name="List2.tl">
                    <Type name="fun">
                      <Type name="List2.list">
                        <TVar name="’a">
                          <class name="HOL.type"/>
                        </TVar>
                      </Type>
                      <Type name="List2.list">
                        <TVar name="’a">
                          <class name="HOL.type"/>
                        </TVar>
                      </Type>
                    </Type>
                  </Const>
                  <Bound index="0"/>
                </App>
              </App>
            </App>
            <Bound index="0"/>
          </App>
        </App>
      </Abs>
    </App>
  </App>
</term>
```


6.6 Example of PRE_OCL XML file of list generated in step 3


```
 <?xml version="1.0" encoding="UTF-8"?>
<Theorems xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="pre_OCL.xsd">
  <Theorem>
    <name>append_assoc</name>
    <forall>
```

```
    <param>
      <name>xs</name>
      <type>'a List2.list</type>
    </param>
    <body>
      <forall>
        <param>
          <name>ys</name>
          <type>'a List2.list</type>
        </param>
        <body>
          <forall>
            <param>
              <name>zs</name>
              <type>'a List2.list</type>
            </param>
            <body>
              <operation>
                <name>HOL.eq</name>
                <operation>
                  <name>List2.append</name>
                  <operation>
                    <name>List2.append</name>
                    <constant>
                      <name>xs</name>
                    </constant>
                    <constant>
                      <name>ys</name>
                    </constant>
                  </operation>
                  <constant>
                    <name>zs</name>
                  </constant>
                </operation>
                <operation>
                  <name>List2.append</name>
                  <constant>
                    <name>xs</name>
                  </constant>
                  <operation>
                    <name>List2.append</name>
                    <constant>
                      <name>ys</name>
                    </constant>
                    <constant>
                      <name>zs</name>
                    </constant>
                  </operation>
                </operation>
              </operation>
            </body>
          </forall>
        </body>
      </forall>
    </body>
  </forall>
</Theorem>
<Theorem>
  <name>append_is_Nil_conv</name>
  <forall>
    <param>
      <name>xs</name>
      <type>'a List2.list</type>
    </param>
    <body>
      <forall>
        <param>
          <name>ys</name>
```

```
          <type>'a List2.list</type>
        </param>
        <body>
          <operation>
            <name>HOL.eq</name>
            <operation>
              <name>HOL.eq</name>
              <operation>
                <name>List2.append</name>
                <constant>
                  <name>xs</name>
                </constant>
                <constant>
                  <name>ys</name>
                </constant>
              </operation>
              <constant>
                <name>List2.list.Nil</name>
              </constant>
            </operation>
            <operation>
              <name>HOL.conj</name>
              <operation>
                <name>HOL.eq</name>
                <constant>
                  <name>xs</name>
                </constant>
                <constant>
                  <name>List2.list.Nil</name>
                </constant>
              </operation>
              <operation>
                <name>HOL.eq</name>
                <constant>
                  <name>ys</name>
                </constant>
                <constant>
                  <name>List2.list.Nil</name>
                </constant>
              </operation>
            </operation>
          </operation>
        </body>
      </forall>
    </body>
  </forall>
</Theorem>
<Theorem>
  <name>hd_Cons_tl</name>
  <forall>
    <param>
      <name>xs</name>
      <type>'a List2.list</type>
    </param>
    <body>
      <operation>
        <name>HOL.implies</name>
        <operation>
          <name>HOL.Not</name>
          <operation>
            <name>HOL.eq</name>
            <constant>
              <name>xs</name>
            </constant>
            <constant>
              <name>List2.list.Nil</name>
            </constant>
          </operation>
```

```
          </operation>
          <operation>
            <name>HOL.eq</name>
            <operation>
              <name>List2.list.Cons</name>
              <operation>
                <name>List2.hd</name>
                <constant>
                  <name>xs</name>
                </constant>
              </operation>
              <operation>
                <name>List2.tl</name>
                <constant>
                  <name>xs</name>
                </constant>
              </operation>
            </operation>
            <constant>
              <name>xs</name>
            </constant>
          </operation>
        </operation>
      </body>
    </forall>
  </Theorem>
</Theorems>
```

6.7 XLL Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">

  <xs:element name="xll">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Theorems" minOccurs="1" maxOccurs="1"/>
        <xs:element ref="Datatypes" minOccurs="1" maxOccurs="1"/>
      </xs:sequence>
    </xs:complexType>
    <xs:key name="keyList_methods">
      <xs:selector xpath=".//method"/>
      <xs:field xpath="@name"/>
    </xs:key>
    <xs:keyref name="referList_methods" refer="keyList_methods">
      <xs:selector xpath=".//operation/name"/>
      <xs:field xpath="."/>
    </xs:keyref>
  </xs:element>
  <xs:element name="Theorems">
    <xs:complexType>
      <xs:sequence minOccurs="1" maxOccurs="unbounded">
        <xs:element name="Theorem">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="name" type="xs:string"/>
              <xs:choice minOccurs="1" maxOccurs="1">
                <xs:element ref="forall"/>
                <xs:element ref="exists"/>
                <xs:element ref="operation"/>
              </xs:choice>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="theory" type="xs:string" use="required"/>
```

```
    </xs:complexType>
  </xs:element>
  <xs:element name="forall">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="param">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="name" type="xs:string"/>
              <xs:element name="type" type="xs:string"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="body">
          <xs:complexType>
            <xs:choice minOccurs="1" maxOccurs="1">
              <xs:element ref="forall"/>
              <xs:element ref="exists"/>
              <xs:element ref="operation"/>
            </xs:choice>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="exists">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="param">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="name" type="xs:string"/>
              <xs:element name="type" type="xs:string"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="body">
          <xs:complexType>
            <xs:choice>
              <xs:element ref="forall"/>
              <xs:element ref="exists"/>
              <xs:element ref="operation"/>
            </xs:choice>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="constant">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name"/>
        <xs:element name="type" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="operation">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type="xs:string">
                </xs:element>
        <xs:choice minOccurs="1" maxOccurs="unbounded">
          <xs:element ref="constant"/>
          <xs:element ref="forall"/>
          <xs:element ref="exists"/>
          <xs:element ref="operation"/>
        </xs:choice>
      </xs:sequence>
```

```
    </xs:complexType>
  </xs:element>

  <xs:element name="Type">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Type" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="name" use="required"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="Input">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Type" minOccurs="1" maxOccurs="1"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="Parameter">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Type" minOccurs="0" maxOccurs="1"/>
      </xs:sequence>
      <xs:attribute name="name"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="Datatypes">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Class" minOccurs="1" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Class_Parameters" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element ref="Parameter" minOccurs="0" maxOccurs="unbounded"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
              <xs:element name="method" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element ref="Type" minOccurs="1" maxOccurs="1"/>
                    <!--The output (maybe parameterized)-->
                    <xs:element ref="Input" minOccurs="0" maxOccurs="unbounded"/>
                    <!--Input parameters-->
                  </xs:sequence>
                  <xs:attribute name="name" use="required" type="xs:string"/>
                  <xs:attribute name="static" type="xs:boolean" default="false"/>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute name="name" use="required" type="xs:string"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

6.8 Example on lists: XLL document generated in step 4

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<xll xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="xll.xsd">
  <Theorems xsi:noNamespaceSchemaLocation="pre_OCL.xsd" theory="List">
    <Theorem>
      <name>append_assoc</name>
      <forall>
        <param>
          <name>xs</name>
          <type>'a List2.list</type>
        </param>
        <body>
          <forall>
            <param>
              <name>ys</name>
              <type>'a List2.list</type>
            </param>
            <body>
              <forall>
                <param>
                  <name>zs</name>
                  <type>'a List2.list</type>
                </param>
                <body>
                  <operation>
                    <name>HOL.eq</name>
                    <operation>
                      <name>List2.append</name>
                      <operation>
                        <name>List2.append</name>
                        <constant>
                          <name>xs</name>
                        </constant>
                        <constant>
                          <name>ys</name>
                        </constant>
                      </operation>
                      <constant>
                        <name>zs</name>
                      </constant>
                    </operation>
                    <operation>
                      <name>List2.append</name>
                      <constant>
                        <name>xs</name>
                      </constant>
                      <operation>
                        <name>List2.append</name>
                        <constant>
                          <name>ys</name>
                        </constant>
                        <constant>
                          <name>zs</name>
                        </constant>
                      </operation>
                    </operation>
                  </operation>
                </body>
              </forall>
            </body>
          </forall>
        </body>
      </forall>
    </Theorem>
    <Theorem>
      <name>append_is_Nil_conv</name>
      <forall>
        <param>
          <name>xs</name>
          <type>'a List2.list</type>
```

```
    </param>
    <body>
      <forall>
        <param>
          <name>ys</name>
          <type>'a List2.list</type>
        </param>
        <body>
          <operation>
            <name>HOL.eq</name>
            <operation>
              <name>HOL.eq</name>
              <operation>
                <name>List2.append</name>
                <constant>
                  <name>xs</name>
                </constant>
                <constant>
                  <name>ys</name>
                </constant>
              </operation>
              <constant>
                <name>List2.list.Nil</name>
              </constant>
            </operation>
            <operation>
              <name>HOL.conj</name>
              <operation>
                <name>HOL.eq</name>
                <constant>
                  <name>xs</name>
                </constant>
                <constant>
                  <name>List2.list.Nil</name>
                </constant>
              </operation>
              <operation>
                <name>HOL.eq</name>
                <constant>
                  <name>ys</name>
                </constant>
                <constant>
                  <name>List2.list.Nil</name>
                </constant>
              </operation>
            </operation>
          </operation>
        </body>
      </forall>
    </body>
  </forall>
</Theorem>
<Theorem>
  <name>hd_Cons_tl</name>
  <forall>
    <param>
      <name>xs</name>
      <type>'a List2.list</type>
    </param>
    <body>
      <operation>
        <name>HOL.implies</name>
        <operation>
          <name>HOL.Not</name>
          <operation>
            <name>HOL.eq</name>
            <constant>
              <name>xs</name>
```

```
              </constant>
              <constant>
                <name>List2.list.Nil</name>
              </constant>
            </operation>
          </operation>
          <operation>
            <name>HOL.eq</name>
            <operation>
              <name>List2.list.Cons</name>
              <operation>
                <name>List2.hd</name>
                <constant>
                  <name>xs</name>
                </constant>
              </operation>
              <operation>
                <name>List2.tl</name>
                <constant>
                  <name>xs</name>
                </constant>
              </operation>
            </operation>
            <constant>
              <name>xs</name>
            </constant>
          </operation>
        </operation>
      </body>
    </forall>
  </Theorem>
</Theorems>
<Datatypes>
  <Class name="a">
    <method name="List2.list.Cons">
      <Type name="List2.list"/>
      <Input>
        <Type name="List2.list"/>
      </Input>
    </method>
  </Class>
  <Class name="PL">
    <method static="true" name="HOL.implies">
      <Type name="HOL.bool"/>
      <Input>
        <Type name="HOL.bool"/>
      </Input>
      <Input>
        <Type name="HOL.bool"/>
      </Input>
    </method>
    <method static="true" name="HOL.conj">
      <Type name="HOL.bool"/>
      <Input>
        <Type name="HOL.bool"/>
      </Input>
      <Input>
        <Type name="HOL.bool"/>
      </Input>
    </method>
    <method static="true" name="HOL.disj">
      <Type name="HOL.bool"/>
      <Input>
        <Type name="HOL.bool"/>
      </Input>
      <Input>
        <Type name="HOL.bool"/>
      </Input>
```

```
  </method>
  <method static="true" name="HOL.eq">
    <Type name="HOL.bool"/>
    <Input>
      <Type name="HOL.bool"/>
    </Input>
    <Input>
      <Type name="HOL.bool"/>
    </Input>
  </method>
  <method static="true" name="HOL.Not">
    <Type name="HOL.bool"/>
    <Input>
      <Type name="HOL.bool"/>
    </Input>
  </method>
</Class>
<Class name="Arith">
  <method static="true" name="Orderings.ord_class.less_eq">
    <Type name="HOL.bool"/>
    <Input>
      <Type name="HOL.type"/>
    </Input>
    <Input>
      <Type name="HOL.type"/>
    </Input>
  </method>
  <method static="true" name="Groups.abs_class.abs">
    <Type name="HOL.type"/>
    <Input>
      <Type name="HOL.type"/>
    </Input>
  </method>
  <method static="true" name="Orderings.ord_class.less">
    <Type name="HOL.type"/>
    <Input>
      <Type name="HOL.bool"/>
    </Input>
    <Input>
      <Type name="HOL.type"/>
    </Input>
  </method>
  <method static="true" name="Orderings.ord_class.max">
    <Type name="HOL.type"/>
    <Input>
      <Type name="HOL.type"/>
    </Input>
    <Input>
      <Type name="HOL.type"/>
    </Input>
  </method>
  <method static="true" name="Orderings.ord_class.min">
    <Type name="HOL.type"/>
    <Input>
      <Type name="HOL.type"/>
    </Input>
    <Input>
      <Type name="HOL.type"/>
    </Input>
  </method>
  <method static="true" name="Divides.div_class.div">
    <Type name="HOL.type"/>
    <Input>
      <Type name="HOL.type"/>
    </Input>
    <Input>
      <Type name="HOL.type"/>
    </Input>
```

```
      </method>
      <method static="true" name="Set.member">
        <Type name="HOL.type"/>
        <Input>
          <Type name="HOL.bool"/>
        </Input>
      </method>
      <method static="true" name="SetInterval.ord_class.atLeastLessThan">
        <Type name="HOL.type"/>
        <Input>
          <Type name="HOL.bool"/>
        </Input>
      </method>
      <method static="true" name="Groups.uminus_class.uminus">
        <Type name="HOL.type"/>
        <Input>
          <Type name="HOL.type"/>
        </Input>
      </method>
      <method static="true" name="Nat.Suc">
        <Type name="Nat.nat"/>
        <Input>
          <Type name="Nat.nat"/>
        </Input>
      </method>
    </Class>
    <Class name="List2.list">
      <Class_Parameters>
        <Parameter name="alpha">
          <Type name="'a"/>
        </Parameter>
      </Class_Parameters>
      <method name="List2.Rep_list">
        <Type name="HOL.bool"/>
        <Input>
          <Type name="Datatype.node"/>
        </Input>
      </method>
      <method name="List2.hd">
        <Type name="'a"/>
      </method>
      <method name="List2.tl">
        <Type name="List2.list"/>
      </method>
      <method name="List2.append">
        <Type name="List2.list"/>
        <Input>
          <Type name="List2.list"/>
        </Input>
      </method>
      <method name="List2.nth">
        <Type name="'a"/>
        <Input>
          <Type name="Nat.nat"/>
        </Input>
      </method>
            <method name="List2.list.Nil">
        <Type name="List2.list"/>
      </method>
    </Class>
  </Datatypes>
</xll>
```


6.9 Text file generated in Step 2

```
append_assoc (HOL.All (\<lambda>xs::'a List2.list. (HOL.All (\<lambda>ys::'a List2.list.
(HOL.All (\<lambda>zs::'a List2.list. (HOL.eq (List2.append (List2.append xs ys) zs)
```

```
(List2.append xs (List2.append ys zs)))))))))

append_is_Nil_conv (HOL.All (\<lambda>xs::'a List2.list. (HOL.All (\<lambda>ys::'a List2.list.
(HOL.eq (HOL.eq (List2.append xs ys) List2.list.Nil) (HOL.conj (HOL.eq xs List2.list.Nil)
(HOL.eq ys List2.list.Nil)))))))

hd_Cons_tl (HOL.All (\<lambda>xs::'a List2.list. (HOL.implies (HOL.Not (HOL.eq xs List2.list.Nil))
(HOL.eq (List2.list.Cons (List2.hd xs) (List2.tl xs)) xs))))
```

6.10 Certified theory generated in Step 5

```
theory List2_certified
imports List2
begin

lemma append_assoc_prefix: shows "(HOL.All (\<lambda>xs::'a List2.list.
(HOL.All (\<lambda>ys::'a List2.list. (HOL.All (\<lambda>zs::'a List2.list.
(HOL.eq (List2.append (List2.append xs ys) zs) (List2.append xs (List2.append ys zs)))))))))"
using append_assoc by fast

lemma append_is_Nil_conv_prefix: shows "(HOL.All (\<lambda>xs::'a List2.list.
(HOL.All (\<lambda>ys::'a List2.list. (HOL.eq (HOL.eq (List2.append xs ys) List2.list.Nil)
(HOL.conj (HOL.eq xs List2.list.Nil) (HOL.eq ys List2.list.Nil)))))))"
using append_is_Nil_conv by fast

lemma hd_Cons_tl_prefix: shows "(HOL.All (\<lambda>xs::'a List2.list.
(HOL.implies (HOL.Not (HOL.eq xs List2.list.Nil))
(HOL.eq (List2.list.Cons (List2.hd xs) (List2.tl xs)) xs))))"
using hd_Cons_tl by fast

ML{*fun atomize_thm thm =  Thm.equal_elim (Object_Logic.atomize (cprop_of thm)) thm *}
ML{*Thm.eq_thm_prop (Object_Logic.rulify @{thm append_assoc_prefix}
                    ,Object_Logic.rulify @{thm append_assoc})*}
ML{*Thm.eq_thm_prop (@{thm append_assoc_prefix},atomize_thm
                    (forall_intr_vars @{thm append_assoc}))*}

ML{*Thm.eq_thm_prop (Object_Logic.rulify @{thm append_is_Nil_conv_prefix},
                     Object_Logic.rulify @{thm append_is_Nil_conv})*}
ML{*Thm.eq_thm_prop (@{thm append_is_Nil_conv_prefix},atomize_thm
                    (forall_intr_vars @{thm append_is_Nil_conv}))*}

ML{*Thm.eq_thm_prop (Object_Logic.rulify @{thm hd_Cons_tl_prefix},
                    Object_Logic.rulify @{thm hd_Cons_tl})*}
ML{*Thm.eq_thm_prop (@{thm hd_Cons_tl_prefix},atomize_thm
                    (forall_intr_vars @{thm hd_Cons_tl}))*}

end
```