RAPPORT DE RECHERCHE

L R I

ISABELLE IN CERTIFICATION PROCESSES

NEMOUCHI Y / FELIACHI A / WOLFF B / PROCH C

Unité Mixte de Recherche 8623
CNRS-Université Paris Sud – LRI

12/2015

Rapport de Recherche N° 1583

CNRS – Université de Paris Sud
Centre d'Orsay
LABORATOIRE DE RECHERCHE EN INFORMATIQUE
Bâtiment 650
91405 ORSAY Cedex (France)

Abstract

Interactive theorem proving is a technology of fundamental importance for mathematics and computer-science. It is based on expressive logical foundations and implemented in a highly trustable way. Applications include very large mathematical proofs and semi-automated verifications of complex software systems. The architecture of contemporary interactive provers such as Coq, Isabelle, or the **hol!** family goes back to the influential LCF system from 1979, which has pioneered key principles like *correctness by construction* for primitive inferences and definitions, *free programmability* in userspace via SML, and *toplevel command interaction*.

The Isabelle System developed into one of the top 5 systems for the logically consistent development of formal theories. In particular the instance of the Isabelle system with higher-order logic called Isabelle/**hol!** is therefore a natural choice as a formal methods tool as required by the Common Criteria on the higher assurance levels EAL5 to EAL7.

The purpose of this paper is to give a brief introduction into the system, an overview over the methodology and its tool support, and high-level mandatory guidelines for evaluators of certifications using Isabelle. This paper is intended to be a complement of a similar text by french certification authorities [Jae08].

Isabelle in Certification Processes

Yakoub Nemouchi, Abderrahmane Feliachi, Burkhart Wolff, and Cyril Proch

LRI, Université Paris-Sud, 91893 Orsay Cedex, France
Email: {nemouchi,feliachi,wolff}@lri.fr
cyril.proch@thales-group.com

16th December 2015

Contents

Chapter 1

Introduction

Formal methods describe a set of mathematically based techniques and tools for specification, analysis and verification of computer systems. They are mainly used to describe and to verify, in a logically consistent way, some properties of these systems. The formal specification and verification approaches rely usually on some underlying logic. The logical foundation of theorem provers makes them a very convenient basis of any formal development, where the specification and the verification activities can be gathered in one formal environment.

Interactive theorem proving is a technology of fundamental importance for mathematics and computer-science. It is based on expressive logical foundations and implemented in a highly trustable way. Applications include huge mathematical proofs and semi-automated verifications of complex software systems. The architecture of contemporary interactive provers such as Coq [Wie06, §4], Isabelle [NPW02] or the HOL family [Wie06, §1] goes back to the influential LCF system [MW79] from 1979, which has pioneered key principles like correctness by construction for primitive inferences and definitions, free programmability in userspace via SML, and toplevel command interaction.

The purpose of this paper is to bring together a body of system information that is generally known in the Isabelle community, but largely scattered in system documentations and papers. This includes a brief introduction into the system, a general overview over the methodology and covers certain aspects of the tool support. The paper proceeds as follows: at first In chapter 2 , we provide a guided tour over the Isabelle system, while in chapter 3, we refer to methodological issues of Isabelle/**hol!** leading to recommendations for evaluators. In section 3.2, we give some general information from Common Criteria standard about formal methods, modeling and associated requirements. In section 3.4 we chose two major extensions of Isabelle, one for code-verifications, one for model-based testing, and discuss their advantages and limits in a high-level certification process. The final discussion contains a

little survey on publications on the topic as well as a summary for evaluators.

Chapter 2

Isabelle: Its System Architecture and Methodology

2.1 Isabelle/HOL

In the following, we will discuss the two questions:

How is Isabelle built?

How should Isabelle be used?

In the context of certifications of critical hard- and software systems, an understanding of its architecture and the underlying methodology may help to understand why Isabelle, if correctly used, can be trusted to a significantly higher extent than conventional software, even more than other automated theorem provers (in fact, Sascha Böhme's work on proof reconstruction [BW10] inside Isabelle revealed errors the SMT solver Z3[dMB08] that is perhaps the most tested conventional system currently on the market ...). Of course, Isabelle as software "contains errors". However, its architecture is designed to exclude that errors allow to infer logically false statements, and methodology may help to exclude that correctly inferred logical statements are just logical artifacts, or logically trivial statements, which can be impressing stunts without any value.

2.2 The Isabelle System Architecture

We will describe the layers of the system architecture bottom-up one by one, following the diagram Figure 2.1.

The foundation of system architecture is still the Standard ML (SML,[MTM97]) programming environment; the default PolyML implementation

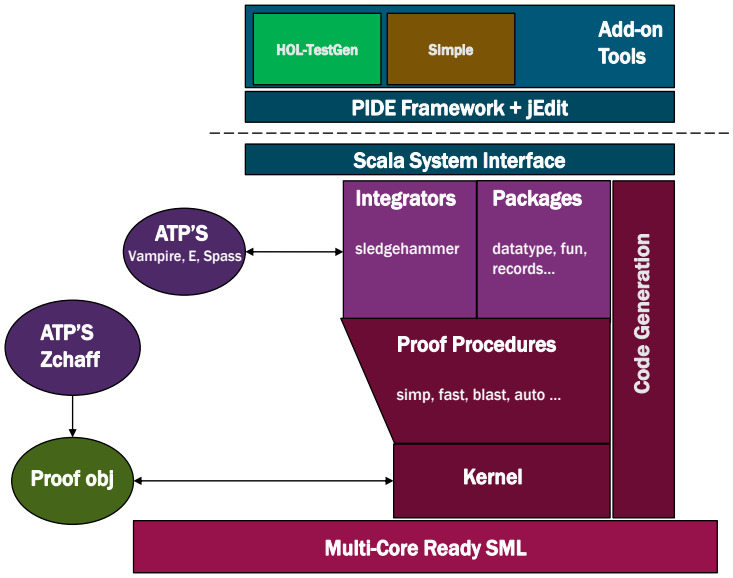

Figure 2.1: The diagram shows the different layers like execution environment, kernel, tactical level and proof-procedures, component level (providing external prover integration like Z3, specification components, and facilities like the code generator, the Scala API to the system bridging to the JVM-World, and the Prover-IDE (PIDE) layer allowing for asynchronous proof and document checking.

www.polyml.org supports nowadays multi-core hardware which is heavily used in recent versions for parallel and asynchronous proof checking when editing Isabelle theories.

On top of this, the logical kernel is implemented which comprises type-checking, term-implementations and the management of global contexts (keeping, among many other things, signature information and basic logical axioms). The kernel provides the abstract data-types `thm`, which is essentially the triple (Γ, Θ, ϕ), written $\Gamma \vdash_\Theta \phi$, where Γ is a list of *meta-level assumptions*, Θ the *global context*, containing, for example, the signature and core axioms of `HOL` and the signature of group operators, and a *conclusion* ϕ, i. e.a formula that is established to be derivable in this context (Γ, Θ). Intuitively, a `thm` of the form $\Gamma \vdash_\Theta \phi$ is stating that the kernel certifies that ϕ has been derived in context Θ from the assumptions Γ.

There are only a few operations in the kernel that can establish `thm`'s, and

the system correctness depends *only* on this trusted kernel. On demand, these operations can also log proof-objects that can be checked, in principle, independently from Isabelle; in contrast to systems like Coq, proof objects do play a less central role for proof checking which just resides on the inductive construction of `thm`'s by kernel inferences shown, for example, in [PP10].

On the next layer, proof procedures were implemented - advanced tactical procedures that search for proofs based on higher-order rewriting like `simp`, tableau provers such as `fast`, `blast`, or `metis` , and combined procedures such as `auto`. Constructed proofs were always checked by the inference kernel.

The next layer provides major components — traditionally called *packages* — that implement the *specification constructs* such as *type abbreviations*, *type definitions*, etc., as discussed in section 2.4 in more details. Packages may also yield connectors to external provers (be it via the `sledgehammer` interface or via the `smt` interface to solvers such as Z3), machinery for (semi-trusted) code-generators as well as the Isar-engine that supports structured-declarative and imperative "apply style" *proofs* described in section 2.7.

The Isar - engine [Wen02] parses specification constructs and proofs and dispatches their treatment via the corresponding packages. Note that the Isar-Parser is configurable; therefore, the syntax for, say, a data-type statement and its translation into a sequence of logically safe constant definitions (constituting a "model" of the data type) can be modified and adapted, as well as the automated proofs that derive from them the characterizing properties of a data-type (distinctness and injectivity of the constructors, as well as induction principles) as `thm`'s available in the global context Θ thereafter. Specification constructs represent the heart of the methodology behind Isabelle: new specification elements were only introduced by "conservative" mechanisms, i. e. mechanisms that maintain the logical consistency of the theory by construction; internally these constructs introduce declarations and axioms of a particular form. Note that some of these specification constructions, for example type definitions, require proofs of methodological side-conditions (like the non-emptyness of the carrier set defining a new type).

We mention the last layer mostly for completeness: Recent Isabelle versions posses also an API written in Scala, which gives a general system interface in the JVM world and allows to hook-up Isabelle with other JVM-based tools or front-ends like the jEdit client. This API, called the "Prover IDE" or "PIDE" framework, provides an own infrastructure for controlling the concurrent tasks of proof checking. The jEdit-client of this framework is meanwhile customized as default editor of formal Isabelle *sessions*, i. e. the default user-interface the user has primarily access to. PIDE and its jEdit client manage collections of theory documents containing sequences of specification constructs, proofs, but also structured text, code, and machine-checked results of code-executions. It is natural to provide such theory documents as part

of a certification evaluation documentation.

2.3 Isabelle and its Meta-Logic

The Isabelle kernel natively supports minimal higher-order logic called *Pure*. It supports for just one logical type prop the meta-logical primitives for implication $\_ \Longrightarrow \_$ and universal quantification $\bigwedge x.\ P\ x$. The meta-logical primitives can be seen as the constructors of *rules* for various logical systems that can be represented inside Isabelle; a conventional "rule" in a logical textbook:

$$\frac{A_1 \cdots A_m}{C} \tag{2.1}$$

can be directly represented via the built-in quantifiers $\bigwedge$ and the built-in implication $\Longrightarrow$ as follows in the Isabelle core logic Pure:

$$\bigwedge x_1 \ldots x_n .\ \ A_1 \Longrightarrow \ldots \Longrightarrow A_m \Longrightarrow \mathrm{C} \tag{2.2}$$

... where the variables x_1, ..., x_n are called *parameters*, the premises A_1, ..., A_m *assumptions* and C the conclusion; note that $\Longrightarrow$ binds to the right. Also more complex forms of rules as occurring in natural deduction style inference systems like:

$$\frac{\begin{array}{c}[A]\\ \vdots \\ B\end{array}}{A \to B} \tag{2.3}$$

can be represented by $(A \Longrightarrow B) \Longrightarrow A \to B$. Thus, the built-in logic provided by the Isabelle Kernel is essentially a language to describe (systems of) logical rules and provides primitives to instantiate, combine, and simplify them. Thus, Isabelle is a *generic* theorem prover. New object logics can be introduced by specifying their syntax and natural deduction inference rules. Among other logics, Isabelle supports first-order logic, Zermelo-Fraenkel set theory and the instance for Church's higher-order logic HOL. Moreover, Isabelle is also a generic system framework (roughly comparable with Eclipse) which offers editing, modeling, code-generation, document generation and of course theorem proving facilities; to the extent that some users use it just as programming environment for **sml!** or to write papers over checked mathematical content to generate LaTeX output. Many users know only the theorem proving language **isar!** for structured proofs and are more or less unaware that this is a particular configuration of the system, that can be easily extended. Note that for all of the aforementioned specification constructs and proofs there are specific syntactic representations in **isar!**.

Higher-order logic (HOL) [Chu40, And86, And02] is a classical logic based on a simple type system. It is represented as an instance in Pure. HOL provides the usual logical connectives like $\_ \wedge \_$, $\_ \rightarrow \_$, $\neg \_$ as well as the object-logical quantifiers $\forall x.\ P\,x$ and $\exists x.\ P\,x$; in contrast to first-order logic, quantifiers my range over arbitrary types, including total functions $f :: \alpha \Rightarrow \beta$. HOL is centred around extensional equality $\_ = \_ :: \alpha \Rightarrow \alpha \Rightarrow$ bool. HOL is more expressive than first-order logic, since, e. g., induction schemes can be expressed inside the logic. Being based on a polymorphically typed λ-calculus, **hol!** can be viewed as a combination of a programming language like **sml!** or Haskel, and a specification language providing powerful logical quantifiers ranging over elementary and function types.

Isabelle/HOL is the session based on the embedding of HOL into Isabelle/Pure. Note The that simple-type system as conceived by Church for HOL has been extended by Hindley/Milner style polymorphism with type-classes similar to Haskel[WB89, Wen97].

2.4 The Isabelle Methodology

The core of the logic is done via an axiomatization of the core concepts like equality, implication, and the existence of an infinite set, the rest of the library is derived from this core by logically safe ("conservative") extension principles which are syntactically identifiable constructions in Isabelle files. In the following, we will briefly describe the axiomatic foundation of Isabelle/HOL and describe the most common conservative extension principles.

2.4.1 The Logical Core of HOL.

In the entire library (so the Isabelle session "HOL" which is also referred to as "Main" in theory imports), there are only 11 axioms in form of foundational axioms of the HOL-logic:

1. The equality symbol is axiomatized as an equality, i. e. it is reflexive, extensional, and satisfies the Leibniz-property (equals can be replaced by equals in any context P). The Hilbert-Operator is bound to choose the value characterized by equality:

```
axiomatization
where refl        : t = (t::α) and
      subst       : s = t ⟹P s ⟹P t and
      ext         : (⋀ x::α. (f x ::β) = g x) ⟹
                      (λx. f x) = (λx. g x) and
      the_eq_trivial: (THE x. x = a) = (a::'a)
```

2. The following axioms establish a relation between implication and rule formation, and between implication and equality, as well as `True`, `∀ x. P x` and `False` and (which are abbreviations for `((λx::bool. x)= (λx. x))`, `(P = (λx. True))` and `(∀ P. P)`, respectively):

```
axiomatization
where impI        : (P ⟹Q) ⟹ P → Q and
      mp          : P →Q ⟹ P  ⟹Q and
      iff         : (P→Q) → (Q→P) → (P=Q) and
      True_or_False: (P=True) ∨(P=False)
```

3. Finally, a type `ind` is postulated to have an interpretation by an infinite carrier set. Instead of the more common form to state the axiom of infinity: `∃f::ind⇒ind. injective(f)∧¬surjective(f)` , this axiom comes in two parts over two constants `Zero_Rep` and `Suc_Rep`:

```
   axiomatization Zero_Rep :: ind and Suc_Rep :: ind ⇒ind where
     Suc_Rep_inject:     Suc_Rep x = Suc_Rep y ⟹x = y and
     Suc_Rep_not_Zero_Rep: Suc_Rep x ≠ Zero_Rep
```

   On this basis, the type of natural numbers is constructed via an inductive definition, the integer and rational numbers via quotient constructions, etcpp.

4. A further axiom is devoted for another form of the Hilbert-Choice operator:

```
   axiomatization Eps :: ('a ⇒bool) ⇒'a
   where someI: P x ⟹ P (Eps P)
```

An Isabelle/HOL version coming from a trusted distribution site should *only* have these axioms. Note that in the "src/HOL" folder containing the system libraries, there are many example theories and sub-sessions that actually state their own axioms; a prudent Isabelle theory evaluator should make sure that none of these sessions were included.

2.5 Specification Constructs: Isabelle Conservative Extensions

Besides the logic, the instance of Isabelle called Isabelle/HOL offers support for specification constructs mapped to conservative extensions schemes, i. e. a combination of type and constant declarations as well as (internal) axioms of a very particular form. We will briefly describe here *type abbreviations*, *type definitions*, *constant definitions*, *datatype definitions*, *primitive recursive definitions*, *well-founded recursive definitions* as well-as Locale constructions.

We consider this as the "methodologically safe" core of the Isabelle/HOL system.

Using solely these conservative definition principles, the entire Isabelle/HOL library is built which provides a *logically safe language base* providing a large collection of theories like sets, lists, Cartesian products $\alpha \times \beta$ and disjoint type sums $\alpha + \beta$, multi-sets, orderings, and various arithmetic theories which only contain rules derived from conservative definitions.

2.5.1 Type Abbreviations (Synonyms).

For example, typed sets are built in the Isabelle libraries via type synonyms on top of **hol!** as functions to bool; consequently, the constant definitions for set comprehension and membership are as follows[1]:

```
type_synonym 'α set = 'α ⇒bool

definition Collect :: ('α ⇒bool) ⇒'α set
where    Collect S = S

definition member:: 'α ⇒'α set ⇒bool
where    member s S = S s
```

Isabelle's powerful syntax engine is instructed to accept the notation $\{x \mid P\}$ for Collect $\lambda x.\ P$ and the notation $s \in S$ for member $s\ S$. As can be inferred from the example, constant definitions are axioms that introduce a fresh constant symbol by some closed, non-recursive expressions; these types of axioms are logically safe since they work like an abbreviation. The syntactic side-conditions of the axioms are mechanically checked, of course. It is straightforward to express the usual operations on sets like $\_ \cup \_$, $\_ \cap \_ :: \alpha\,\text{set} \Rightarrow \alpha\,\text{set} \Rightarrow \alpha\,\text{set}$ as definitions, too, while the rules of typed set-theory are derived by proofs from them.

2.5.2 Datatypes.

Similarly, a logical compiler is invoked for the following statements introducing the types option and list:

```
datatype 'a list = Nil | Cons 'a 'a list
datatype 'a option = None | Some 'a
```

Here, [] and $a\#l$ are alternative syntax for Nil and Cons $a\ l$; moreover, $[a, b, c]$ is defined as alternative syntax for $a\#b\#c\#[]$. Similarly, the option type shown above is given a different notation: αoption is written as $\alpha_\perp$, None as $\perp$, and Some X as $\lfloor X \rfloor$. Internally, recursive datatype definitions

[1] To increase readability, the presentation is slightly simplified.

are represented by type- and constant definitions. Besides the *constructors* None, Some, Nil and Cons, the statement above defines implicitly the match-operation case x of $\bot \Rightarrow F \mid \lfloor a \rfloor \Rightarrow G\ a$ respectively case x of $[] \Rightarrow F \mid (a\#r) \Rightarrow G\ a\ r$. From the internal definitions (not shown here) many properties are automatically derived like distinctness $[] \neq a\#t$, injectivity of the constructors or induction schemes.

2.5.3 Well-founded Recursive Function Definitions.

Actually, there is a parser for primitive and well-founded recursive function definition syntax. For example, the sort-operation can be defined by:

```
fun ins ::
    'α:: linorder ⇒'α List.list ⇒'α List.list
where
  ins x [] = [x]
| ins x (y#ys) = (if x < y then x #y#(ins x ys) else y#(ins x ys))

fun sort ::'α:: linorder List.list ⇒'α List.list
where
  sort  [] = []
| sort (x#xs) = ins x (sort xs)
```

which is again compiled internally to constant and type definitions. Here, $\alpha :: \text{linorder}$ requires that the type α is a member of the *type class* linorder. Thus, the operation sort works on arbitrary lists of type $(\alpha :: \text{linorder})$ list on which a linear ordering is defined. The internal (non-recursive) constant definition for the operations ins and sort is quite involved and requires a termination proof with respect to a well-founded ordering constructed by a heuristic. Nevertheless, the logical compiler will finally derive all the equations in the statements above from this definition and makes them available for automated simplification.

The theory of partial functions is of particular practical importance. Partial functions $\alpha \rightharpoonup \beta$ are then defined as functions $\alpha \Rightarrow \beta\,\text{option}$ supporting the usual concepts of domain $\text{dom}\ f \equiv \{x \mid f\ x \neq \text{None}\}$) and range $\text{ran}\ f \equiv \{x \mid \exists y.\ f\ y = \text{Some}\ x\}$. Partial functions can be viewed as "maps" or dictionaries; the *empty* map is defined by $\emptyset \equiv \lambda x.$ None, and the update operation, written $p(x \mapsto t)$, by $\lambda y.\,\text{if}\ y = x\,\text{then Some}\ t\,\text{else}\,p\ y$. Finally, the override operation on maps, written $p_1 \oplus p_2$, is defined by $\lambda x.\,\text{case}\ \ p_1\ x\ \ \text{of None} \Rightarrow p_2 x \mid \text{Some}\ X \Rightarrow \text{Some}\ X$.

2.5.4 Type definitions.

Type definitions allows for a safe introduction of a new type. Other specification constructs, for example datatype, are based on it. The underlying

construction is simple: any non-empty subset of an existing type can be turned into new type. This is achieved by defining an isomorphism between this set and the new type; the latter is introduced by two fresh constant symbols (representing the abstraction and the concretization function) and three internally generated axioms. As a simple example, consider the definition of type containing three elements. This type is represented by the first three natural numbers:

```
typedef three = {0::nat,1,2}
  apply (rule_tac x= 0 in exI)
  apply blast
  done
```

In order to enforce that the representing set on the right hand side is non empty, the package requires for this new type a proof of non-emptyness:

```
typedef three = {0::nat,1,2}
 1. ∃x. x ∈{0, 1, 2}
```

To use this new type we need to finish the proof of non empty set started by the use of `typedef` which can be done differently. For example we can finish the proof using existing theorems on the logical operator ∃ in Isabelle/HOL. To see all Isabelle's theorems related to ∃ we use the Isabelle command `find_theorems`. The query searches for theorems whose name contains an "ex" substring. One of the results is:

```
find_theorems name : exI
HOL.exI: ?P ?x ⟹∃x. ?P x
```

The searched theorems is applied in the following. In our case, the Isabelle proof method *rule_tac* is used, a resolution step, which unifies the theorem HOL.exI against the first proof goal in a resolution step:

```
apply (rule_tac x= 0 in exI)
apply blast
done
```

Its application in the proof allows to replace the schematic variable $?x$ by the constant 0 in our proof; this is specified by the key word **in** followed by the name of the theorem. The other schematic variable $?P$ is automatically filled in (using higher-order unification), which is possible since only one solution remains. The remainder of the proof consists of a call to the highly automated method `blast`, which does the trick for the necessary set-theoretic proof.

It remains to point out that the same proof can be done by different proof-style called *structued proof* or *Isar-proof*. The same proof can be represented in this style as follows:

```
typedef three = {0::nat,1,2}
  proof
  show 1 ∈{0, 1, 2}
    by blast
  qed
```

After finishing the proof about the definition of this new type, many theorems will be deduced automatically by Isabelle. We can check the new deduced theorems related to this new type by using the command `find_theorems`. In the concrete example, there are 82 new theorems deduced that were related to this type definition.

```
find_theorems name : three
searched for name: three
found 82 theorems (40 displayed)
```

2.5.5 Inductively defined predicates.

This section is dedicated to the most important definition principle after recursive functions and datatypes: inductively defined predicates. We will introduce a known example in Isablle manuals: the set of even (natural) numbers. For more complicated examples see [NPW02]. The specification construct allows for building the *least* set which is closed under a given collection of introduction rules; in our case: one rule that states 0 is an even number, and the other one rule that states that if we add 2 to every even number we will get an even number. Using the keyword `inductive`, we declare the constant `even` to be a predicate that allows as to get the set of natural numbers with desired properties. (Note that sets and boolean functions are treated the same.)

```
inductive even :: nat ⇒bool
where
  zero[intro!]: even 0
 |step[intro!]: even n ⟹even (Suc(Suc n))
```

Note that the declaration of the rules comes, as usual in many places in the Isar-language, with an instrumentation: for both rules, the names `zero` and `step` were introduced, and with a number of *attributes* it can be stated *how* the given rule or `thm` should be *used* in proofs: the keyword `[intro!]` indicates that they should be used as introduction rules in proof search. After the inductive statement, Isabelle generates a fixed point definition for `even` and proves theorems about it. These theorems include the introduction rules specified in the declaration, an elimination rule for case analysis and an induction rule for the global judgement. To inspect these theorems we can again use `find_theorems` which results in:

```
find_theorems name:even
Three.even.cases: even ?a⟹ (?a = 0 ⟹?a)⟹
                (⋀ n. ?a = Suc(Suc n)⟹ even n⟹ ?P)⟹ ?P
Three.even.induct: even ?x⟹ ?P 0⟹
                 (⋀ n. even n⟹ ?P n⟹ ?P (Suc(Suc n)))⟹ ?P ?x
Three.even.zero: even 0
Three.even.step: even ?n ⟹even (Suc(Suc ?n))
```

We can refer to these theorems by automatically-generated names, for example: `Three.even.cases`, `Three.even.induct` ...

2.5.6 Type classes.

We will introduce another important concept. Type-classes can be seen as a simple modularization concept (similar locales, but with less expressive power), which is particularly well integrated into the type system. Similar to Haskel, type classes κ restrict type variables to belong to a particular class of types having common properties. Following a popular example on using type classes, the introduction of a new class `plus` and its operation $\oplus$ is done by this Isabelle/Isar fragment:

```
class plus =
fixes plus ::'a ⇒'a ⇒'a (infixl ⊕70)
instantiation nat :: plus
begin
  print_context
  primrec plus_nat::nat⇒ nat ⇒nat
  where
  (0::nat )⊕ n = n
  | Suc m ⊕n = Suc (m ⊕n)
  instance proof qed
end
```

The type of the operation $\oplus$ carries a class constraint `'a::plus` on its type variable, meaning that only types of class `plus` can be instantiated for `'a`. To locally instantiate a type-class by an other existing type we use the command `instantiation`. For example to instantiate `plus` on `nat` we write the key word `instantiation` and the name of existing type that we want to instantiate, and the type of the operation which is in our case `nat ⇒ nat ⇒ nat`. Now we define the $\oplus$ function and give a semantic to to it on type `nat`. Note that all function names are written by the combination of the name of the class operation and the name of the instance which the class operation will be applied on (example`plus_nat`). In case of uncertainty, these names may be inspected using the command`printcontext` as follow:

```
instantiation nat :: plus
begin
```

```
    print_context
    ...
    Isabelle output:
    nat::Three.plus
    plus_nat ≡Three.plus_classe.plus :: nat ⇒nat ⇒nat
```

In general, assumptions were assumed in the context of the instantiations, proofs for those assumptions are mandatory in instantiations. Such proofs are done using the command `instantiation` before the end of the context of the instantiations. In our example, the proof is a standard phrase necessary for technical reasons. We can also add many instantiations for the operations. For example the operation $\oplus$ of the class `plus` can be applied on the type of products:

```
instantiation prod :: (plus, plus)plus
begin
fun plus_prod :: 'a*'b ⇒'a*'b ⇒'a * 'b
where (x,y) ⊕(w,z) = (x ⊕w, y ⊕z)
instance proof qed
end
```

Now, in a term $(3, 4, 5) \oplus (1, 2, 3)$, the type inference will infer that there is actually a series of instantiations that define this product on triples ... More in depth explanations for type classes are in [NPW02].

While type-classes have a strictly weaker expressive power that Isabelle's Locales to be discussed in the sequel, they have the advantage that the types can be inferred completely automatic; their annotation can therefore omitted in most cases. Furthermore, the lack of dependent types (a concept existing in Coq) can in some practical cases by compensated by type-classes; it is, for example, perfectly possible to define the bit vector type "32 word" and "64 word" inside a word-library providing types "$'\alpha$ word" (here, "32" is a syntactic synonym for a type-class of types that are representable by 32 bits). Thus, type-classes can establish dependencies of types from values which is impossible in a standard Hindley-Milner type-system.

Locales.[Bal10]

Locales are Isabelle's approach for dealing with parametric theories. They have been designed as a module system that can adequately represent the complex inter-dependencies between structures found in abstract algebra, but have proven fruitful also in other applications. We will briefly discuss major features of locales.

As a prerequisite, recall that the general format of a *logical rule* represented

in Isabelle/Pure is:

$$\frac{A_1 \cdots A_m}{C} \quad \text{where } x_1 \ \ldots \ x_n \text{ are free variables}$$

On the level of the Isar-language, a rule of this form can equivalently represented as:

```
fixes x1 ... xn
assumes A1
and ...
and Am
shows C
```

Parameters and assumptions together form a *local context*. A formula C is a *theorem in a local context* if it is a *conclusion*. A *locale* is just local context that have been made persistent. As a particular feature, they allow for introducing local syntax for the x_i and individual prover instrumentation for the assumptions. To the user, however, they provide powerful means for declaration, combination, and for reuse of theorems proved in them. The following example [Bal10], is the formalization of partial order with locale `partial_order`.

```
locale partial_order=
fixes less_equal :: nat ⇒nat ⇒bool (infixl ≲50)
assumes refl [intro,simp]: x ≲x
and anti_sym [intro] : ⟦x ≲y ; y ≲x ⟧⟹x = y
and trans [trans] : ⟦x ≲y ; y ≲z ⟧⟹x ≲z
```

In this locale the parameter is `less_equal`, which is binary predicate with infix syntax $\lesssim$. The parameter syntax is available in the subsequent assumptions, which correspond to the familiar partial order "axioms". Isabelle recognizes unbound names as free variables. In locale assumptions, the are implicitly universally quantified. That is, `x ≲y ⟹ y ≲z ⟹ x ≲z` in fact means `⋀x y z. x≲y ⟹ y ≲z ⟹ x ≲z`. There are two Isar commands to inspect a locale: *printlocale* lists the names of all locales of current theory; *printlocale* α prints the parameters and assumptions of locale α; the variation *printlocale*!α additionally outputs the conclusions that are stored in the locale. For the Isar command:

```
printlocale partial_order
```

the system produces the output:

```
locale partial_order
fixes less_equal :: nat ⇒nat ⇒bool
assumes partial_order ?P ?≲}
```

Analogously, for:

```
printlocale! partial_order
```

the output

```
locale partial_order
fixes less_equal:: nat ⇒nat ⇒bool
assumes partial_order ?P ?≲
notes partial_order_axioms =
(partial_order ?P ?≲) [attribute <attribute>]
notes refl    = (?x ?≲ ?x) [HOL.intro,\ap simp]
and   anti_sym = (?x ?≲ ?y ⟹?y ?≲ ?x ⟹?x = ?x) [HOL.intro]
and   trans   = (?x ?≲ ?y ⟹?y ?≲ ?z ⟹?x ?≲ ?z) [trans]
```

is produced. Here, the keyword `notes` denotes a conclusion element. There is two conclusions, which were added automatically. Instead there is only one assumption, namely `partial_order} (op≲)`. The locale declaration has introduced the predicate `partial_order` to the theory. This predicate is called the *locale predicate*. Its definition may be inspected by the command:

```
partial_order_def
```

corresponding to the output:

```
partial_order ?less_equal \ap \equiv
(∀ x. ?less_equal x x) ∧
(∀ x y. ?less_equal x y ⟹?less_equal y x ⟹x = y)
(∀ x y z. ?less_equal x y ⟹?less_equal y z ⟹?less_equal x z)
```

Each conclusion has *foundational theorem* as counterpart in the theory. Technically, this is simply the theorem composed of local context and conclusion. For the transitivity, for example, we have the output:

```
partial_order ?less_equal ⟹?less_equal x y ⟹
?less_equal y z ⟹?less_equal x z
```

The specification of a locale is fixed, but its list of conclusions may be extended through Isabelle commands that take a *target* argument. In the following, two examples on two Isabelle commands that accept a target. The first example on the command `definition` and the second example is on the command `lemma`.

```
definition (in partial_order)
strict_less :: nat ⇒nat ⇒bool
where x ≳ y = x ≲ y  ∧  x ≠ y
```

The strict order `strict_less` with infix syntax $\gtrsim$ is defined in terms of the locale parameter `less_equal` and the general equality of the object logic we work in. The definition generates a constant `partial_order.strict_less` with definition `partial_order.less_def`:

```
1       partial_order ?less_equal ⟹
2       partial_order.less ?less_equal x y = (?less_equal x y ∧x ≠ y)
```

The context of a locale can be extended by a block of commands, delimited by `beginn` and `end`, like when we start a new theory. The main restriction when we use a block of commands, is that the block refer to the same target (the same locale). If the block of commands follows a locale declaration, that makes this locale the target. In other cases, the target for a block may be given withe `context` command. In the example below, we will introduce two new definitions for the locale `partial_order`, in those new definitions we will introduce the notion of infimum ad supremum for partial orders.

```
1    context partial_order
2     begin
3       definition is_inf
4       where is_inf x y i = (i ≲x ∧i ≲y ∧(∀ z. z ≲x ∧z ≲y ⟶z ≲ i))
5       definition is_sup
6       where is_sup x y s = (x ≲s ∧y ≲s ∧(∀ z. x ≲z ∧y ≲z ⟶s ≲z))
7    end
```

2.5.7 ML Code.

It is possible inside Isabelle documents to directly access the underlying ML-layer of the system architecture, and even extend the environment of the underlying ML interpreter/compiler. One can include the fragment:

```
1       ML{* fun fac x = if x = 0 then 1 else x * fac(x-1); *}
```

in a document and then later on evaluate:

```
1       ML{* fac 20; *}
```

Since Isabelle itself sits as a collection of ML modules in this SML environment, it is possible to access its kernel and tactical functions:

```
1       ML{* open Tactic;
2            fun mis x = res_inst_tac [(x, x)] {@thm exI} 1*}
```

which defines a new tactic that applies just the existential-introduction rule of **hol!**. This is the key to build large and own tactic procedures and even tools inside the Isabelle environment. Note that the fragment `{@thm exI}` is called an *antiquotation*; it is expanded before being passed to the SML compiler with code that accesses the `thm exI` (see section section 2.4, pp8.) in the Isabelle database for theorems. By additional SML-code, this tactic can be converted into a Isar-*method*, which can be bound to own syntax inside the Isar-language. Thus, the proof language is technically extensible by own, user-defined proof-commands (see [Wen15] for the details).

2.6 Isabelle libraries

Isabelle libraries are predefined theories for users. New theories can be defined using Isabelle specification contructs (i. e. constant definitions) and reasoning around those new definitions can be established using Isabelle lemmas. In general, the predefined libraries implement a known theories like: set theory [NPW14], a theory on natural numbers [NP00], lists [Nip13], functions ... We have to notice that the cited theories are included in HOL[NWP13], which is an Isabelle instantiation for higher order logic. In this section we will focus on the theories used in the specification of our test theory and the diferent case studies that we will introduce to the reader.

2.6.1 Records

An Isabelle record [Wen15, NWS+] is a data structure that contain a number of fields. In its essence a record a representation the algebraic structure of tuples. Inside records theory, and using Isabelle specification contructs a new tools are defined (i. e.records selectors, records update function, more field, records refinement scheme ...). The Isar keyword used to declare records in Isabelle is *record*.

```
record ('a, 'b) state =
  field1   :: 'a
  field2   :: 'b
```

In this example we had declared an Isabelle record type named *state*, that contain two fields *field1, field2* and supports two types *'a, 'b*. After the definition of this record type a set of Isabelle theorems are generated automatically.

```
Record1.state.select_defs(1):
  field1 ≡
     id ∘ Record.iso_tuple_fst Record.tuple_iso_tuple ∘
     Record.iso_tuple_fst state_ext_tuple_Iso
```

This theorem is used to retrieve field named `field1` from the record `state`. Another theorem used to update the same field is generated automatically:

```
Record1.state.update_defs(1):
 field1_update ≡
     Record.iso_tuple_fst_update state_ext_tuple_Iso ∘
    (Record.iso_tuple_fst_update Record.tuple_iso_tuple ∘id)
```

Moreover, a predefined records library [NWS+] containing a generic theorems, lemmas and operations that we can apply on record types. Records are isomorphic to compound tuple types. In Isabelle, to implement records, an explicite theory on isomorphism was defined as library for records. An example on operations and lemmas contained by the theory is as following:

```
datatype ('a, 'b, 'c) tuple_isomorphism =
  Tuple_Isomorphism 'a ⇒'b ×'c 'b ×'c ⇒'a

primrec
  repr :: ('a, 'b, 'c) tuple_isomorphism ⇒'a ⇒'b ×'c where
  repr (Tuple_Isomorphism r a) = r

primrec
  abst :: ('a, 'b, 'c) tuple_isomorphism ⇒'b ×'c ⇒'a where
  abst (Tuple_Isomorphism r a) = a

definition
  iso_tuple_fst_update ::
    ('a, 'b, 'c) tuple_isomorphism ⇒('b ⇒'b) ⇒
      ('a ⇒ 'a)
where iso_tuple_fst_update isom f = abst isom ∘apfst f ∘repr isom

definition
  iso_tuple_snd_update ::
    ('a, 'b, 'c) tuple_isomorphism ⇒('c ⇒'c) ⇒
      ('a ⇒ 'a)
where iso_tuple_snd_update isom f = abst isom ∘apsnd f ∘repr isom

lemma update_accessor_congruence_foldE:
  assumes uac: iso_tuple_update_accessor_cong_assist upd ac
    and r: r = r' and v: ac r' = v'
    and f: ⋀v. v' = v ⟹f v = f' v
  shows upd f r = upd f' r'
  using uac r v [symmetric]
  apply (subgoal_tac upd (λx. f (ac r')) r' = upd (λx. f' (ac r')) r')
   apply (simp add: iso_tuple_update_accessor_cong_assist_def)
  apply (simp add: f)
  done
```

Other operations on records like extending record type are defined too.

2.6.2 Functions

The HOL instantiation for Isabelle contains a theory on total functions [Nip12]. A set of operations and lemmas are defined in this theory. An Isabelle function is seen as an application f: $E \to F$, where E is the domain and F is the range of f, in the following some Isabelle definition that exist in this theory are presented:

```
definition id :: 'a ⇒'a where
  id = (λx. x)
definition comp :: ('b ⇒'c) ⇒('a ⇒'b) ⇒
                   'a ⇒ 'c (infixl o 55)
```

```
5     where f o g = (λx. f (g x))
6
7     lemma id_apply [simp]: id x = x
8       by (simp add: id_def)
9
10    lemma comp_apply [simp]: (f o g) x = f (g x)
11      by (simp add: comp_def)
```

In those two examples *id* specify the identity function and *comp* (has as syntax the symbol *o*) specify function composition. Actually, the theory *Fun.thy* is an extension of the *Set.thy* theory, other definitions like domain of the function, range, image ... are implemented in *Set.thy*.

2.7 Isabelle Proofs

In addition to types, classes and constants definitions, Isabelle theories can be extended by proving new lemmas and theorems. These lemmas and theorems are derived from other existing theorems in the context of the current theory. Isabelle offers various ways to construct proofs for new theorems, we distinguish two main categories: forward and backward proofs. In addition to Isabelle proofs, some external proofs can be integrated – in a logically safe way – and compiled into an Isabelle proof.

2.7.1 Local forward proofs.

The goal of a forward proof is to derive a new theorem from old ones. This is done either by instantiating some unknowns in the old theorems, or by composing different theorems together.

The instantiation can be done using the `of` and `where` operators as follows: `thm[of inst1 inst2 ...]` or `thm[where var1=inst1 and var2=inst2 ...]`. If we consider for example the existential introduction theorem called `exI` and given by `?P ?x ⟹∃ x. ?P x`. The unknown variable `x` can be instantiated with a fixed variable `a` using the following command `exI[of _ a]` which is equivalent to `exI[where x=a]`. Note that when using `of` the instances of the variables appear in the same order of appearance of the unknown variables in the theorems. Consequently, we can avoid instantiating a variable by giving a dummy value in the position of its corresponding instance.

The second way of deriving theorems is by composing different theorems together using the `OF` or `THEN` operators. The first operator `OF` is used to compose one theorem to others. For a theorem `th1` given by `A ⟹ B` and a theorem `th2`given by `A'`, the theorem `th1[OF th2]` results from the unification of `A` and `A'` and thus instantiating the unknowns in `B`. Theorems with multiple premises can be composed to more than one theorem given as arguments to the `OF` operator. For example, given the conjunction introduction

theorem `conjI` given by `?P ⟹?Q ⟹?P ∧ ?Q` and the reflexivity theorem `ref` given by `?x = ?x`, the composition of these theorem `conjI[OF refl[of a] refl[of b]]` results in the following theorem `a = a ∧ b = b`. In a similar way, the `THEN` operator is used to compose different theorems together. The theorem `th1[THEN th2]` is obtained by applying the rule `th2` to the theorem `th1`. For example, composing a theorem `th1` given by `a = b` with the symmetry rule `sym` given by `?s = ?t ⟹?t = ?s` is written `th1[THEN sym]` and the result is `b = a`.

2.7.2 Global backward proofs.

The usual and mostly used proof style is the backward or goal-directed proof style. First, a proof goal is introduced then the proof is performed by simplifying this goal into different subgoals and, finally, prove the resulting subgoals from existing theorems. The proofs are build using natural deduction by applying some existing (proved) inference rules. For each logical operator, two kinds of rules are defined: introduction and elimination rules.

The backward proofs can be structured in two different ways:

1. Apply style proofs, where the proof goal is simplified using a succession of rules applications. This results in a so-called apply-script, describing the proof steps. An example of such a proof is given in the following:

```
lemma conj_rule: ⟦P; Q ⟧⟹P∧ (Q ∧P)
  apply (rule conjI)
    apply assumption
  apply (rule conjI)
    apply assumption
    apply assumption
  done
```

   Although this proof style is easy to apply, long apply-scripts can become unreadable and hard to maintain. A more structured and safe way to write the proofs is by using the Isar language.

2. Structured Isar proofs allow for writing sophisticated and yet still fairly human-readable proofs. The Isar language defines a set of commands and shortcuts that offer more control on the proof state. An example of a structured induction proof is given in the following:

```
lemma
  fixes n::nat
  shows 2 * (∑ i=0..n. i) = n * (n + 1)
  Proof (induct n)
    case 0
      have 2 * (∑ i=0..n. i) = (0::nat)
        by simp
```

```
8           also have (0::nat) = 0 * (0 + 1)
9             by simp
10          finally show ?case .
11        next
12         case (Suc n)
13          have 2 * (∑ i=0..Suc n. i) = 2*(∑ i=0..n. i) + 2 *(n + 1)
14            by simp
15          also have 2*(∑ i=0..n. i) = n * (n + 1)
16            by (rule Suc.hyps)
17          also have n * (n + 1) + 2 * (n + 1) = Suc n * (Suc n + 1)
18            by simp
19          finally show ?case .
20        qed
```

For the sake of this presentation, we appeal to an "immediate intuition" of a mathematically knowledgable reader; for detailed introduction into the structured proof language, the reader is referred to the Isar Reference Manual of the System documentation.

Locales can be directly referred to in proofs. For example, one could in a constructivist version of **hol!** (see `src/HOL/ex/Higher_Order_Logic.thy`) state and prove:

```
1   locale classical =
2   assumes classical: ( A ⟹A) ⟹A
3   theorems (in classical)
4    Peirces_Law: ((A →B) → A) → A
5    proof
6    ...
7    qed
```

Thus, the effect of the "(in classical)" clause in the example above is to add additional assumptions into the local context. A skeptical evaluator might therefore insist on proofs of the existence of witnesses for the locale, i. e. a proof for $\exists x.\,\mathrm{partial\_order}\ x$. Since in a classical setting the existence of a function can be stated via the Hilbert-operator, that decides for a Turing machine that it terminates for a given input, a very skeptical evaluator might even insist on a constructive witness for this existence proofs.

2.8 Isabelle/HOL system features

Finally, Isabelle/HOL manages a set of *executable types and operators*, i. e., types and operators for which a compilation to **sml!**, OCaml, Scala, or Haskel is possible. Setups for arithmetic types such as int have been done allowing for different trade-offs between trust and efficiency. Moreover any datatype and any recursive function are included in this executable set (providing

that they only consist of executable operators). Of particular interest for evaluators is the use of the Isar command:

$$\textbf{valid} \quad \text{sort}[1,7,3] \tag{2.4}$$

In the context of the definitions subsection 2.5.3, it will compile them via the code-generator to SML code, execute it, and output:

$$[1,3,7] \tag{2.5}$$

This provides an easy means to inspect constructive definitions and to get easy feedback for given test examples for them. See the part "Code generation from Isabelle/HOL theories" by Florian Haftmann from the Isabelle system documentation for further details..

Of particular interest for evaluators or certifications are Isabelle's features for semantically supported typesetting: within the document element:

```
text{* This is text containing $\lambda$'s and $\beta$'s ... *}
```

for example, arbitrary LaTeX code can be inserted for using technical and mathematical notation of annotations of formal document elements. Inside a text-document, the *document antiquotation* mechanism already mentioned in subsection 2.5.7 can be applied:

```
text{* Text containing theorems like {@thm exI} ... *}
```

which results in a print of theorems directly from their formal Isabelle presentation. It is possible to define new antiquotations, for example to track security requirements or security claims in theorems or tests. A detailed description of document antiquotations is found in the "Isar Reference Manual" by Makarius Wenzel from the Isabelle system documentation. It is even possible to define own antiquotations in Isabelle for categories of the common criteria like protection profiles, security targets, requirements, security properties etc. For all these entities, be it informal or formal, declarations and applications of antiquotations can be used in text fragments that allow for a direct consistency checking over the entire document. Since a concrete setup for such mechanism offers a number of deviation points, we refrain in this document on *mandatory* recommandations and refer to a future document on *styleguide recommandations*.

During a certification process, evaluators are encouraged to use the Isabelle/jedit user-interface directly (and not just the generated .pdf documentation), since it allows for an in-depth inspection and exploration of the formal content of a theory: tooltips reveal typing information, evaluations of critical expressions can often be done by the `value ...` document item, and operator-symbols occurring in HOL-expressions were hyper-linked to referring definitions or binding occurrences. Note, however, that a user-interface

is a dozen system layers away from a Isabelle inference kernel which opens the way for implementation errors in display and editing components, increasing the risk of misinterpretations. A final check of an entire document should therefore be made in the (GUI-less) build mode (which enforces also stronger checking).

Chapter 3

Isabelle/HOL in certification processes

3.1 Introduction

Recently, theorem provers have been widely used in the area of computer systems security and certification and, for instance, in Common Criteria. The Common Criteria (CC) [Mem06] is a well-known and recognized computer security certification standard. The standard is centered around the role of the *developer*, who provides implementation but also "artefacts of compliance with the level of security targeted", while the *evaluator* "confirms the compliance of the information supplied" as well as determines "completeness, accuracy and quality" of the deliverables.

Especially wrt. "completeness, accuracy and quality" of specifications and proofs, formal methods and especially mechanically proof checking techniques can push the trust and the reproducibility of the results to levels not obtainable by a human certification expert alone. This explains why at its higher assurance levels, the CC requires the use of formal methods for specification and verification. A well-established formal specification formalism must be used to model the system and of the different security policies. A reliable theorem prover is needed to prove and verify different properties of the specification. Recent theorem provers offer rich and powerful formal environments that are very suitable for both activities.

Among the important number of theorem provers available nowadays, we concentrate on the Isabelle theorem prover[1]. Following [Hal08], the Isabelle System, developed into one of the top five systems for the logically consistent development of formal theories. In particular the instance of the Isabelle system with higher-order logic called Isabelle/HOL is therefore a natural choice as a formal methods tool as required by the Common Criteria on the

[1] At time writing, the current version is Isabelle2013-2.

higher assurance levels EAL5 to EAL7.

As a contribution, the chapter culminates in some high-level mandatory guidelines and recommendations for both developers and evaluators of certification documents using Isabelle. It attempts to be a complement to [Jae08].

3.2 Common Criteria: Normative Context

For high levels of certification (i.e. for EAL5 to EAL7) in the Common Criteria [Mem06] some requirements introduce the use of formal methods at diverse phases of the design process. Regarding to the level of security target required, the use of formal methods match different objectives.

3.2.1 Certification Level: different use of formal methods

The next table resumes for each level the main requirements and impacts from formal methodology point of view.

level EAL	Objective
EAL5	This EAL represents a meaningful increase in assurance from EAL4 (methodically designed, tested, and reviewed) by requiring semiformal design descriptions, a more structured (and hence analysable) architecture, and improved mechanisms and/or procedures that provide confidence that the TOE will not be tampered with during development.
EAL6	This EAL represents an important increase in assurance from EAL5 (semiformally designed and tested) by requiring more comprehensive analysis, a structured representation of the implementation, more architectural structure (e.g. layering), more comprehensive independent vulnerability analysis, and improved configuration management and development environment controls.
EAL7	This EAL represents a meaningful increase in assurance from EAL6 (semiformally verified design and tested) by requiring more comprehensive analysis using formal representations and formal correspondence, and comprehensive testing.

In addition to the normative definitions of the security levels, the CC standard defines the possibility of intermediate levels of security when a requirement is evaluated at an higher level than required by the level targeted. The addition of the symbol "+" represent this kind of evaluation (for example EAL4+).

With regard to high level certifications, the requirements on formal methods are more and more intrusive and the models are more and more detailed

(from a high level architecture for EAL5 to a structured formal design for EAL7).
The Common Criteria defines two different roles the *developer* and *evaluator*. These two different roles shall comply different requirements of CC or more precisely, each requirement of CC is declined in different actions for developer and evaluator. From a general point of view, the developer shall:

- realize the design, the documentation, the implementation and the validation of the target.
- provide artifacts and elements of compliance with the level of security targeted.

In another hand the evaluator shall:

- confirm the compliance of the information supplied (by the developer) with requirements of the security level,
- determine the completeness, the accuracy and in a general manner the quality of the deliverables.

This chapter is intended to detail these two tasks of an evaluator with respect to implementations and validations done with the Isabelle/HOL system.

3.2.2 Requirements addressed by formal or semiformal models

With regard to high level certifications, the main requirements addressed by the use of formal methods are:

- ADV_SPM.1 requires a formal TOE *security policy model*(SPM for short). This model is generally a high level model which capture the main security properties and abstract behavior of the target.
- ADV_FSP.6 requires a semi-formal *functional specification* (FSP for short) with an additional formal specification. This second constraint concerns an intermediate functional level of design and it is considered as pertinent from a formal point of view, to manage a formal model (and not a semi-formal) which is a refinement of the initial model defines for ADV_SPM.1. The use of this intermediate formal model is efficient to define a formal specification.
- ADV_TDS.6 requires a complete semi-formal and modular design with high-level *(TOE) design specification* (TDS). This final design requirement introduces the architecture of the target and the notions of modules and interfaces. The main objective is to define and simply specify the structure of the design and provide a proof of correspondance

between specifications of the subsystems and the functional specification.

This simple overview of some CC requirements implies that a formal approach based on a formal refinement definition is compliant to assure consistency between the different models and the diverse views and objectives considered in these requirements.

3.2.3 Formal methods: other requirements impacted

Another requirements for high level certification (EAL7) are not directly connected with formal methods but they can be addressed by the use of formal methods (see chapter 3.4):

- ATE_FUN.2: The objectives are for the developer to demonstrate that the tests in the test documentation are performed and documented correctly, and to ensure that testing is structured such as to avoid circular arguments about the correctness of the interfaces being tested. Although the test procedures may state pre-requisite initial test conditions in terms of ordering of tests, they may not provide a rationale for the ordering. An analysis of test ordering is an important factor in determining the adequacy of testing, as there is a possibility of faults being concealed by the ordering of tests.
- AVA_VAN.5: A methodical vulnerability analysis is performed by the evaluator to ascertain the presence of potential vulnerabilities. The evaluator performs penetration testing, to confirm that the potential vulnerabilities cannot be exploited in the operational environment for the TOE. Penetration testing is performed by the evaluator assuming an attack of High potential.

The generation of test cases from a formal model can be an interesting approach to optimize the efforts of the modeling and the formal proof: this formal specification-based testing approach is not a classic approach in industrial world but seems compliant with the two previous CC requirements.

3.3 Methodological Recommendations for the Evaluator

As said earlier, there are four potential dangers of a formal proof system that it wrongly accepts the desired theorem "This operating system is secure":

1. Inherent inconsistency of the logics (e. g., **hol!**) or inconsistent use of the logics (introduction of inconsistent axioms by one way or the other).

2. The incorrect implementation of Isabelle the Isabelle Kernel and of the **hol!** instance in it.

3. The incorrect package implementation realizing advanced specification constructions like type definitions etc.

4. Since Isabelle is highly configurable, there is a certain danger of obfuscation of bogus-proofs.

Beyond the more philosophical objections[2], the risk outlined by the by first item is in fact **minimal**: Higher-order logic is an extremely well studied object of academic interest [And86, GM93], and while there are known limits in proving soundness and completeness inside a **hol!**-prover, they just stimulated a lot of recent research to come a "formal proof over **hol!** in **hol!**" as close as possible, e.g. by adding to **hol!** an axiom over the existence of a sufficiently large cardinal [Har06, MOK13].

The risk outlined by the second item is also **very small**. The reasons are threefold:

A Some of the aforementioned soundness proofs cover also the implementation aspects of the core of a provers of the **hol!**-family (**hol!**-light, ...).

B The specific architecture of provers of the LCF family (HOL4, Isabelle, HOL-light, Coq) enforces that any proof is actually checked by by this fairly small core.

C These core-inferences can optionally be protocoled in an proof-object which can, in principle, in case of serious doubt be checked by another implementation of a **hol!**-prover. However, since these objects tend to be very large, this approach requires decent engineering. Fortunately, this should only be necessary in exceptional cases.

The risk of the third item is **minimal** as far as the described standard conservative standard extension schemes such as `type_synonym`'s, `datatype`'s, `definition`'s and `fun`'s, `typedef`'s, `specification`'s, `inductive`'s, typeclasses and locales are concerned. The same holds for diagnostic commands like `type`, `term`, `valid`, etc. that do not change the global context of a theory. These are fairly well-understood schemes which have in parts been proven formally correct for similar systems such as the HOL4 system[KAMO14]. These schemes cover the largest parts of the Isabelle/HOL libraries. Here lies the main advantage of the LCF-approach and the methodology to base libraries on conservative (logically safe) definitions.

[2]For example, the fundamental doubt in the existence of infinite sets[And86]...

The risk is **small** as far as other standard extension schemes are concerned; since extension schemes generate internally axioms, there have been reported consistency problems with combinations of other extension schemes such as `consts` and `defs` as well as `defs (overloaded)`; the Isabelle reference manual points out that the internal checks of Isabelle do not guarante soundness.[3]

It remains the risk of item four, which is concerned with the resulting methodology in "how to use Isabelle". For very large theory documentations, it must be considered **non-negligeable**. It is the key-issue addressed in the remainder of this section.

3.3.1 On the use of SML

As mentioned earlier, Isabelle is an open environment that allows via

```
1   ML{* SML ML code *}
```

to include arbitrary SML programs, in particular programs that make direct inferences on top of the kernel. Per se, this use of Isabelle is not unsafe; critical parts of the **hol!** library use this mechanism. Isabelle is designed to have user land SML code extensions, and the kernel protects itself against logical inconsistencies coming from ML extensions. However, there are a few deliberate opt-outs, and furthermore, it is in principle possible to obfuscate them in Isabelle ML code such that an evaluator may be fooled by a text appearing to be an Isabelle proof but isn't in the sense of the inference kernel. Thus, besides the principle possibility that a pretty-printed theorem does not state what it appears to state by some misuse of mathematical notation (an inherent problem of any formal method), there is the possibility of fake-proofs as a consequence of ML code and (re)-configurations of the ISAR proof language.

If SML-code is accepted in an evaluation, it has to be made sure — potentially by extra justifications or external experts with Isabelle implementation expertise — that this code does not implicitly generate axioms, registers oracles and defines proof methods equivalent to `sorry` (or variants like `sorry_fun`) to be discussed in the sequel; in any case, the evaluation is substantially simpler if SML-code is strictly avoided.

3.3.2 Axioms and Bogus-Proofs

Obviously, when using the Isar `axiomatization` construct allowing to add an arbitrary axiom, it is immediately possible to bring the system in an inconsistent state. The immediate methodological consequence is to ban it

[3] See Isabelle Isar-Reference Manual (Version 2013-2, pp. 103): "It is at the discretion of the user to avoid malformed theory specifications!"

from use in to be evaluated theories completely (such that it is only internally used inside specification constructs in and and in the aforementioned foundational axioms coming with the system distribution) and to restrict theory building on conservative extensions. This is also common practice in scientific conferences addressing formal proof such as ITP.

However, there are more subtle ways to introduce an axiom that leads to inconsistency. First, there is a mechanism in Isabelle to register *oracles* into the system. They can be used for a particularly simple, but logically unsafe integration of external provers into Isabelle and can be used inside self-defined tactics. Logically, an oracle is a function that produces axioms on the fly. It is an instance of the axiom rule of the kernel, but there is an operational difference: The system always records oracle invocations within proof-objects of theorems by a unique tag. Of course, oracle invocations should again be avoided in a certified proof.

A particular instance of the oracle mechanism is the `sorry` proof method. This is method is always applicable and closes any (sub)-proof successfully, and a useful means in top-down proof developments in Isabelle. Unnecessary to repeat that no `sorry` statements should remain in a proof document underlying certification. By the way, the system is by default in a mode in which it refuses to generate proof documents containing `sorry`'s, only by explicitly putting it in a mode called `quick_and_dirty` this can be overcome. There are several ways to activate `quick_and_dirty`, by it by explicit ML statements like `quick_and_dirty:=true`, be it in the `ROOT.ML`-files (till version 2013-1), or be it in the session- configuration files `ROOT`-files (since version 2013).

Oracles and `sorry`'s are particularly dangerous in methodological foundation proofs (type or type-class is non-empty, recursions well-founded), since the use of the the oracle-tag inside the corresponding proof-objects gets lost on the level of type expressions. Thus, a `sorry` could introduce inconsistent types whose "effects" could be used in bogus-proofs depending on them.

We will discuss this a little more in detail: Recall that deduction in Isabelle/**hol!** is centered around the requirement that types and type-classes are non-empty. This is a consequence of the fact that the β-reduction rule $((\lambda\, x :: \tau.E)E' \rightarrow E[x := E'])$ is executed pervasively during deduction, be in in resolution or rewriting steps. It is well-known however, that β-reduction is unsound in the presence of empty types[4]. Thus, an obfuscated `sorry` in a methodological proof leaves no other than very local traces in the proof objects and can be exploited much later via an inconsistent type in a proof based on this type definition; the exploit could again be obfuscated by an-

[4]Consider the case of τ having a semantic interpretation into an empty set $I(\tau) = :$ then the semantic interpretation of the function $(\lambda\, x :: \tau.E)$ must be in the function space: $^D =$ where D is the space of interpretations for the type τ' of E. Obviously, there is no possible result for the application ...

other self-defined proof-method, say `auto'` which will be hard to detect by inspection. The only systematic way to rule out obfuscated bogus-proof is either by ruling out ML-constructs or by checking *all* proof objects of the entire theory.

3.3.3 On the use of external provers

The Isabelle distribution comes with a number of external provers, namely:

- `sledgehammer` : its use is uncritical, since it remains completely extern to proof documentations and is only used for the generation of high-level Isabelle proofs, that were certified by the kernel.
- `blast`, `metis`: these are internal devices but also uncritical, since their results were used via a proof object certification.
- `smt`: this method uses, for example, the external SMT-solver Z3. The integration is carefully made and uses no oracles - instead, a form of tactical proof re-construction mechanism is used [BW10] that is logically safe.

Other external provers have to be considered carefully; in particular integrations using the oracle-mechanism should be ruled out.

3.4 Extensions of Isabelle: Guidelines for the Evaluator

As said earlier, the ML code should be considered harmful in theories to be evaluated. There are, however, a number of add-ons on Isabelle, which can be considered as tools in their own right and which heavily use ML code inside.

3.4.1 An Example: The Isabelle/Simpl

Isabelle/Simpl is an verification environment built conservatively on Isabelle/HOL. It supports a sequential imperative programming language, for which it defines its syntax, semantics, Hoare Logics and a verification condition generator (again derived), which form together a complete verification environment. Together with an (untrusted) parser that compiles C programs into Isabelle/Simpl[GAK12], this particular environment follows a similar program verification technique like Frama-C/Why/AltErgo ([CKK+12, FP13],alt-ergo.lri.fr) or VCC/Boogie/Z3[BW10].

The entire environment is part of the Isabelle-oriented "Archive of formal Proofs", see afp.sourceforge.net in general and afp.sourceforge.net/entries/Simpl.shtml in particular.

The environment has been used for one of the most ambitious code-verification projects recently, the verification of the L4-Microkernel (cf. www.ertos.nicta.com.au/research/l4.verified, [KEH+09]).

In itself, Isabelle/Simpl can be considered nearly as as "trustable" as Isabelle/HOL itself : the library is built upon conservative extensions of the HOL -kernel, and the ML extensions are done by Isabelle developers themselves and stood the test of the time. Program verification proofs establishing that a Simpl-program is correct with respect its (pre-post-condition) specifications can be handled by the same evaluation procedures as any other Isabelle development.

However, as in any process involving the verification of C programs, the C parser and its transition from "real C" to the idealized imperative language Simpl has to be considered with a wise dose of scepticism. Here is are whole spectrum of different glimpses possible: since the C parser defines a semantics-by-translation for its fragment of C, the question remains unproven that this semantics is faithful to the semantics of the real C compiler generating production-level code (which involves questions on compiler correctness, semantic faithfulness of the execution environment, correctness of compilation optimizations, hardware-correctness, etc.). The problem has been addressed via particular validation techniques of the parsing process [GAK12], but is, in full generality, unsolvable.

3.4.2 HOL-TestGen

HOL-TestGen[5](see Figure 3.1) is an interactive, i. e., semi-automated, test generation tool for specification-based tests built upon Isabelle/HOL. Instead of using Isabelle/HOL as "proof assistant," it is used as modeling environment for the domain specific background theory of a test (the *test theory*), for stating and logically transforming test goals (the *test specifications*), as-well as for the test generation method implemented by Isabelle's tactic procedures. In a nutshell, the test generation method consists of:

1. a *test case generation* phase, which is essentially an equivalence partitioning procedure of the input/output relation based on a **cnf!**-like normal form computation,

2. a *test data selection* phase, which essentially uses a combination of constraint solvers using random test generation and the integrated SMT-solver Z3 [dMB08],

[5]HOL-TestGen was never used to: test complex real systems in the marcket, and concurrent code before this thesis

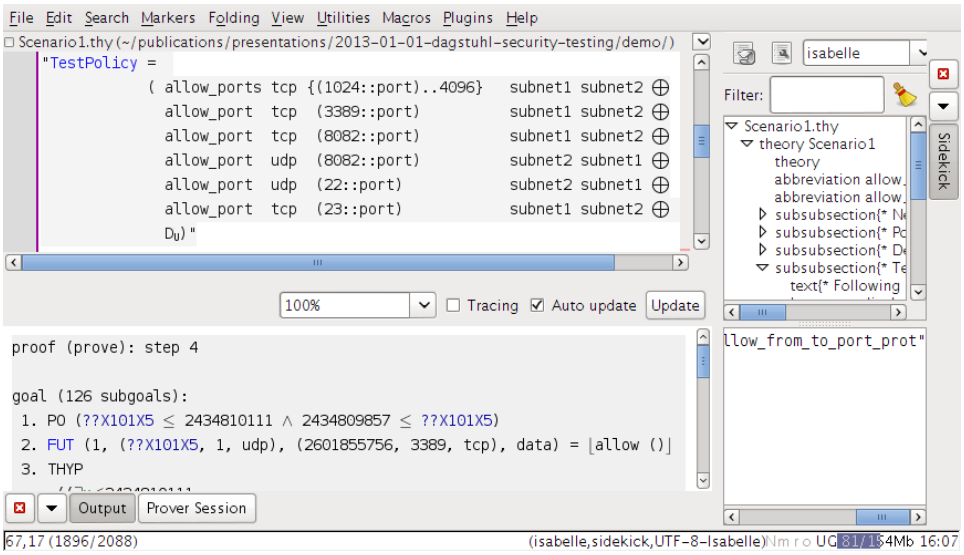

Figure 3.1: An Isabelle session showing the jEdit client as Isabelle Interface. The upper-left sub-window allows one to interactively step through a test theory comprising test specifications while the lower-left sub-window shows the corresponding system state of the spot marked in blue in the upper window.

3. a *test execution* phase, which reuses the Isabelle/HOL code-generators to convert the instantiated test cases to test driver code that is run against a system under test.

A detailed account on the symbolic computation performed by the test case generation and test selection procedures is contained in [BBKW12]. The test case generation method is basically an *equivalence partitioning* combined with a variable splitting technique that can be seen as an *(abstract) syntax testing* in the sense of the ISO 29199 specification [Int12, Sec. 5.2.1 and 5.2.4].

The equivalence partitioning separates the input/output relation of a program under test (PUT), usually specified by pre- and post-conditions, into classes for which the tester has reasons to believe that PUT will treat them the same.

Of course, the HOL-TestGen approach inherits all glory, but also all limitations of a testing approach: The entire specification is reduced via specific *test purposes* and underlying *test hypothesis* ("pick one out of the equivalence class, and it's going to be ok for all class members") to a *finite* number of tests to be checked. These purposes and hypothesis' may be difficult to justify and need careful inspection, more difficult than having just a universal statement over the entire input/output relation. On the other hand, testing can estab-

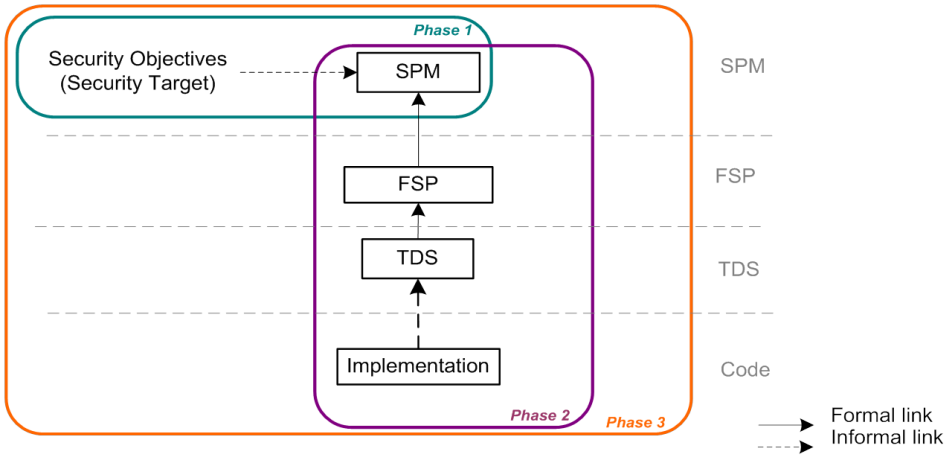

Figure 3.2: Refinement steps for a formal development approach compliant to CC

lish confidence over the **real system**, and makes no modeling assumptions — like the Simpl-approach subsection 3.4.1 — over the underlying hardware, the correct modeling of behavior of hardware components such as sensors, the compiler, and the equivalence of the assumed operational semantics of the used programming language(s) with the actually executed one. For this reason, it can be safely stated that for certifications of the highest-levels, a suitable *combination* of test and proof techniques will be necessary. Proofs for the higher levels of the models establishing the desired security properties in a *Target Of Evaluation* TOE, tests for establishing that the assumptions made in the lower levels of the models correspond to the reality in the TOE.

3.5 Recommendations for CC certifications

3.5.1 A refinement based approach for CC evaluation

The figure 3.2 presents a refinement scheme which implements different refinement steps from security policy model SPM to implementation. With this approach, the properties demonstrated on an abstract SPM are formally preserved down to the levels of the functional specification model FSP and a TOE specification design model, the TSD. At each level of abstraction the dedicated model and its associated proofs demonstrate the security properties and are compliant with the CC requirement. The use of a formal refinement methodology demonstrates the consistency between each refined model and preserve the properties demonstrated at high level of abstraction.

The evaluation of this kind of approach can be conducted in three different phases by the evaluator:

- Phase 1: Verification of the proof of the SPM formal specification. On the initial abstract model, a verification shall be conducted to check the relevance of the security objectives modeling in the formal model with the informal specification. A second point is the verification of the model soundness to assure than the model is not inconsistent (refers to chapter 3.3.2).

- Phase 2: Refinement of the SPM formal specification. A first step of this phase is the verification of the refinement process and methodology. On each refinement, verifications on the properties and on the soundness of the model are conducted. From the initial abstract model, on each intermediate concrete model, the evaluator checks the traceability (i.e. the traceability of the requirements) between models. An informal link can be considered between the last formal model of the TDS and the implementation. A bi-directional detailed traceability of the security requirements shall be managed between this two different artefacts to verify the implementation of the security requirements and than the implementation contains only desired requirements [6].

- Phase 3: General and transverse activities. This last phase consists mainly of the verification on the proofs and on justifications on the tools used as support for development and design. The complete traceability from the security target to the implementation is verified included traceability between each refinement steps of formal models. During this phase, the evaluator replay the proofs and check the consistency of the formal properties and assumptions defined on the environment and the context (see 2.8 last paragraph for details of facilities supplied by Isabelle/HOL. The use of keywords to report the proof of parts of the proof obligations is forbidden (for example the use of the `sorry` proof method, see chapter 3.3.2 for details).

When formal methods are used, some practices should be applied to facilitate the work of the evaluator and be more efficient.

- Formal models should be defined in accordance with some naming convention informations and is a huge help for traceability.

- Formal models should be define in accordance with "coding" rules ([Jae08]). The proofs associated can be replay.

- Documentation and deliveries should respect templates and integrate traceability with requirements or elements from input specifications. From this point, the use of the Isabelle interface should be interesting with regard to its functionalities, refers to 2.8.

[6] to check than no parts of the code violate the security properties by side effects.

Chapter 4

Summary

4.1 Background References

The most notable text describing the scientific history behind the LCF-family of **hol!** provers is done by by Mike Gordon[Gor00]. It covers the beginning of the entire research programme from 1972 to the mid-80ies, ranging from foundational issues of the logic over contributions to type-systems (as the "Hindley-Milner-Polymorphism")[Mil78] to the issue of the practical, safe implementation of rewrites and decision procedures [Pau99].

The LCF research programme was in parallel to another notable source of nowadays interactive theorem proving technologies: the Automath-project. In 1968, N.G. de Bruijn designs the first computer program to check the validity of general mathematical proofs, using typed λ-calculi as a direct means to represent proof objects as such. The emphasis of this programme was initially on proof-checking; de Bruijn's system Automath eventually checked every proposition in a primer that Landau had written for his daughter on the construction of real numbers as Dedekind cuts. A descendant of this family, which also has deeply influenced the Isabelle kernel design (proof objects, core inferences) is the Coq system (see http://coq.inria.fr).

Another notable survey on research programme is contained in the papers contained in *A Special Issue on Formal Proof* distributed by the American Mathematical Society (see http://www.ams.org/notices/200811/, but also [Hal08]), which presents nicely the relevance of modern ITP technology for purely mathematical problems (an argument, which has been strengthened recently by the formal proof of the Feit-Tompson theorem, whose precise formulation has haunted mathematicians for decades [Gon13], and the formal proof of the Kepler-conjecture, which is a known mathematical problem for about 400 years.).

4.2 Concluding Remarks and a Summary

We have presented the Isabelle/**hol!** system and pointed out the essential arguments, why by a particular combination of system-architecture and methodology, the system is suited to give the currently highest possible guarantee on a formal proof in particular and a logical theory development in general. In a sense, Isabelle/**hol!** offers the same guarantees for logical systems as Coq[Jae08], and in some sense better guarantees than, for example, the B method or model-checkers like FDR. Isabelle/**hol!** is therefore a natural choice for evaluations in the higher certification levels EAL5 to EAL7 in the Common Criteria (CC) [Mem06].

If the methodological side-conditions are respected which can be reduced essentially to an number syntactic checks, the formal consistency of the entire certification document containing formal specifications, proofs of consistency and the proofs of security properties, refinement-proofs between the different abstraction layers, and finally test-case generations as well as test-results can be guaranteed, and the evaluator can therefore concentrate on the more fundamental questions: does the model represent the right thing? are the modeling assumptions justified?

As "take-home-message" we would summarize these side-conditions as follows:

- Use a trusted, unmodified Isabelle version from the distribution.
- Check the restriction to definitional axioms only, enforce the use of "safe" specification constructs discussed here.
- Rule out `axiomatization`, `sorry`, their variants or disguised equivalents (such as oracle declarations).
- In particular `sorry`'s or equivalent constructions in methodological proofs have to be ruled out.
- Check the quick-and-dirty mode status.
- Exploring a TOE interactively, for example by jEdit, which allows for inspecting theories and definitions, their animation, the checking of types and of proof details, is a great means to increase confidence for an evaluator. However, the final check should be done in a non-interactive mode (pretty-printing and display machinery is actually quite far from the kernel and can be erroneous in itself).
- The main theorem in an CC evaluation is presumably of the form: "the security property X stated in the context of the security model Y is satisfied for the functional model Z under some conditions A in some

locale B". A skeptical evaluator may insist on proofs that A and B are actually satisfiable, under circumstances even in a constructive sense.

- A conservative evaluator should restrict or ban `ML`-statements (with the possible exception of declarations of antiquotations), otherwise inspect `ML`-statements with particular care.

The internal code generator (also used in `code`-antiquotations and `value`-statements) stood the test of the time, but enjoys not quite the same level of trust as the proof facilities. The generation of proof objects for a complete theory is in principle possible, but should not be necessary except in case of a concrete suspicion of a fraudulent proof attempt.

Bibliography

[And86] Peter B. Andrews. *An introduction to mathematical logic and type theory: to truth through proof*. Academic Press Professional, Inc., San Diego, CA, USA, 1986.

[And02] Peter B. Andrews. *Introduction to Mathematical Logic and Type Theory: To Truth through Proof*. Kluwer Academic Publishers, Dordrecht, 2002.

[Bal10] Clemens Ballarin. *Tutorial to Locales and Locale Interpretation*, 2010.

[BBKW12] Achim D. Brucker, Lukas Brügger, Matthias P. Krieger, and Burkhart Wolff. HOL-TestGen 1.7.0 user guide. Technical Report 1551, Laboratoire en Recherche en Informatique (LRI), Université Paris-Sud 11, France, 2012.

[BW10] Sascha Böhme and Tjark Weber. Fast lcf-style proof reconstruction for z3. In *ITP*, pages 179–194, 2010.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, pages 56–68, June 1940.

[CKK+12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: a software analysis perspective. In *International Conference on Software Engineering and Formal Methods (SEFM'12)*, pages 233–247. Springer, October 2012.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS*, 2008.

[FP13] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013.

[GAK12] David Greenaway, June Andronick, and Gerwin Klein. Bridging the gap: Automatic verified abstraction of c. In *ITP*, pages 99–115, 2012.

[GM93] Mike J. C. Gordon and Tom F. Melham. *Introduction to HOL*. Cambridge University Press, 1993.

[Gon13] Georges Gonthier. Engineering mathematics: the odd order theorem proof. In *POPL*, pages 1–2, 2013.

[Gor00] Mike Gordon. From lcf to hol: a short history. In *Proof, Language, and Interaction*, pages 169–185. MIT Press, 2000.

[Hal08] Thomas C Hales. Formal proof. *Notices of the AMS*, 55(11):1370–1380, 2008.

[Har06] John Harrison. Towards self-verification of hol light. In *IJCAR*, pages 177–191, 2006.

[Int12] ISO/IEC DIS 29119: Software and Systems Engineering—Software Testing. ISO Draft International Standard, July 2012.

[Jae08] Eric Jaeger. Remarques relatives á lï£¡emploi des m éthodes formelles (déductives) en sécurité des systémes dï£¡information. 2008. 51 Boulevard de la Tour-Maubourg 75700 Paris SP 07, France.

[KAMO14] Ramana Kumar, Rob Arthan, Magnus O. Myreen, and Scott Owens. HOL with definitions: Semantics, soundness, and a verified implementation. In *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, pages 308–324, 2014.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES*, pages 207–220. ACM, 2009.

[Mem06] The Common Criteria Recognition Agreement Members. Common criteria for information technology security evaluation. http://www.commoncriteriaportal.org/, Sep 2006.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348 – 375, 1978.

[MOK13] Magnus O. Myreen, Scott Owens, and Ramana Kumar. Steps towards verified implementations of hol light. In *ITP*, pages 490–495, 2013.

[MTM97] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[MW79] R. Milner and C.P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*. Lecture Notes in Computer Science. Springer, 1979.

[Nip12] Tobias Nipkow. Theory fun, 2012.

[Nip13] Tobias Nipkow. Hol/list.thy, 2013.

[NP00] Tobias Nipkow and Lawrence C Paulson. Hol/nat.thy, 2000.

[NPW02] Tobias Nipkow, Larry C. Paulson, and Markus Wenzel. *Isabelle/**hol!**—A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, 2002.

[NPW14] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. Theory set, 2014.

[NWP13] Tobias Nipkow, Markus Wenzel, and Larry Paulson, Dec 2013.

[NWS+] Wolfgang Naraschewski, Markus Wenzel, Norbert Schirmer, Thomas Sewell, and Florian Haftmann. Theory record.

[Pau99] Lawrence C. Paulson. A generic tableau prover and its integration with isabelle. *J. UCS*, 5(3):73–87, 1999.

[PP10] Leaf Petersen and Enrico Pontelli, editors. *Proceedings of the POPL 2010 Workshop on Declarative Aspects of Multicore Programming, DAMP 2010, Madrid, Spain, January 19, 2010*. ACM, 2010.

[WB89] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *POPL*, pages 60–76, 1989.

[Wen97] Markus Wenzel. Type classes and overloading in higher-order logic. In *TPHOLs*, pages 307–322, 1997.

[Wen02] Markus M Wenzel. *Isabelle/Isar—a versatile environment for human-readable formal proof documents*. PhD thesis, Technische Universität München, Universitätsbibliothek, 2002.

[Wen15] Makarius Wenzel. *The Isabelle/Isar Reference Manual*, May 2015.

[Wie06] Freek Wiedijk. *The Seventeen Provers of the World: Foreword by Dana S. Scott (Lecture Notes in Computer Science / Lecture Notes in Artificial Intelligence)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.