Transformational Program Development in the UniForM Workbench

Christoph Lüth
Bremen Institute for Safe Systems, FB 3, Universität Bremen
Postfach 330440, 28334 Bremen
cxl@informatik.uni-bremen.de

April 23, 1997

Abstract

In this paper, the modelling of transformational program development inside a tactical theorem prover is described, together with a systematic way of building graphical user interfaces for applications based on a tactical prover. Combined with logical embeddings of formal methods into the prover, this yields a unifying framework for tool-supported formal program development.

The work described here uses the theorem prover Isabelle, but is applicable to other tactical theorem provers as well.

1 Introduction

During recent years, the need for formal methods in software development has been recognised increasingly. Along with this recognition, awareness has grown that “there is no single theory for all stages of the development of software” (C. A. R. Hoare [Hoa96]), and that formal methods have to be supported by appropriate tools. Thus if formal methods are to gain industrial relevance there is the need for a framework integrating different formal methods and formal methods tools.

The main aim of UniForM project [KPO+95] is to develop such a framework. Different formal methods and tools are combined into one universal development environment, with a common user interface and repository management covering the whole of the software life cycle.

The integration of different formal methods and tools proceeds on two levels. On the first level, we can directly combine existing tools in the UniForM workbench. On the second level, different formal methods are combined by logically embedding them into the tactical theorem prover Isabelle/HOL [Pau94]. Based on this embedding, one can uniformly describe transformation program development in the context of different formal methods such as Z or CSP. These two levels of combination are complementary: the flexibility and generality of the tactical theorem prover is complemented by tools that directly support one formal method — such as FDR [For95] to support CSP — in which they can possess remarkable proof power.
The methodology of the UniForM workbench emphasises transformational program development. This paper describes our modelling of transformational program development inside a tactical theorem prover; and it sketches a way in which graphical user interfaces for applications based on tactical theorem provers can be built.

2 Transformational Program Development in Isabelle

We will give a short introduction into LCF provers in general. The material covered here is introductory; for details of the theoretical background, see [KSW96].

2.1 LCF Theorem Provers and Isabelle

The family of LCF theorem provers originate from the seminal Edinburgh LCF system by Milner, Wadsworth and Gordon in the 70's [GMW79]. Their main characteristics are the way they are embedded into the functional programming language ML\(^1\), and their flexibility due to so-called tactics.

An LCF prover essentially consists of a collection of ML modules and functions. The user types ML expressions or ML programs, which are evaluated and their result printed. This design obviously leaves something to be desired as far as user-friendliness is concerned, but it is very powerful and extendible. Using specific functions provided by the prover, the user can program tactics as ML programs, allowing a higher degree of proof automation.

The main contemporary LCF provers are the HOL system [GM93] and Isabelle [Pau94], the latter of which we will describe in the following. Isabelle differs from other LCF provers in three respects:

- Firstly, it is generic over the logic it uses. Whereas for example the original LCF was based on the logic of computable functions (hence the acronym LCF) as developed by Scott and Strachey at the end of the 60’s, Isabelle can be instantiated to a wide degree of logics, ranging from simple first-order logic, sequent calculus to modal logic, constructive type theory and Zermelo-Fränkel set theory.

  For our purposes, we use Isabelle/HOL, the instantiation with Church’s classical higher-order logic.

- Secondly, it has the concept of a meta-variable. Meta-variables can be regarded as “holes” in a term which can be filled in by substitutions during unification. The efficient handling of meta-variables by Isabelle alleviates the user of the need to prove side conditions of the “Variable \(x\) is not free in \(P\)” when applying a rule.

- Thirdly, Isabelle has a powerful rewriter, which can use theorems of the form \(s = t\) as rewrite rules in order to automate proofs.

\(^1\)In fact, ML (the acronym stands for Meta-Language) was conceived as a command language for the LCF prover in the first place.
2.2 Proof in Isabelle

There are two basic proof methods in Isabelle: *backward resolution* and *forward resolution*.

Forward resolution is a way to derive new theorems: if we have two theorems $P \leftarrow \frac{Q}{S}$ and $\frac{P}{T}$, and we can find substitutions $\sigma, \tau$ such that $\sigma(Q) = \tau(R)$, then we can derive (by the associativity of the implication) the new theorem $\frac{\sigma(P)}{\tau(S)}$.

Backward resolution drives the proof activity. If we wish to prove a goal $P$, then backward resolution with a theorem $\frac{Q_1 \ldots Q_n}{Q}$ means that we find a substitution $\sigma$ which when applied to $Q$ yields $P$, $\sigma(Q) = P$. To prove $P$, we now have to prove $\sigma(Q_1), \sigma(Q_1), \ldots, \sigma(Q_n)$.

These new goals are called *subgoals*. The main way to prove the subgoals is by proving them separately. Isabelle keeps track of them, and displays them as a list of numbered subgoals rather than one conjunction.

Isabelle will assist in finding the substitutions mentioned above, but this involves higher-order unification which is in general undecidable, so it is not guaranteed that Isabelle will find such a substitution if there exists one. Further, there are additional ways of proving a subgoal, like application of rewrite rules, and proof by assumption.

2.3 Transformational Program Development

In general, transformational development is described by the sequence of specifications $SP_i$:

$$SP_1 \rightsquigarrow \ldots \rightsquigarrow SP_n$$

In a full transformational development $SP_1$ is the requirement specification and $SP_n$ the executable specification (from which a program can be generated), but a transformational development may describe any subsequence of this as a single step in the overall development process, deriving a more refined specification or program from a more general one.

In this framework, we consider the $SP_i$ to be arbitrary formulae of higher-order logic, and development steps $SP_i \rightsquigarrow SP_{i+1}$ are described by a relation $\rightsquigarrow$ called the *transformation relation*. Performing the transformational development above amounts to proving formula (1) by preconceived rules called *transformations* (we will below show examples of these). Steps in the proof correspond to applying transformation rules.

A transformation is given by a *logical core theorem* of the following general form:

$$\forall P_1, \ldots, P_n. A \Rightarrow I \rightsquigarrow O$$

where $P_1, \ldots, P_n$ are the *parameters* of the rule, $A$ the *applicability condition*, $I$ the *input pattern* and $O$ the *output pattern*. $\rightsquigarrow$ is the transitive and reflexive transformation relation.

The transformational development is started by creating an initial proof state

1. $SP_1 \rightsquigarrow ?Z$

where $?Z$ is the Isabelle notation for a meta variable representing the final result.
The transformation given by the core theorem 2 is applied by performing the following sequence of tactical operations: first, a resolution with the transitivity of $\leadsto$ is carried out. This leads to a proof state with two subgoals:

1. $SP_1 \leadsto ?Y$
2. $?Y \leadsto ?Z$

where $?Y$ is the new intermediate specification and $?Z$ remains the ultimate target of the development. In the second step, $?Y$ is substituted by the transformed specification $SP_1$: by forward resolution of the logical core theorem 2 and the elimination rules for the universal quantifier $\forall$ and the implication $\exists$

\[
\forall x. P(x) \quad \text{spec} \quad \frac{?A \Rightarrow ?B \quad ?A}{?B} \quad \text{mp}
\]

one obtains the logical core theorem in a form where the variables $P_1, \ldots, P_n$ bound by the universal quantifier are substituted by meta-variables $?P_1, \ldots, ?P_n$

\[
\frac{A'}{?P_1 \leadsto ?P_n}
\]

We now find a substitution $\sigma$ such that $\sigma(I') = SP_1$, and resolve the transformed core theorem 3 with subgoal 1. The unification of the conclusion of 3 and subgoal 1 yields a substitution for the meta-variable $?Y$, which is the transformed program $SP_2 \triangleq \sigma(O')$; applying it to the applicability conditions yields the proof obligation $L \triangleq \sigma(A')$. The proof obligations will appear as a new subgoal, since they also need to be proven to make the transformation sound; in fact, the applicability condition is typically a conjunction $A = A_1 \land \ldots \land A_n$, so $L = \sigma(A_1') \land \ldots \land \sigma(A_n')$ is a conjunction as well, which will show up as $n$ subgoals. The proof state after applying the transformation thus reads

1. $\sigma(A_1')$
   
   $\vdots$

   $n$. $\sigma(A_n')$

   $n + 1$. $SP_2 \leadsto ?Z$

The application of the next transformation will be focused on subgoal $n + 1$ and so forth. This way, transformational developments can be represented within the infrastructure of Isabelle, allowing browsing and copying developments and abstract operations on them.

The sequence of steps just described forms the tactical sugar of the transformation. The tactical sugar can vary in many respects; e.g. it might contain standard proof procedures which remove trivial proof obligations, or it may use more sophisticated, semantic matching techniques [Shi96]. Hence, a transformation rule is given by a core theorem and the tactical sugar governing its application. The same core theorem can give rise to more than one transformation rule by endowing it with different tactical sugar.

A development is closed by resolving with the reflexivity law $?A \leadsto ?A$ which simply unifies the current result $SP_1$ of the transformational development with the final result $?Z$. Note that this does not imply that the specification is executable in any sense, it just allows termination of the transformational development.

The application of the next transformation will be focused on subgoal $n + 1$ and so forth. This way, transformational developments can be represented within the infrastructure of Isabelle, allowing browsing and copying developments and abstract operations on them.

The sequence of steps just described forms the tactical sugar of the transformation. The tactical sugar can vary in many respects; e.g. it might contain standard proof procedures which remove trivial proof obligations, or it may use more sophisticated, semantic matching techniques [Shi96]. Hence, a transformation rule is given by a core theorem and the tactical sugar governing its application. The same core theorem can give rise to more than one transformation rule by endowing it with different tactical sugar.

A development is closed by resolving with the reflexivity law $?A \leadsto ?A$ which simply unifies the current result $SP_1$ of the transformational development with the final result $?Z$. Note that this does not imply that the specification is executable in any sense, it just allows termination of the transformational development.
The hard part when working with real-life transformations is finding the right instantiation of the rule's parameters. Once that has been decided on, Isabelle will find the substitution $\sigma$ of the the input pattern $I'$ with the current specification. In all but the most trivial cases, this instantiation will not be found automatically by Isabelle's unification, but users will have to supply the instantiation themselves (after careful thought on their part). Note also that the unification of $I'$ with $SP_1$ can fail, which means that the transformation is not applicable here.

2.4 The Transformation Application System TAS

The Transformation Application System is designed to hide the internal tactical steps, the existence of meta variables and other Isabelle technicalities etc. from the user. It further doesn’t display the subgoals arising from proof obligations, leaving them to be proven either with a different interface to Isabelle, or to be given to external decision procedures (e.g. model checkers).

The users of the Transformation Application System will not have to worry about the details of how the transformational process is implemented within Isabelle — in fact, they will not need to have any knowledge of Isabelle at all. Since the proof of side conditions can be deferred to a later stage, the user can concentrate on the main design decisions of transformational program development: which transformation to apply, and how to instantiate its parameters.

The graphical user interface for TAS is described below.

2.5 Examples

The most simple transformation rules are those obtained by using an equation to fold or unfold it. For example, if we have a commutative binary operator $f$, then we can exchange its arguments. The core theorem for this rule reads:

$$\text{comm} \equiv \forall (\forall x,y. f\ x\ y = f\ y\ x) \Rightarrow f\ a\ b = f\ b\ a$$

Of course this theorem is rather trivially true (and hence the transformation is sound). The standard tactical sugar for this transformation would be the following ML code

```ml
fun COMMUTE n inst =
  let val commi = read_instantiate inst comm
  val m = (commi RS spec) RS mp
  in EVERY [rtac trans n, rtac m n] end
```

where `inst` is a list of pairs denoting the instantiation of the parameters, and `n` is the subgoal corresponding to the current transformational development. The function `read_instantiate` provided by Isabelle substitutes the parameters (here, only $f$) with the provided values. The next line is the forward resolution of the substituted core theorem with the elimination rules for the universal quantifier and the implication. The main line is given by the `tactical EVERY` with two tactics, `rtac trans n` and `rtac m n`, as its argument. The tactical `EVERY` means that Isabelle executes all the argument tactics in turn, until one of them fails. The two tactics are the backward resolution (`rtac`) with the
transitivity \texttt{trans}, and the transformed core theorem \texttt{m}. (Note both tactics need the subgoal as an additional argument, here \texttt{n}.)

All of this is just the sequence of steps described above written out as an Isabelle tactic.

A slightly more sophisticated version of the tactical sugar might read like the above, but have with the fourth line

\begin{verbatim}
EVERY [rtac trans \texttt{n}, rtac \texttt{m} \texttt{n},
\hspace{1em} simp_tac (arith_ss addsimps add_ac) \texttt{n}]
\end{verbatim}

The tactic \texttt{simp_tac} starts the rewriter (called the simplifier) with the rewriting set given by the rules of arithmetic (\texttt{arith_ss}) and associativity and commutativity for the addition (\texttt{add_ac}). This tactic would automatically prove, and hence discharge, the proof obligation arising from applying \texttt{COMMUTE} if one instantiates the argument with the addition.

An example of a realistic transformation in this framework, showing that our approach also scales up to real-life transformations, has been given in [KSW96]. Both the core theorem and logical sugar of the transformations implemented there are far too involved and lengthy to be shown here.

### 2.6 Transformations for Other Formal Methods

The transformational development paradigm also carries over to other formal methods. For example, for CSP the transformation relation can be a process ordering—there is in fact whole hierarchy of orderings [OH86]. The proof obligations can either be proven using the algebraic laws in CSP (which in the encoding of CSP in Isabelle/HOL are Isabelle theorems), or by using the model-checker FDR. FDR can only be used for finite processes, so it will become more applicable in the later stages of the transformational development, as the specification approaches executability.

For Z, the transformation relation could be model inclusion (i.e. \texttt{SP}_1 \rightsquigarrow \texttt{SP}_2 iff. \texttt{Mod(SP}_1) \supseteq \texttt{Mod(SP}_2)) or refinement of specifications. In both cases, the open question under investigation at the moment is the formulation of appropriate higher-level transformations, which help the user to guide the development process in the right way (an example of such a transformation is the Global Search transformation described in [KSW96]).

### 3 Tool Support

Since one of the main objectives of the \texttt{UniForM} project is to allow the non-expert user to perform at least part of the development themselves, there is a crucial need for tools with a graphical user interface which are easy and intuititve to use.

#### 3.1 System Architecture: Generic and Open

These tools are implemented with a highly generic and open system design, building entirely on well-documented, public domain systems. We will here only briefly sketch the implementation; a more detailed description can be found in [KLMW96a] and [KLMW96b].
The system is entirely implemented in Standard ML (SML) (see Fig. 1), because one can extend Isabelle conservatively by writing ML functions, using the abstract datatypes provided by Isabelle. ML’s typing discipline and structuring mechanisms protect the theorem-proving core of Isabelle from being logically corrupted, and provide a closely coupled and safe (in particular, typed) interaction with Isabelle. Moreover, one can take advantage of SML’s powerful structuring mechanisms to obtain a highly generic and open system architecture.

To implement the graphical user interface, we are using the interface description and command language Tcl/Tk, encapsulated into Standard ML by the sml_tk package (also developed at the University of Bremen). This package provides abstract ML datatypes for the Tcl/Tk objects, thus allowing the programmer to use the interface building library Tk without having to program the control structures of the interface in the untyped, interpretative language Tcl.

Detailed information on sml_tk can be found in [LWW96], or at the sml_tk home page (http://www.informatik.uni-bremen.de/~cxl/sml_tk/).

The generic graphical user interface GenGUI builds on the interface description facilities provided by sml_tk to provide a generic graphical user interface. It is implemented as a functor (a parameterised module)

\[
\text{functor GenGUI}(\text{structure appl: APPL\_SIG}) = ... \]

which provides a graphical user interface for any application described by the signature APPL\_SIG.

A consequence of this approach is that all tools obtained by instantiating GenGUI have a uniform visual appearance. Their main window always consists of two areas: the assembling area in the upper part, and the construction area in the lower part. The assembling area contains icons representing the objects, which can be dragged, moved and dropped onto each other, whereas the construction area allows a more refined manipulation of an object’s internals.

### 3.2 The Transformation Application System TAS

TAS is the system for transformational program development in Isabelle described above. Figure 2 shows a screen shot of TAS. In the upper part of the screen, the assembling area shows a collection of icons, representing transformational program developments, transformation rules and parameter instanti-
The construction area in the lower part of the screen shows the current transformational development.

### 3.3 IsaWin— a Graphical User Interface for Isabelle

IsaWin can be used as an interface to Isabelle in its own right, as well as to prove the proof obligations arising from transformational developments using TAS, or even the correctness of the transformations of TAS.

Figure 2 shows a screen shot of IsaWin. The icons in the assembling area represent theorems, proofs, two types of rule sets, and theories (collections of type declarations, theorems and rule sets). In the construction area, proofs are carried out; once a proof is finished, it can be turned into a theorem. The operations include backward resolution by dropping a theorem onto a proof, forward resolution by dropping a theorem onto a theorem, or rewriting by dropping a rule onto a proof.

### 4 Related and Future Work

The transformational approach to program development has a long tradition, starting from the Munich CIP Project [BBB+85]. During the PROSPECTRA project [HK93], a system has been implemented that enabled the formalisation of transformation rules and their use during the software development process; however, this system was severely hampered by its unstructured design and limited reasoning power, defects which we aimed to remedy by using a powerful prover and a programming language with powerful structuring concepts.
In KIDS [Sm91], programs are developed by transforming problem specifications to programs. First, high-level transformations such as global search are used to transform the problem specification to an inefficient program which is then optimised by low-level transformations. In KIDS, there is no way to check the soundness of the implemented transformations. Here our approach offers a complementary aspect to KIDS since we can prove the correctness of the transformation before applying it, and discharge the resulting proof obligations in the Isabelle system.

The main emphasis during development has been put on a clear and generic system architecture rather than bells and whistles. Having achieved the former, we are going to concentrate on the latter, and are going to implement extensions such as better error handling, pretty printing using mathematical notations and focusing (applying a transformation rule or an Isabelle tactic to a subterm of the current goal, leading to the concept of a generic focus) in the near future. Further, instantiations of our framework for use with CSP and Z are currently being developed.

References


