Transforming (almost) without programming: a case of interoperability among theorem provers[☆]

J. Aransay[a], J. Divasón[a], J. Heras[b], L. Lambán[a], V. Pascual[a], A. L. Rubio[a], J. Rubio[a]

[a] *Departamento de Matemáticas y Computación, Universidad de La Rioja, Spain*
[b] *School of Computing, University of Dundee, UK*

Abstract

In this paper, we report on an experiment in porting formal theories among proof assistants. Specifically, a formalization developed in Isabelle/HOL is partially translated into ACL2. The innovative features of our approach are twofold. On the one hand, it is essentially based on XML transformation technology (and so, it is almost "programming-free"); on the other hand, we provide also an integration with model specification facilities based on UML + OCL (the first aspect is not independent from the second one). Our motivating case study is an Isabelle formalization on diagonalizing integer matrices, whose "proof schema" has been automatically translated into ACL2.

1. Introduction

A well known issue in the field of automated theorem proving is the interoperability among tools. Communities working with different provers make very important efforts to define structures, proofs and libraries. The workload involving these efforts has promoted research in at least two directions: comparison among provers and collaboration among them. On the one hand, when the same proof can be carried out in two different provers is important to be able to compare the obtained results, both in terms of the techniques and tactics used as in terms of the efficiency and accuracy of the proof. On the other hand, the complexity of some of these proofs suggests the interest of being able to transfer, even partially, both structures and proof processes among different provers. The existence of means for transferring these artifacts among provers could facilitate a comparison among them, and it could improve the throughput of the efforts made on each of the tools.

In this paper, we present an experimental proposal for interoperating between two particular provers. This approach follows some important driving

[☆]Partially supported by Ministerio de Ciencia e Innovación, project MTM2009-13842, by European Union's 7th Framework Programme under grant agreement nr. 243847 (ForMath), and by Universidad de La Rioja, research grant FPI-UR-12.

principles. First, it is not intended to be a universal solution. The diversity and heterogeneity of the different provers, both in purely syntactic aspects as in the underlying logics, make more feasible an approach which considers specific transformations capturing the particularities of a domain of application. In our particular case, we will address interoperability from Isabelle to ACL2 in a case study dealing with matrix manipulation. More specifically, we will start from an already-built development in Isabelle/HOL and we will obtain its corresponding proof schema in ACL2. Even if the initial theory is implemented in Higher-Order Logic, the very nature of the problem, related to matrix properties, allows us to translate it safely into a First-Order Logic tool as ACL2.

A second key objective of our approach is to minimize programming tasks, since they are time consuming and error prone. We think that these two aspects (trying to get universal tools, and programming all the interoperability from scratch) are the most important reasons why many integration experiments have been uncompleted in the past. This second goal is achieved by using well known tools and standards, especially those based on XML. The absence of big programming efforts decreases the importance of not proposing a universal structure; our current experiment is not an *ad-hoc* development, since it can be applied to other close situations, by simply adapting our XML resources.

Finally, a distinguishing feature of the proposal is that it is not limited to interoperability between provers, but it seeks integration with model-based components. In our application domain (that of *mathematics formalization*) much of the proving strategy is based on the design of data structures: a carrier, some operators, and then the statements are essentially equalities among expressions. Even if theorem provers in general (and Isabelle and ACL2 are no exception) privilege the functional presentation, it is quite standard now to move from functional to (side-effect free) object-oriented specifications. In that way, the basic structures that must be defined to implement a proof in a theorem prover are very close to object-oriented models used for decades in the software engineering community. In our opinion, it would be very interesting to have a way to combine both worlds, so that some of the many tools available in the field of software engineering can also serve to the theorem proving community. To achieve this goal, we propose to transform Isabelle specifications to Ecore/OCL models processed using EMF.

In order to get all these different components combined, we present an intermediate language, called XLL (standing for Xmall Logical Language). This language, defined by an XML schema, is provided with the minimum essential elements such that it is possible to transfer an (first-order like) Isabelle development both to ACL2 as to Ecore/OCL, capturing the basic logical structure of each system. Importantly, we have designed the language XLL to have only the essentials (Xmall = XML + small) required to achieve interoperability between systems.

The paper is structured as follows: in the next section we present briefly each one of the three systems involved in the interoperability setting (Isabelle, ACL2 and EMF); in Section 3 we describe our interoperability architecture, devoting Section 4 to the explanation of the XLL language. Section 5 will show

a case study related to matrix manipulation, finishing with some conclusions and future work.

2. Overview of tools

In this section, we provide a short description of the main characteristics of the three tools involved in our interoperability setting: Isabelle, ACL2 and EMF.

2.1. Isabelle

Isabelle [10] 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 functional 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 [3], also known as simple type theory.

A detailed description of the metalogic can be found in [11]. 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.
- A set of inference rules which act over terms of type *prop*, and that express the properties of the connectors of the metalogic.

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 (HOL), in which we will focus, since our later developments will be mainly carried out on top of it. HOL is the most widely used logical setting by the Isabelle community, so that usually Isabelle/HOL is commonly referred as Isabelle. In order to define HOL over the metalogic, one must define both a type system, capturing the properties of the HOL type system, and a set of axioms (or rules) which define our logical system.

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 consists in the successive application of tactics or tacticals (functions mapping the statement to one or various easier statements) that must lead to the original premises or to a trivial *true* statement; thus, proofs are developed in a backward style. Then, another language called *Isar* 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.

2.2. ACL2

ACL2 [8] 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 [13]. 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. The logic includes axioms for propositional logic and for a number of primitive Common Lisp functions and data types.

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, and so on). 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). Definitions made with `defun`, statements introduced by `defthm`, and other definitional entities (as the *existential* definitions built with `defun-sk`) are called *events* in ACL2 terminology.

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 [8].

Executability is a relevant feature of ACL2: 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.

2.3. EMF

The Eclipse Modeling Framework (EMF) project [12] is a complex Eclipse plugin that provides a framework for describing models and then generating other models or code from them. In fact, modeling and programming can be considered the same thing when working with EMF. This framework brings them together as two well-integrated tasks of the same job, because EMF unifies three important technologies: Java, XML and UML. An EMF model integrates the three technologies and can be defined using either of them.

The (meta)model used to represent models in EMF is called Ecore. Ecore is a simplified and small subset of UML (full UML supports much more ambitious modeling than the core support in EMF). Ecore 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. Despite having this variety of representations, it is well known that UML models (and consequently Ecore models) have certain limitations in expressing some kinds of logical constraints. For that reason, Ecore models can be enriched with OCL constraints [14]. This possibility becomes a requirement in our framework, as in order to get interoperability with automated provers it is essential to be able to express complex constraints. In particular, EMF provides the technological support which allows us to validate a particular instance of an Ecore model annotated with OCL constraints [9].

In addition, the EMF community has developed different subprojects that enable the development of models repositories (see, for example the CDO or Teneo projects [4]), which in our context could be used to develop repositories of specifications and proofs.

3. An architecture for interoperability

Figure 1 displays the main components of our architecture for interoperability. There are several outstanding features of this architecture that we explain in the rest of this section. These features are the main components, the role of programming and the intermediate artifacts (with a special emphasis in the XLL component).

3.1. Main components

The main components of the architecture are Isabelle, ACL2 and EMF. Within our setting, we start from an Isabelle theory that is ported to both an ACL2 program and an Ecore/OCL model. An Isabelle theory includes essentially four parts:

1. Some datatypes representing the concepts where the theory is carried out.
2. Some definitions of functions.
3. Statements of properties.

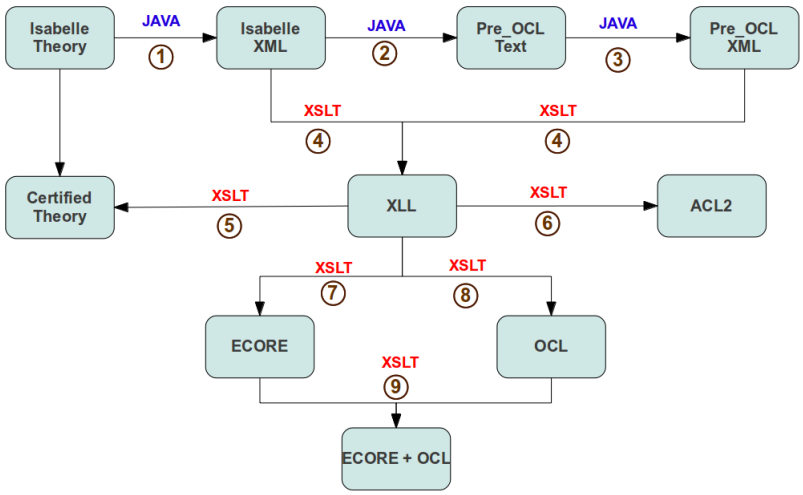

Figure 1: Architecture diagram.

4. The proofs of the statements.

Let us illustrate these four components with a simple example dealing with *lists*. First, we define in Isabelle the corresponding datatype:

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

Then, we define an operation to *concatenate* lists:

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

Finally, we state the associativity of *append*, and prove it in Isabelle:

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

In order to get the ACL2 output, we are mainly interested in the two central items: definitions and statements. Our final objective is to get a series of statements in ACL2 replaying the proof developed in Isabelle. To this aim, we need some knowledge about the functions involved (Item 2 in the previous enumeration). Since ACL2 is untyped, to produce ACL2 headers, we only need in principle to know its arity, which can be directly inferred from the statements. Let us stress that the *bodies* of the functions are not necessary to translate the statements; furthermore, it would not be possible to obtain the bodies without knowing the data types representation, an information which is missing in our

transforming schema. Nevertheless, due to the fact that ACL2 functions must be total, it is usually needed to introduce predicates to state that free variables or constants appearing in operations and formulas belong to the appropriate domains (for instance, that an index for a list or a matrix position should be a natural number); these premises and typing 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 delegate these internal design decisions, which are crucial in the final development of a proof, to the ACL2 user. 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.

It is worth mentioning that 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 implementations or representations from Isabelle is not required for this transformation (nor for ACL2). In order to complete the Ecore model, we also use theorems' statements, since they will be translated as OCL restrictions in that model.

3.2. The role of programming

As announced from the very title of the paper, programming is restricted to a minimum. It is true that designing XSLT sheets could be identified as a kind of *declarative* programming, but it should be also clear that this avoids most of the *dangerous* errors that we associate to the task of programming in C, C++, Java and the like (as no reward comes without a price, the debugging task is rather painful in XSLT).

We note that at Figure 1 three arrows (numbered 1, 2 and 3) are labeled with "Java". However, all these steps are auxiliary, utility processes. For instance, in Step 2, we find that each single theorem, datatype definition, or operation in Isabelle gives place to an XML file (this is made by a facility integrated in the standard Isabelle distribution). In general, it will be necessary to convert a multitude of theorems' statements, data type information, definitions... to XML, so we automated this process by making a Java program to perform this task. This very simple 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.

As shown in Figure 1, the essential part of the transformation and translation processes in our architecture is performed by using XSLT (Steps 4 to 9). Extensible Stylesheet Language Transformations (XSLT) is a declarative language for transforming XML documents into other XML documents (or other objects such as HTML for web pages or plain text). To introduce it briefly: an 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. 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 [16], which is designed to address the common problem of how composing XML processes. In our case, we use XProc to make consecutive XSLT transformations; more specifically, a pipeline of transformations, where the output of the k-th transformation is the input to the $(k+1)$-th one.

3.3. Intermediate artifacts

The heterogeneity of the artifacts handled in each tool involved is one of the key challenges that must be addressed in an interoperability context as the one we are presenting. For this reason, it is essential to carry out the transformation process into several simpler steps that allows us to adapt the technical requirements (input and output formats, file structure, and so on) incrementally.

In our case, we have divided the entire process into several steps using different intermediate artifacts. For example, 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 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 among some of the intermediary languages developed and the original theorems. One of these XML files 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 following an XML structure through an XML Schema (see Appendix 6.2 in [1]). This Pre_OCL XML file is completed (Step 4) with the specification of the datatypes used in the Isabelle theory (typedefs, definitions, functions, and so on) to obtain an XLL document. This document, written in the XLL language, is the keystone of the architecture, so we will explain it in the following section.

4. The specification language XLL

In this section, we introduce an XML-based specification language called *XLL* (for *Xmall Logical Language*). The language XLL is defined through an XML schema which consists of two parts:

1. A specification of *datatypes* (or classes), including for each datatype a name plus a family of operators (or methods); this part is essentially imported from the schema of Isabelle XML.
2. A set of logical statements, expressing some properties of the datatypes involved.

The aim of the first layer is twofold. First, it is used to define UML classes in an Ecore model, providing a context for the OCL restrictions which will be generated. As a second purpose, it defines a dictionary for the operations that can appear in the logical part. The second section of XLL defines a kind of First-Order Logic language, which is enough to cover both the expressiveness of ACL2 and the corresponding subset of OCL.

To illustrate these aspects, let us resume the example of concatenation of lists. The XLL expression corresponding to the Isabelle code in Subsection 3.1 is presented in Figure 2. If the cumbersome XML tags are dropped out, an expression equivalent to that of Figure 2 could be:

$$\forall xs, ys, zs : a\ List,\ append(xs, append(ys, zs)) = append(append(xs, ys), zs)$$

In the previous expression, in addition to some general logical constructs (quantifier, equality, and so on), the only operation is *append*. This implies that, in the datatype section, there should be a type or class (*List*) endowed with an operator (or method) named *append*. The first part of our XLL schema deals with this kind of information.

As for the syntax of the expressions, XLL defines (in its second part) essentially a typed First-Order Logic language. Following ideas from Isabelle XML, the quantifiers ($\forall$, $\exists$) are dealt with separately, since they have a *binding scope* (a body). The rest of operations are treated homogeneously. This includes both the user defined methods (as *append* in the example) and the propositional logical connectives (considering as such the equality symbol, too). To make homogeneous the processing, the propositional connectives are grouped in the first part of the XLL schema (the one referring to datatypes) into a class called *PL* (for *Propositional Logic*). These *primitive* operations will be translated *literally* to any other specification language (OCL, ACL2 and so on).

From an XML technical point of view, the only aspect deserving attention is the necessary link between the names of methods in the datatypes part and the names of the operations in the logical expressions: the latter must be chosen among the former. Since the set of names is not a constant, it is necessary to use the XML tools `key` and `keyref`, to refer on each concrete XLL document to the family of operation names. This family is not always located at the same place, and it must be collected by seeking over each concrete document instance. The Isabelle XML schema contains enough information to recover that collection, but this task would need complex XPATH instructions. Unfortunately, the standard *XML-Schema* (or XSD, in short) only allows us to use a subset of XPATH [15]. This is the reason why the Isabelle XML schema has not been fully reused. We need to redefine a new schema where the names of methods are more explicitly presented, in such a way that the required XPATH commands are simple enough to be employed in combination with XSD. Then, a small XProc script (grouping four simple XSLT sheets) allows us to obtain an XLL datatype specification from several Isabelle-XML documents.

With respect to the types in the logical expressions, they can be user-defined classes or elementary data which can be easily inferred from the context. Only

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

Figure 2: XLL for the associativity of *append*.

in cases of implicit coercion, some additional type annotations are necessary. For instance, in our driving example (integer matrix manipulation), a constant as the number 1 can denote either an entry of a matrix or an index for a row or column. In the first case, 1 should be considered as an integer; on the contrary, in the second one, 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>
```

The complete XLL schema can be found in Appendix 6.7 of [1]. In the architecture presented in the previous section, XLL documents (that is to say, XML documents compliant with our XLL schema) are generated from Isabelle formalizations. Then, from an XLL document we can obtain an Ecore/OCL model: the datatype part provides the UML context where interpreting the constraints corresponding to the XLL expressions (Step 7 in Figure 1). Furthermore, from the same XLL document an ACL2 set of statements can be also generated (Step 6), essentially forgetting the datatypes 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.

5. Case study: a diagonal matrix form

As explained previously, our main driving example was a formalization developed in Isabelle/HOL about integer matrices manipulation. Precisely, the undertaken problem was the modeling of a procedure to reduce an integer matrix to a *diagonal form*. The algorithm is quite standard, a variant of Gauss elimination, adapted to a situation where the entries of matrices range over a ring, instead of over a field (see, for instance, [2]).

The main result looks as follows. 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 corresponding Isabelle statement is:

```
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))"
```

One important property of the developed proof is that it is *constructive*. That is to say, the existential quantifier about P and Q (and then B) is proved by giving explicit *witnesses* P and Q, produced in an algorithmic way. In such a

situation, a good feature is to get executable programs from the proof. However, using the full power of HOL implies, in addition to ease the life of the (human) prover, that some constructions are not amenable directly to an executable version. In our case, the most prominent aspect is the representation of matrices as functions whose range (of nonzero elements) is finite; this representation prevents us from evaluating them inside Isabelle.

The traditional way of dealing with this setting [6] is to transform the formalization in order to generate (Haskell or ML) programs, which can be then executed. Our approach is different. Instead of generating code in a programming language, we generate the schema of a proof in a different proof assistant (namely, ACL2). Being ACL2 also a programming language, an ACL2 proof (when completed) will allow us to obtain executable programs, with ensured correctness.

The ACL2 output corresponding to the Isabelle main statement consists of two ACL2 events:

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

The first function definition establishes the conditions where the main theorem must be true, determining the witnesses for the matrices P, Q and B. Note that the identifiers naming the different functions are constructed by using the natural terminology in Ecore/OCL (and therefore in XLL, too). Let us also stress that the statement of the theorem is *conditional*: even if ACL2 (as Common Lisp) is a (statically) untyped language, a type dynamic predicate (imposing that the input A is an integer matrix) is required.

6. Conclusions and further work

In the frame of the ForMath European project [5], several proofs assistants are used to develop libraries of formalized mathematics, with the objective, in particular, of providing enough resources to verify mathematical software. Thus, the provers Coq, Isabelle/HOL and ACL2 have been useful (each one has its own strengths and weaknesses) to formalize different subjects in the project.

Therefore, it is natural (and it was explicitly stated in the objectives of the project) trying to make coexist different proof assistants in a same formalization effort.

Interoperability among proof assistants is an old (and important) topic in automated reasoning. Without doing here a review of this topic, let us comment on the points we consider original in our proposal. First, it is not a universal technology. We focus on very concrete problems related to mathematics formalization, and which are first-order in nature (that last constraint avoids us many troubles with the different underlying logics in each theorem prover). Second, we present an approach where the need of programming is reduced to a minimum. Both features fit well together: since there is very few programming to do, our strategies can be easily adapted when (slightly) moving the context (and so, it is not only an *ad-hoc* solution to a very particular problem).

The reason why we can relieve the programming tasks (either in C or ML or Haskell languages, or even in the internal languages of each prover) is that we are applying well-known XML utilities. In particular, we have designed an XML-based specification language called XLL, capturing the essentials of the kind of properties to be expressed. Unsurprisingly, it is related to another innovative feature of our contribution: the integration with model-based technologies (and more concretely with the EMF tool suite [4]). When providing UML/OCL models of our formalizations, our initial goal was to give the first steps towards a repository of formalized theories, inspired by already-proven experiences (like the CDO or Teneo projects [4]). Waiting for a deeper development of this idea, up to now Ecore technology is used in our transformation plan to validate, in an intermediate step, the conversion made from Isabelle theories. Being our topic of study theorem provers, it is quite evident that we are worried about the correctness of our mappings. In addition to the Ecore intermediate validation, we are able to go back from XLL documents to Isabelle theories (Step 5 in Diagram 1 in Section 3), automatically proving (in Isabelle!) the behavioral equivalence between that Isabelle theory and the original input one. On the contrary, 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.

In summary, we consider our approach simultaneously simple and robust enough to be applied in the area of mathematical certified computation. This claim is justified because the case study of a diagonalization process of integer matrices is not trivial with respect to both the difficulty and the size of the proofs (ca. 6000 lines, 30 definitions and 220 lemmas).

As for open research lines, there is room for many improvements. Perhaps the most obvious one is to produce evidence of the usefulness of our proposal of moving specifications and "proofs schemes" from Isabelle to ACL2. To this aim, it would be necessary to complete other proofs in ACL2 obtained from Isabelle ones. It could be the case that, to be fruitful, it could be necessary to translate more structure from the proofs (we are thinking of translating the *dependencies graph* among lemmas, in order to give to ACL2 *hints* about how performing a proof).

Another further work would be to explore the capability of our architecture (and in particular of our XLL specification language) to be applied to theorem provers different from Isabelle or ACL2. This path has been already started to be walked in the paper [7], where, in the context of verifying Java programs, XLL has been used to translate statements from Coq to ACL2, allowing us to discharge automatically proof obligations that, otherwise, should be proved with human interaction by using Coq.

References

[1] J. Aransay, J. Divasón, J. Heras, L. Lambán, V. Pascual, A. L. Rubio, J. Rubio. A report on an experiment in porting formal theories from Isabelle/HOL to Ecore and ACL2. Technical Report. ForMath European project. `http://wiki.portal.chalmers.se/cse/pmwiki.php/ForMath/ForMath`, 2013.

[2] G. H. Bradley. Algorithms for Hermite and Smith Normal Matrices and Linear Diophantine Equations. *Mathematics of Computation* **25** (116) (1971), 897–907.

[3] A. Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic* **5** (2) (1940), 56–68.

[4] EMF Project Site. `http://www.eclipse.org/modeling/emf/`

[5] ForMath European project. `http://wiki.portal.chalmers.se/cse/pmwiki.php/ForMath/ForMath`.

[6] F. Haftmann, T. Nipkow. Code Generation via Higher-Order Rewrite Systems. *Lecture Notes in Computer Science* **6009** (2010), 103–117.

[7] J. Heras, G. Mata, A. Romero, J. Rubio, R. Sáenz. Verifying a plaftorm for digital imaging: a multi-tool strategy. `http://arxiv.org/abs/1303.1420`, 2013.

[8] M. Kaufmann, P. Manolios, J S. Moore. Computer-Aided Reasoning: An Approach, Kluwer, 2010.

[9] MDT/OCLinEcore. `http://wiki.eclipse.org/MDT/OCLinEcore`

[10] T. Nipkow, L. Paulson and M. Wenzel. Isabelle/HOL: A proof assistant for Higher-Order Logic. Springer, 2002.

[11] L. Paulson. The foundation of a generic theorem prover, *Journal of Automated Reasoning* **5** (3) (1989), 363–397.

[12] D. Steinberg, F. Budinsky, M. Paternostro, Ed Merks. EMF: Eclipse Modeling Framework. Addison-Wesley, 2009.

[13] G. L. Steele. Common Lisp the Language, 2nd edition. Digital Press, 1990.

[14] J. Warmer, A. Kleppe. The Object Constraint Language: Getting Your Models Ready for MDA, 2nd Edition. Addison-Wesley, 2003.

[15] XML Path Language. W3C Recommendation. `http://www.w3.org/TR/xmlschema-1`.

[16] XProc Tutorial. `http://www.xfront.com/xproc/`.