Obtaining an ACL2 specification from an Isabelle/HOL theory⋆

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, Spain
jesus-maria.aransay@unirioja.es, jose.divasonm@unirioja.es,
jonathan.heras@unirioja.es, lalamban@unirioja.es, mvico@unirioja.es,
arubio@unirioja.es, julio.rubio@unirioja.es

Abstract. In this work, we present an interoperability framework that
enables the translation of specifications (signature of functions and
lemma statements) among different theorem provers. This translation
is based on a new intermediate XML language, called XLL, and is per-
formed almost automatically. As a case study, we focus on porting de-
velopments from Isabelle/HOL to ACL2. In particular, we study the
transformation to ACL2 of an Isabelle/HOL theory devoted to verify an
algorithm computing a diagonal form of an integer matrix. Moreover, we
provide a formal proof of a fragment of the obtained ACL2 specification
— this shows the suitability of our approach to reuse in ACL2 a proof
strategy imported from Isabelle/HOL.

1 Introduction

In the frame of the ForMath European project [1], several theorem provers are
used to verify mathematical algorithms, with an emphasis on Coq/SSReflect [11]
but also using intensively Isabelle/HOL [19] and ACL2 [16]. Due to this diversity
of tools, it was natural to investigate how different provers could collaborate, in
some manner, in the same formalisation effort.

Numerous contributions have been made along the years in the area of the-
orem proving interoperability. We give here just a few strokes of the brush, by
saying that translations among proof assistants can be of two kinds: *deep* and
shallow. In the former, deep translations, e.g. [6,12,15,18], the soundness of the
transformation is ensured, and thus, it is necessary to analyse semantics issues
(underlying logics, language expressiveness, and so on). In the latter, shallow
translations, e.g. [10,17,21], only the syntactical structure is translated from the
source formalism to the target one.

In this work, and starting from a complete formalisation in Isabelle/HOL, we
develop a set of tools that translates a *proof plan* to ACL2, looking for efficient

(but verified) executability. Since we do not have a *deep* Isabelle/HOL–ACL2 translator at our disposal, we try to materialise the previous observation by writing a *shallow* porting mechanism. Even if we do not aim at doing a survey of the state of the art in the field, it is worth noting that approaches abound in the literature. The Omega system [23] has been fruitfully used through the years to perform proof planning strategies. As far as we can determine, it is unable to integrate with the theorems provers we are interested in. A different tool is the Evidential tool bus [9]. Evidential's design principle is that of *semantic neutrality*; our work here is probably not different from it, but could be seen as an *ad-hoc* case study of what they call *translators* (in our case, from Isabelle/HOL to ACL2). Another meaningful possibility for our development would have been the use of the THF0 [4] language, to which several tools in TPTP are translated; indeed, a subset of Isabelle/HOL statements can be already translated to THF0, in order to enable the communication with external automated theorem provers. Applying a similar idea to ACL2 seems an interesting idea, but our interest in XML based tools, such as Ecore and OCL (see [3] for details), leads us to propose a different approach.

Our approach translates function signatures and statements, while proofs and function bodies are not ported. In principle, this weak process could be considered unsafe (this criticism could be also applied to any *shallow* strategy). Nevertheless, our key idea is based on the following argument: *the family of function signatures and statements in a formalisation encodes a* proof scheme *that can be reused in any other system.* Of course, some constraints must be added to render sensible this claim. For instance, the target framework must be expressive enough to receive the formulas from the source environment (at least, in the concrete problem to be ported). Additionally, such a reuse may be not optimal (otherwise, something as a *deep* translation would be accomplished), because both the data structures and the working style of each theorem prover can be very distant. In any case, at some convenient abstract level, the *sketch of the proof* can be translated, saving a significant amount of time. At the end of the process, when a complete proof is (re)built in the target system, the question about the soundness of the translation is no longer relevant.

The above proposal is instantiated in this paper in a particular case study, where we go from Isabelle/HOL to ACL2, transforming a complete proof in Isabelle/HOL (related to integer matrices manipulation) into an (incomplete) ACL2 specification. The essence of the proof is captured in this transformation, showing the adequacy of our contribution.

The organization of the paper is as follows. Our general framework to inter-operate is briefly described in Section 2. Section 3 is devoted to comment on the Isabelle/HOL theory developed and the translation process, while Section 4 deals with the completion of the ACL2 specification until a proof of a fragment of the theory is obtained. The paper ends with conclusions, further work and the bibliography. The paper is backed with a report [3] in which we have thoroughly described the architecture of the tool, two case studies, and the steps which can be applied to produce new translations.

2   A (minimal) framework to interoperate

The framework presented in [3] — from now on called *I2EA* (Isabelle/HOL to Ecore and ACL2) — allows the transformation of Isabelle/HOL specifications to both Ecore models [2] and ACL2 specifications; the role of Ecore is presented in [3]. In this paper, we only focus on proving the following concept: *the I2EA framework can be used to translate Isabelle/HOL specifications to ACL2, and to reuse a proof scheme in ACL2 imported from Isabelle/HOL*. To this aim, we just use the components of the I2EA framework shown in Figure 1. We describe the components of that diagram in the following subsections.
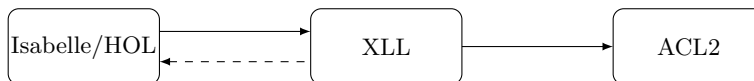

Fig. 1. (Reduced) Architecture of the I2EA framework.

2.1   Isabelle

Isabelle [19] is a generic interactive proof assistant, on top of which different logics can be implemented; the most explored of this variety of logics is higher-order logic (or *HOL*), and it is also the logic where a greater number of tools (code generation, automatic proof procedures) are available.

The HOL type system is rather simple; it is based on non-empty types, function types ($\Rightarrow$) and type constructors κ that can be applied to already existing types (*nat, bool*) or type variables (α, β). Types can be also introduced by enumeration (*bool*) or by induction, as lists (by means of the *datatype* command). Additionally, new types can be also defined as non-empty subsets of already existing types by means of the *typedef* command; the command takes a set defined by comprehension over a given type $\{x :: \alpha . P\,x\}$, and defines a new type σ, as well as *Rep* and *Abs* morphisms between the types. *Type annotations* can be made explicit to the prover, by means of the notation $x :: \alpha$, and can solve situations where types remain ambiguous even after type inference.

Isabelle also introduces type classes in a similar fashion to Haskell; a type class is defined by a collection of operators (over a single type variable) and premises over them. For instance, the library has type classes representing arithmetic operators (like sum or unary minus). Concrete types (*int*, *real*, *set*, and so on) are proved to be *instances* of those type classes. The expression ($x :: \alpha :: plus$) imposes that the type variable α poses the structure and properties stated in the *plus* type class, and can be later replaced exclusively by types which are instances of such a type class. Type classes provide operator overloading, enabling to reuse symbols for different types ($0 :: nat$ and $0 :: int$).

2.2   ACL2

ACL2 [16] stands for "A Computational Logic for 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 [24]. 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 mechanised reasoning in the logic. Thus, the system constitutes an environ-
ment 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 including axioms for propositional logic
and for a number of primitive Common Lisp functions and data types.

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 and ensuring that no inconsistencies are
introduced. The operator `defun-sk` introduces new functions that represent ex-
istential quantifiers, following the idea of *Skolemization*.

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, and so on). Sophisticated heuristics for discovering an (often suit-
able) 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). 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 providing a suitable collection of definitions and
lemmas, used in subsequent proofs as rewrite 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" [16].

2.3   XLL

XLL, for *Xmall Logical Language*, is an XML-based specification language. Its
definition is done through an XML schema [3, Appendix 6.7] which consists of
two parts:

1. A specification of *data types* (or classes), including for each data type a name
   plus a family of operators (or methods).
2. A set of logical statements, expressing some properties of the data types
   involved.

The first part defines a dictionary for the operations that can appear in the
second one. In the second part, XLL defines essentially a typed first-order logic

language. The propositional connectives are grouped in the first part of the XLL schema, the one referring to data types, and will be translated *literally* to any other specification language (for example, ACL2) as *primitive* operations.

With respect to the types in the logical expressions, they can be user-defined classes or elementary data types which can be easily inferred from the context. Only in cases of implicit coercion, some additional type annotations are necessary. For instance, in integer matrix manipulation, the constant 1 can denote either an entry of a matrix or an index for a row or column. In the former case, 1 should be considered as an integer; on the contrary, in the latter, it must be considered as a natural number. These disambiguation annotations are encoded inside the very logical expression, by using enriched arguments like:

```
<constant> <name>1</name> <type>Nat</type> </constant>
```

Additionally, the schema checks that the statements of the properties contain operations that exclusively appear in the XLL file itself (in the specification part); the XLL schema ensures that the properties stated in the file are referred to a certain context (a set of data types and operations).

We have not specified a formal semantics of XLL; it is a simple language in which types, operations and logical statements over them (in a typed first-order logical language) can be expressed. The language is enough to cover both the expressiveness of ACL2, and a first-order fragment of Isabelle/HOL.

In the case study presented in Section 3, XLL documents (that is to say, XML documents compliant with our XLL schema) are generated from Isabelle/HOL formalisations. As an intermediary step, we use a set of libraries generating XML documents from Isabelle specifications, part of the Isabelle standard distribution. Namely, we generate a collection of XML files from an Isabelle/HOL theory, which are subsequently transformed into an XLL file. Furthermore, from that XLL document an ACL2 set of statements can be also generated, essentially forgetting the data types part, because ACL2 is an environment without explicit static typing; nevertheless, the type annotations in the logical expressions are used to generate predicates checking dynamically ACL2 types, as we will explain later. From the XLL document, we are able to produce an Isabelle theory, and automatically prove (in Isabelle!) the behavioural equivalence between the generated Isabelle theory (from the XLL document) and the original Isabelle theory — see [3] for an example. However, it is not possible to reconstruct the Isabelle theory from the produced ACL2 specification, because, having ACL2 a weaker type system than Isabelle/HOL, we irretrievably lost information in the translation.

Each one of the previous steps is automatic, except the initial choice of the types, operations and lemmas which are of interest for our development (types and operations dependencies are also solved by the tool). The user is in charge of choosing the definitions (and lemmas) that will be exported, and she has to decide what is the correct level of granularity to export a set of functions (and lemmas) detailed-enough to be useful for the proof-scheme, but also abstract-enough to give a proof-scheme independent from the concrete representation of

the source theorem prover. This is why we have labelled the whole generation process as *almost* automatic.

3   Transforming an Isabelle/HOL formal development to ACL2: a diagonal matrix form

In this section, we apply the previously defined interoperability setting to an Isabelle/HOL formalisation of some well-known results about integer matrices. It is important to highlight that, even if the theory is written in HOL, the problem is essentially of a first-order nature, and therefore the information that is lost when going from Isabelle/HOL to XLL (and then to ACL2) does not prevent us from getting a sensible specification. Thus, we consider an Isabelle/HOL development (described in [3]) which defines a verified method to reduce a given matrix to a diagonal form, i.e. a method to compute a diagonal matrix which is *equivalent* to the initial one — two matrices A and B are equivalent if there exist two invertible matrices P and Q such that $B = PAQ$.

Then, the corresponding Isabelle/HOL formalisation includes the basic matrix operations (addition and multiplication) and properties of the ring of integer matrices. The main result of this Isabelle/HOL theory can be expressed as follows:

Lemma 1. *Given an integer matrix A, there exist three integer matrices P, Q and B such that:*

- *$B = PAQ$;*
- *P and Q are invertible matrices;*
- *B is a diagonal matrix.*

The diagonal matrix presented in the previous lemma is usually computed in many algorithms as an intermediary step in the computation of the Smith Normal Form (see [5, 7]); indeed, this particular matrix has its own interesting properties.[1]

3.1   An Isabelle/HOL formalisation of Lemma 1

Let us briefly describe the Isabelle/HOL formalisation that leads us to prove Lemma 1 (the interested reader can find a more complete description in [3]).

One of the most relevant decisions in the initial steps of a formalisation is the choice of a suitable representation for the objects involved in the development; in this particular case, integer matrices. In our Isabelle/HOL theory, the family

---

[1] In spite of the fact that the calculation on the Smith Normal Form is a more renowned result than the one presented in Lemma 1, the diagonal form is enough in many calculations. For example, the homology of a chain complex over a ring can be obtained using the diagonal form of the differential maps represented as matrices. This situation is usual in some programs for Symbolic Computation in Algebraic Topology; thus, the presented result has its own interest in that area.

of matrices is represented as the set of functions with two arguments of type
`nat` and finitely many non-zero positions. This functional representation eases
the definition of operations over matrices and the proof of properties (it has
been introduced in Isabelle, and successfully used, as a part of the Flyspeck
project [20]). The formal definition is:

```
type_synonym 'a infmatrix = "nat => nat => 'a"

definition nonzero_positions ::
        "('a::zero) infmatrix ⇒ (nat x nat) set" where
        "nonzero_positions A = {pos. A (fst pos) (snd pos) ~= 0}"

definition "matrix = {(f::(nat ⇒ nat ⇒ 'a::zero)).
                      finite (nonzero_positions f)}"

typedef 'a matrix = "matrix :: (nat ⇒ nat ⇒ 'a::zero) set"
```

The library offers several definitions and properties over this data type;
in particular, operations *nrows* and *ncols* that, making use of the Hilbert's
ϵ operator, return the maximum row and column which contain nonzero el-
ements. We had to define elementary operations on matrices; in particular,
there are two basic operations for our development (*interchange_rows_matrix*
and *row_add_matrix*) that exchange two rows of a matrix, and replace a row by
the sum of itself and another row multiplied by an integer respectively (the corre-
sponding operations acting on columns are also defined). We introduce here the
definition of *interchange_rows_matrix*, as well as the definition of its *functional*
behaviour over the *underlying* representation of matrices presented previously
(in this case, *functions*):

```
definition interchange_rows_infmatrix ::
        "int infmatrix ⇒ nat ⇒ nat ⇒ int infmatrix"
  where "interchange_rows_infmatrix A n m ==
      (λi j. if i=n then A m j else if i=m then A n j else A i j)"

definition interchange_rows_matrix ::
        "int matrix ⇒ nat ⇒ nat ⇒ int matrix"
  where "interchange_rows_matrix A n m==
        Abs_matrix (interchange_rows_infmatrix (Rep_matrix A) n m)"
```

The previous definition relies on the type morphisms `Abs_matrix` and
`Rep_matrix`, which perform the conversion between the type (`matrix`) and the
underlying type (`infmatrix`, an abbreviation of the functional representation
of matrices). It makes use of the function `interchange_rows_infmatrix`, which
represents the functional behaviour of the elementary operation.

Using these functions (and their column counterparts), we can define several
auxiliary results and finally state and prove Lemma 1 in Isabelle/HOL.

```
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))"
```

The proof of this result (from a conceptual point of view) is *constructive*; that is, the witnesses (P, Q and B in this case) for the existential expression (it applies the existential quantifier to P, Q and B) are explicit and could be algorithmically produced. Roughly speaking, the procedure used in the proof to compute the equivalent diagonal matrix is analogous to the Gauss-Jordan elimination method [8, Section 28.3].

The fact that the essence of the proof is constructive would allow us to obtain executable programs from it. However, it is not possible to directly execute the corresponding expressions inside Isabelle due to the representation of matrices (based on the abstract type *matrix*). It is well-known that, in such a situation, we could *refine* the data structure representation (by taking, for instance, lists of lists to represent matrices), in order to get executable code from Isabelle. Our approach is however different: we look for a proof in a different theorem prover (ACL2), where executability will be guaranteed.

3.2   From the Isabelle/HOL theory to XLL

As a first step in the translation of the Isabelle/HOL theory to ACL2, the corresponding XLL document — an XML instance compliant with the XLL schema — is generated through a series of automatic steps, as presented in [3]. The XLL description of the theory consists of two different components: data types and logical statements.

The former contains the specification of the data types appearing in the source Isabelle/HOL theory (their names and the collection of functions where the types appear as parameters, which are selected by the tool). This information is automatically organized in a *class*, an XLL structure which is used to represent each type and its operations (and that in our Ecore experiments is later assigned to a UML class), whose XLL description (for the type `matrix`) appears in Figure 2 (we only include elementary operations over matrices).

The second component of the resulting XLL document consists of a set of statements establishing the properties of the entities (data and methods) involved in the theory. To illustrate this component, we include in Figure 3 the XLL description of the lemma which states in Isabelle/HOL the idempotence property of `interchange_rows_matrix` — the square of this function is the identity.

```
lemma interchange_rows_matrix_id:
 shows
 "interchange_rows_matrix (interchange_rows_matrix A n m) n m = A"
```

```
<Class name="Matrix.matrix">
  <Class_Parameters>
    <Parameter name="alpha">
      <Type name="Int.int"/>
    </Parameter>
  </Class_Parameters>
        ...
  <method name="Diagonal_form.interchange_rows_matrix">
    <Type name="Matrix.matrix"/>
    <Input name="n"><Type name="Nat.nat"/></Input>
    <Input name="m"><Type name="Nat.nat"/></Input>
  </method>
  ...
</Class>
```

Fig. 2. XLL for the generated `matrix` class.

3.3   From XLL to ACL2

From the XLL description of the Isabelle/HOL theory, and only applying XSLT transformations [25], we can obtain an ACL2 specification. For instance, the ACL2 function obtained from the XLL file of Figure 2 is:

```
(defun Diagonal_form.interchange_rows_matrix (A n m)
    (declare (ignore A n m)) nil)
```

The body of this function is empty (in fact, the value nil is returned by default to allow the compilation of the function) since data types are very linked to the way of working in each proof assistant. Therefore, it is unlikely that the Isabelle/HOL representation of matrices will be the most useful one to work in ACL2. Then, we delegate the task of defining a suitable representation of the data types to a further step in the development process (see Section 4).

In the same way, theorems like the one presented in Figure 3 are also translated to ACL2.

```
(defthm interchange_rows_id
    (implies
      (and (matrix_integerp A) (natp n) (natp m))
      (equal (Diagonal_form.interchange_rows_matrix
            (Diagonal_form.interchange_rows_matrix A n m) n m) A)))
```

Using this procedure, we translate the whole Isabelle/HOL development into ACL2. The ACL2 version of Lemma 1 is stated as follows.

```
(defun-sk exists_Diagonalize_theorem (A)
  (exists (P Q B)
      (and (Diagonal_form.is_invertible P)
```

```
<Theorem>
    <name>interchange_rows_id</name>
    <forall>
      <param> <name>A</name> <type>Int.int Matrix.matrix</type> </param>
      <body>
       <forall>
          <param> <name>n</name> <type>Nat.nat</type> </param>
          <body>
            <forall>
              <param> <name>m</name> <type>Nat.nat</type> </param>
              <body>
                <operation>
                  <name>HOL.eq</name>
                  <operation>
                    <name>Diagonal_form.interchange_rows_matrix</name>
                    <operation>
                      <name>Diagonal_form.interchange_rows_matrix</name>
                      <constant> <name>A</name> </constant>
                      <constant> <name>n</name> </constant>
                      <constant> <name>m</name> </constant>
                    </operation>
                    <constant> <name>n</name> </constant>
                    <constant> <name>m</name> </constant>
                  </operation>
                  <constant> <name>A</name> </constant>
                </operation>
              </body>
            </forall>
          </body>
        </forall>
      </body>
    </forall>
  </Theorem>
```

Fig. 3. XLL for the interchange_rows_id theorem.

```
        (and (Diagonal_form.is_invertible Q)
        (and (equal B (Groups.times_class.times
                       (Groups.times_class.times P A) Q))
        (and (Diagonal_form.is_square P (Matrix.nrows A))
        (and (Diagonal_form.is_square Q (Matrix.ncols A))
        (Diagonal_form.Diagonalize_p B
           (max (Matrix.nrows A) (Matrix.ncols A))))))))))))

(defthm Diagonalize_theorem
  (implies (matrix_integerp A)
           (exists_Diagonalize_theorem A)))
```

We claim that the previous statements, transferred from the Isabelle/HOL formal development, can be used as a guideline to achieve a similar formalisation in ACL2. The following section includes a small example illustrating this fact.

4 An experiment in reusing (schemes of) proofs

As a driving example to translate proof schemes, we consider a small theory about the basic operations over matrices described in Section 3. In particular, we are interested in proving that the elementary matrices $P_{i,j}$ (identity matrices where the i-th and j-th rows are swapped) are invertible. The actual statement of the lemma explains that the square of a $P_{i,j}$ (encoded using the function P_ij) matrix is the identity matrix.

```
lemma P_ij_invertible:
  assumes n: "n < a" and m: "m < a"
  shows "(P_ij a n m) * (P_ij a n m) = one_matrix (a)"
```

The main components of the theory required to prove the above lemma are the functions:

– interchange_rows_matrix, that exchanges two rows of a matrix;
– P_ij, that defines the elementary matrix P_{ij} in dimension a.

```
definition P_ij :: "nat ⇒ nat ⇒ nat ⇒ int matrix"
  where "P_ij a n m ==
        interchange_rows_matrix (one_matrix a) n m"
```

and the lemmas:

– interchange_rows_matrix_id, which states the idempotency of the function interchange_rows_matrix (see the end of Subsection 3.2).
– PA_interchange_rows, that relates the interchange_rows_matrix operation to the left product by the P_ij matrices.

```
lemma PA_interchange_rows:
  assumes n:"n < nrows A" and m: "m < nrows A"
          and na: "nrows A <= a"
  shows "interchange_rows_matrix (A::int matrix) n m =
        (P_ij a n m) * A"
```

This specification can be considered as a suitable strategy to prove lemma P_ij_invertible in Isabelle/HOL and, as we will show, the same strategy can be replicated in ACL2 to prove the same result.

We omit the XLL instance provided by this source Isabelle/HOL theory (the interested reader can extract it from [3]) to directly present the ACL2 specification which is automatically produced by the I2EA framework. The file generated

by the I2EA framework consists of two parts: the headers of the functions and the lemmas.

In the function section, we find not only the specification of the functions defined in the Isabelle/HOL theory, but also all the functions involved in the theorems of such a theory which are defined in other libraries (for instance, the definition of the identity matrix `Matrix.one_matrix`), and also predicate recognisers (in this case, the `matrix_integerp` function is the recogniser for integer matrices), which replace in ACL2 some Isabelle typing information:

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

(defun Diagonal_form.interchange_rows_matrix (x1 x2 x3)
    (declare (ignore x1 x2 x3 )) nil)

(defun Matrix.nrows (x1)
    (declare (ignore x1)) nil)

(defun Groups.times_class.times (x1 x2)
    (declare (ignore x1 x2)) nil)

(defun Diagonal_form.P_ij (x1 x2 x3)
    (declare (ignore x1 x2 x3)) nil)

(defun Matrix.one_matrix (x1)
    (declare (ignore x1)) nil)
```

Using the headers of these functions as a guideline, we must provide a concrete representation for integer matrices and define the rest of the functions — we re-use an ACL2 matrix library presented in [13], where matrices are encoded as lists of vectors, and several background lemmas are provided; in addition, ACL2's pre-defined functions are used to define the body of some functions (e.g. the "∗" ACL2's function is used to define the body of the function `Groups.times_class.times`).

Once this task is carried out, we can focus on the lemmas generated by the I2EA framework.

```
(defthm interchange_rows_id
  (implies (and (matrix_integerp A) (natp n) (natp m))
           (equal (Diagonal_form.interchange_rows_matrix
                     (Diagonal_form.interchange_rows_matrix A n m)
                    n m) A)))

(defthm PA_interchange_rows
  (implies (and (natp n) (matrix_integerp A) (natp m) (natp a)
                (< n (Matrix.nrows A)) (< m (Matrix.nrows A))
                (<= (Matrix.nrows  A) a))
```

```
          (equal (Diagonal_form.interchange_rows_matrix A n m)
                 (Groups.times_class.times
                              (Diagonal_form.P_ij a n m) A))))
```

```
(defthm P_ij_invertible
   (implies (and (natp n) (natp a) (natp m) (< n a) (< m a))
            (equal (Groups.times_class.times
                       (Diagonal_form.P_ij a n m)
                       (Diagonal_form.P_ij a n m))
                   (Matrix.one_matrix a))))
```

ACL2 is able to find the proof of the first two lemmas without any external help; however, it gets stuck when proving lemma `P_ij_invertible`. We can suggest ACL2 to use the lemmas `PA_interchange_rows` and `interchange_rows_id` to finish the proof, but the system is not able to use them. Inspecting ACL2's proof attempt, we realise that ACL2 needs a lemma which states that the function `Diagonal_form.P_ij` generates an integer matrix.

```
(defthm P_ij_matrix_integerp
   (implies (and (natp a) (natp n) (natp m))
            (matrix_integerp (Diagonal_form.P_ij a n m))))
```

Once this lemma is introduced in the system, ACL2 finishes the proof of `P_ij_invertible`. Let us note that `P_ij_matrix_integerp` is taken for granted in Isabelle/HOL, since this type information is already provided in the definition of `P_ij`.

As foreseen, the previous discussion shows that we can import the Isabelle/HOL proof scheme into ACL2, but some additional lemmas can be necessary to complete the proof — in our experiments, those auxiliary lemmas are always related to predicate lemmas such as `P_ij_matrix_integerp`. These ACL2 lemmas containing the information encoded in the Isabelle functions target types, in the form of recognisers, will be automatically generated in future releases of the I2EA framework. The case study on matrices has proven itself useful to give us feedback on the kind of information that is represented differently in Isabelle and ACL2, but still necessary on both tools.

5   Conclusions and future work

In this paper, we have described a facility to transform Isabelle/HOL theories into ACL2 specifications. We have shown, through a concrete case study, that the transferred-information is enough to reconstruct a proof in ACL2. In particular, the original Isabelle theory consists of 5952 lines of code, 222 lemmas, and 54 definitions. Those lemmas and definitions have been filtered to extract 119 lemmas, and 32 definitions that have been translated to ACL2. Finally, the ACL2 development consists of 58 definitions (19 of them using defun-sk) and 119 lemmas.

The drawbacks of our approach (with respect to other mainstream approaches to interoperate between theorem provers) are the following ones:

- Our proposal is not *universal*, in the sense that it is not proposed as a general solution to the interoperability problem.
- Our proposal is *partial*, because when going from Isabelle/HOL to ACL2, it is evident that no higher-order Isabelle theory could be translated to a, necessarily first-order logic, ACL2 specification.
- Our proposal is *incomplete* (even for the fragment of Isabelle/HOL that we are considering), since we port only function signatures and statements, while definitions and proofs are not transferred in the process.
- Our proposal requires the expert knowledge of the user to choose the relevant lemmas and definitions to generate a useful proof-scheme.

On the positive side, the benefits of the presented framework are:

- Our proposal was developed *quickly* (at least when comparing it with the effort required to embed a system in another one); in the implementation we used many already available XML tools, reducing the programming needs to a minimum [3].
- Our proposal is *flexible*, because due to the lightweight technology used, we have been able to modify our XLL schema to adapt it to other close situations, without reprogramming the whole framework, see [14].
- And last, but not least, our proposal *works*, since we have shown how a nontrivial formalisation (a diagonalisation algorithm for integer matrices) has been translated profitably to ACL2, as required in our ForMath setting.

We think that the global balance is positive. More research and experiments are needed in order to get more evidences of the interest of this kind of *shallow* interoperability approach. As future work, we should translate other (first-order like) Isabelle/HOL theories to ACL2; for instance, it would be interesting to study algorithms for symbolic matrices presented in [22]. Moreover, we should generalise our approach to other proof assistants. In this last line, some successful experiences have been already made: we have used XLL as intermediary language to port Coq statements to ACL2 in a context of Java programming verification, see [14].

References

1. ForMath: Formalisation of Mathematics, European project. `http://wiki.portal.chalmers.se/cse/pmwiki.php/ForMath/ForMath`.
2. MDT/OCL in Ecore. `http://wiki.eclipse.org/MDT/OCLinEcore`.
3. J. Aransay et al. A report on an experiment in porting formal theories from Isabelle/HOL to Ecore and ACL2. Technical report, ForMath European project, 2013. `http://wiki.portal.chalmers.se/cse/uploads/ForMath/isabelle_acl2_report`.

4. C. Benzmüller et al. THF0 — The Core of the TPTP Language for Higher-Order Logic. In *IJCAR'08*, volume 5195 of *LNCS*, pages 491–506, 2008.
5. G. H. Bradley. Algorithms for Hermite and Smith Normal Matrices and Linear Diophantine Equations. *Mathematics of Computation*, 25(116):897–907, 1971.
6. M. Codescu et al. Towards Logical Frameworks in the Heterogeneous Tool Set Hets. In *WADT'10*, volume 7137 of *LNCS*, pages 139–159, 2012.
7. H. Cohen. *A Course in Computational Algebraic Number Theory*. Springer, 1995.
8. T. H. Cormen et al. *Introduction to Algorithms*. McGraw-Hill, 2003.
9. S. Cruanes et al. Tool integration with the evidential tool bus. In *VMCAI'13*, volume 7737 of *LNCS*, pages 275–294, 2013.
10. E. Denney. A Prototype Proof Translator from HOL to Coq. In *TPHOLs'00*, volume 1869 of *LNCS*, pages 108–125, 2000.
11. G. Gonthier and A. Mahboubi. An introduction to Small Scale Reflection in Coq. *Journal of Formalized Reasoning*, 3(2):95–152, 2010.
12. M. J. C. Gordon et al. The Right Tools for the Job: Correctness of Cone of Influence Reduction Proved Using ACL2 and HOL4. *Journal of Automated Reasoning*, 47(1):1–16, 2011.
13. J. Hendrix. Matrices in ACL2. In *ACL2'03)*, 2003.
14. J. Heras et al. Verifying a platform for digital imaging: a multi-tool strategy. In *CICM'13*, volume 7961 of *LNCS*, pages 66–81, 2013.
15. M. Jacquel et al. Verifying B Proof Rules Using Deep Embedding and Automated Theorem Proving. In *SEFM'11*, volume 7041 of *LNCS*, pages 253–268, 2011.
16. M. Kaufmann et al. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.
17. C. Keller and B. Werner. Importing HOL Light into Coq. In *ITP'11*, volume 6172 of *LNCS*, pages 307–322, 2011.
18. P. Naumov et al. The HOL/NuPRL Proof Translator - A Practical Approach to Formal Interoperability. In *TPHOLs'01*, volume 2152 of *LNCS*, pages 329–345, 2001.
19. T. Nipkow et al. *Isabelle/HOL: A proof assistant for Higher-Order Logic*. Springer, 2002.
20. S. Obua and T. Nipkow. Flyspeck II: the basic linear programs. *Annals of Mathematics and Artificial Intelligence*, 56(3-4):245–272, 2009.
21. S. Obua and S. Skalberg. Importing HOL into Isabelle/HOL. In *IJCAR'06*, volume 4130 of *LNCS*, pages 17–20, 2006.
22. A. P. Sexton et al. Computing with Abstract Matrix Structures. In *ISSAC'09*, ACM, pages 325–332, 2009.
23. J. H. Siekmann et al. Proof Development with OMEGA. In *CADE-18*, volume 2392 of *LNCS*, pages 144–149, 2002.
24. G. L. Steele. *Common Lisp the Language*. Digital Press, 1990.
25. W3C. XSLT 2.0. `http://www.w3.org/TR/xslt-xquery-serialization/`.