nijmeegs instituut voor informatica en informatiekunde

Dutch Proof Tools Day 2004

Friday, July 9, 2004
University of Nijmegen
Faculty of Science, Mathematics and Informatics
Building A, Room CZN4

Preface

These are the proceedings of the Dutch Proof Tools Day 2004, organized by the Foundations group of the NIII, University of Nijmegen. The papers contained herein give an account of the contents of the talks presented at the meeting. The Dutch Proof Tools Day is organized on an annual basis by the Protagonist group of Proof Tool users in the Netherlands (and Belgium). The aim of the day is to bring together researchers that use (or develop) proof tools to discuss the use, applicability and background of proof tools. More specifically, it provides a forum for young researchers to get in touch with other proof tools users and exchange experiences.

This is the 8th Proof Tools Day. Previous days were held at Utrecht (twice), CWI (twice), Nijmegen, Eindhoven and Gent (Belgium). More information on previous Proof Tools Days and the motivation behind them can be found at `http://www.cs.uu.nl/~wishnu/protagonist/`. For information regarding the Dutch Proof Tools Day 2004 in Nijmegen, see `http://www.cs.kun.nl/fnds/prooftools/`

We are happy that we have been able to form an interesting program, with interesting contributions from researchers from the Netherlands and Belgium and also two invited speakers: Michael Beeson from San José State University (California) and Jacques Carette from McMaster University (Canada). All this would not have been possible without the financial support from the research school IPA (`http://www.win.tue.nl/ipa/`) and the Foundations group at Nijmegen.

The organization has been a joint effort of (in alphabetical order) Luís Cruz-Filipe, Herman Geuvers, Nicole Messink, Bas Spitters, Dan Synek and Freek Wiedijk. The program was put together by Luís Cruz-Filipe, Herman Geuvers, Bas Spitters and Freek Wiedijk.

A formal reduction for lock-free parallel algorithms

Gao, H. and Hesselink, W.H.

Department of Mathematics and Computing Science, University of Groningen, P.O. Box 800, 9700 AV Groningen, The Netherlands
Email: {hui,wim}@cs.rug.nl

Abstract. On shared memory multiprocessors, synchronization often turns out to be a performance bottleneck and the source of poor fault-tolerance. Lock-free algorithms can do without locking mechanisms, and are therefore desirable. Lock-free algorithms are hard to design correctly, however, even when apparently straightforward. We formalize Herlihy's methodology [13] for transferring a sequential implementation of any data structure into a lock-free synchronization by means of synchronization primitives *Load-linked* (LL)/*store-conditional* (SC). This is done by means of a reduction theorem that enables us to reason about the general lock-free algorithm to be designed on a higher level than the synchronization primitives. The reduction theorem is based on refinement mapping as described by Lamport [10] and has been verified with the higher-order interactive theorem prover PVS. Using the reduction theorem, fewer invariants are required and some invariants are easier to discover and easier to formulate.
The lock-free implementation works quite well for small objects. However, for large objects, the approach is not very attractive as the burden of copying the data can be very heavy. We propose two enhanced lock-free algorithms for large objects in which slower processes don't need to copy the entire object again if their attempts fail. This results in lower copying overhead than in Herlihy's proposal.

Keywords & Phrases: Distributed algorithms, Lock-free, Simulation, Refinement mapping

1 Introduction

On shared-memory multiprocessors, processes coordinate with each other via shared data structures. To ensure the consistency of these concurrent objects, processes need a mechanism for synchronizing their access. In such a system the programmer typically has to explicitly synchronize access to shared data by different processes to ensure correct behaviors of the overall system, using synchronization primitives such as semaphores, monitors, guarded statements, mutex locks, etc. Consequently the operations of different processes on a shared data structure should appear to be serialized: if two operations execute simultaneously, the system guarantees the same result as if one of them is arbitrarily executed before the other.

Due to blocking, the classical synchronization paradigms using locks can incur many problems such as convoying, priority inversion and deadlock. A *lock-free* (also called non-blocking) implementation of a shared object guarantees that within a finite number of steps always some process trying to perform an operation on the object will complete its task, independently of the activity and speed of other processes [13]. As lock-free synchronizations are built without locks, they are immune from the aforementioned problems. In addition, lock-free synchronizations can offer progress guarantees. A number of researchers [1, 4, 5, 13–15] have proposed techniques for designing lock-free implementations. The basis of these techniques is using some synchronization primitives such as *compare-and-swap* (CAS), or *Load-linked* (LL)/*store-conditional* (SC).

Typically, the implementation of the synchronization operations is left to the designer, who has to decide how much of the functionality to implement in software using system libraries. The high-level specification gives lots of freedom about how a result is obtained. It is constructed in some mechanical way that guarantees its correctness and then the required conditions are automatically satisfied [3]. We reason about a high-level specification of a system, with a large grain of atomicity, and hope to deduce an implementation, a low-level specification, which must be fine grained enough to be translated into a computer program that has all important properties of the high-level specification.

However, the correctness properties of an implementation are seldom easy to verify. Our previous work [6] shows that a proof may require unreasonable amounts of effort, time, or skill. We therefore develop a reduction theorem that enables us to reason about a lock-free program to be designed on a higher level than the synchronization primitives. The reduction theorem is based on refinement mappings as described by Lamport [10], which are used to prove that a lower-level specification correctly implements a higher-level one. Using the reduction theorem, fewer invariants are required and some invariants are easier to discover and easier to formulate, without considering the internal structure of the final implementation. In particular, nested loops in the algorithm may be treated as one loop at a time.

2 Lock-free transformation

The machine architecture that we have in mind is based on modern shared-memory multiprocessors that can access a common shared address space. There can be several processes running on a single processor. Let us assume there are P (≥ 1) concurrently executing sequential processes.

Synchronization primitives LL and SC, proposed by Jensen et al. [2], have found widespread acceptance in modern processor architectures (e.g. MIPS II, PowerPC and Alpha architectures). They are a pair of instructions, closely related to the CAS, and together implement an atomic Read/Write cycle. Instruction LL first reads a memory location, say X, and marks it as "reserved" (not "locked"). If no other processor changes the contents of X in between, the subsequent SC operation of the same processor succeeds and modifies the value stored; otherwise it fails. There is also a validate instruction VL, used to check whether X was not modified since the corresponding LL instruction was executed. Implementing VL should be straightforward in an architecture that already supports SC. Note that the implementation does not access or manipulate X other than by means of $LL/SC/VL$. Moir [12] showed that $LL/SC/VL$ can be constructed on any system that supports either LL/SC or CAS. A shared variable X only accessed by $LL/SC/VL$ operations can be regarded as a variable that has an associated shared set of process identifiers $V.X$, which is initially empty. The semantics of LL, VL and SC are given by equivalent atomic statements below.

proc $LL(\texttt{ref}\ X : \texttt{val}) : \texttt{val} =$
$\quad \langle\ V.X := V.X \cup \{self\};\ \texttt{return}\ X;\ \rangle$

proc $VL(\texttt{ref}\ X : \texttt{val}) : \texttt{boolean} =$
$\quad \langle\ \texttt{return}\ (self \in V.X)\ \rangle$

proc $SC(\texttt{ref}\ X : \texttt{val}; \texttt{in}\ Y : \texttt{val}) : \texttt{boolean} =$
$\quad \langle\ \texttt{if}\ self \in V.X\ \texttt{then}\ V.X := \emptyset;\ X := Y;\ \texttt{return}\ \mathit{true}$
$\quad\quad \texttt{else return}\ \mathit{false};\ \texttt{fi}\ \rangle$

where *self* is the process identifier of the acting process.

At the cost of copying an object's data before an operation, Herlihy [13] introduced a general methodology to transfer a sequential implementation of any data structure into a lock-free synchronization by means of synchronization primitives LL and SC. A process that needs access to a shared object pointed by X performs a loop of the following steps:(1) read X using an LL operation to gain access to the object's data area; (2) make a private copy of the indicated version of the object (this action need not be atomic); (3) perform the desired operation on the private copy to make a new version; (4) finally, call a SC operation on X to attempt to swing the pointer from the old version to the new version. The SC operation will fail when some other process has modified X since the LL operation, in which case the process has to repeat these steps until consistency is satisfied. The algorithm is non-blocking because at least one out

of every P attempts must succeed within finite time. Of course, a process might always lose to some faster process, but this is often unlikely in practice.

3 Reduction

We assume a universal set $\mathcal{V}$ of typed variables, which is called the *vocabulary*. A state s is a type-consistent interpretation of $\mathcal{V}$, mapping variables $v \in \mathcal{V}$ to values $s[\![v]\!]$. We denote by Σ the set of all states. If $\mathcal{C}$ is a command, we denote by $\mathcal{C}_p$ the transition $\mathcal{C}$ executed by process p, and $s[\![\mathcal{C}_p]\!]t$ indicates that in state s process p can do a step $\mathcal{C}$ that establishes state t. When discussing the effect of a transition $\mathcal{C}_p$ from state s to state t on a variable v, we abbreviate $s[\![v]\!]$ to v and $t[\![v]\!]$ to v'. We use the abbreviation $Pres(V)$ for $\bigwedge_{v\in V}(v' = v)$ to denote that all variables in the set V are preserved by the transition. Every private variable name can be extended with the suffix "." + "*process identifier*". We sometimes use indentation to eliminate parentheses.

3.1 Observed Specification

In practice, the specification of systems is concerned rather with externally visible behavior than computational feasibility. We assume that all levels of specifications under consideration have the same observable state space Σ_0, and are interpreted by their observation functions $\Pi : \Sigma \to \Sigma_0$. Every specification can be modeled as a five-tuple $(\Sigma, \Pi, \Theta, \mathcal{N}, \mathcal{L})$ where $(\Sigma, \Theta, \mathcal{N})$ is the *transition system* [16] and $\mathcal{L}$ is the supplementary property of the system (i.e., a predicate on Σ^ω).

The supplementary constraint $\mathcal{L}$ is imposed since the transition system only specifies safety requirements and has no kind of fairness conditions or liveness assumptions built into it. Since, in reality, a stuttering step might actually perform modifications to some internal variables in internal states, we do allow stuttering transitions (where the state does not change) and the next-state relation is therefore reflexive. A finite or infinite sequence of states is defined to be an *execution* of system $(\Sigma, \Pi, \Theta, \mathcal{N}, \mathcal{L})$ if it satisfies initial predicate Θ and the next-state relation $\mathcal{N}$ but not necessarily the requirements of the supplementary property $\mathcal{L}$. We define a *behavior* to be an infinite execution that satisfies the supplementary property $\mathcal{L}$. A (concrete) specification $\mathcal{S}_c$ *implements* a (abstract) specification S_a iff every externally visible behavior allowed by $\mathcal{S}_c$ is also allowed by S_a. We write $Beh(\mathcal{S})$ to denote the set of behaviors of system $\mathcal{S}$.

3.2 Refinement mappings

A *refinement mapping* from a lower-level specification $\mathcal{S}_c = (\Sigma_c, \Pi_c, \Theta_c, \mathcal{N}_c, \mathcal{L}_c)$ to a higher-level specification $\mathcal{S}_a = (\Sigma_a, \Pi_a, \Theta_a, \mathcal{N}_a, \mathcal{L}_a)$, written $\phi : \mathcal{S}_c \sqsubseteq \mathcal{S}_a$, is a mapping $\phi : \Sigma_c \to \Sigma_a$ that satisfies:

1. ϕ preserves the externally visible state component: $\Pi_a \circ \phi = \Pi_c$.
2. ϕ is a *simulation*, denoted $\phi : \mathcal{S}_c \preccurlyeq \mathcal{S}_a$:

① ϕ takes initial states into initial states: $\Theta_c \Rightarrow \Theta_a \circ \phi$.
② $\mathcal{N}_c$ is mapped by ϕ into a transition (possibly stuttering) allowed by $\mathcal{N}_a$: $\mathcal{Q} \wedge \mathcal{N}_c \;\Rightarrow\; \mathcal{N}_a \circ \phi$, where $\mathcal{Q}$ is an invariant of $\mathcal{S}_c$.

3. ϕ maps behaviors allowed by $\mathcal{S}_c$ into behaviors that satisfy $\mathcal{S}_a$'s supplementary property: $\forall\, \sigma \in Beh(\mathcal{S}_c) : \mathcal{L}_a(\phi(\sigma))$.

Below we need to exploit the fact that the simulation only quantifies over all reachable states of the lower-level system, not all states. We therefore explicitly allow an invariant $\mathcal{Q}$ in condition 2 ②. The following theorem is stated in [11].

Theorem 1. *If there exists a refinement mapping from $\mathcal{S}_c$ to $\mathcal{S}_a$, then $\mathcal{S}_c$ implements $\mathcal{S}_a$.*

Refinement mappings give us the ability to reduce an implementation by reducing its components in relative isolation, and then gluing the *reductions* together with the same structure as the implementation. Atomicity guarantees that a parallel execution of a program gives the same results as a sequential and non-deterministic execution. This allows us to use the refinement calculus for stepwise refinement of transition systems [8]. Essentially, the reduction theorem allows us to design and verify the program on a higher level of abstraction. The big advantage is that substantial pieces of the concrete program can be dealt with as atomic statements on the higher level.

The refinement relation is transitive, which means that we don't have to reduce the implementation in one step, but can proceed from the implementation to the specification through a series of smaller steps.

3.3 Correctness

The safety properties satisfied by the program are completely determined by the initial predicate and the next-state relation. This is described by Theorem 2, which can be easily verified.

Theorem 2. *Let $\mathcal{P}_c$ and $\mathcal{P}_a$ be safety properties for $\mathcal{S}_c$ and $\mathcal{S}_a$ respectively. The verification of a concrete judgment $(\Sigma_c, \Theta_c, \mathcal{N}_c) \models \mathcal{P}_c$ can be reduced to the verification of an abstract judgment $(\Sigma_a, \Theta_a, \mathcal{N}_a) \models \mathcal{P}_a$, if we can exhibit a simulation ϕ mapping from Σ_c to Σ_a that satisfies $\mathcal{P}_a \circ \phi \Rightarrow \mathcal{P}_c$.*

We make a distinction between safety and liveness properties (See [10] for the proof schemes). The proof of liveness relies on the fairness conditions associated with a specification. The purpose for fairness conditions is to rule out executions where the system idles indefinitely with control at some internal point of a procedure and with some transition of that procedure enabled. Fairness arguments usually depend on safety properties of the system.

4 A lock-free pattern

We propose a pattern that can be universally employed for a lock-free construction in order to synchronize access to a shared node of *nodeType*. The interface

```
CONSTANT
  P = number of processes; N = number of nodes
Shared Variables:
  pub: aType; Node: array [1..N] of nodeType;
Private Variables:
  priv: bType; pc: {a1, a2}; x: 1..N; tm: cType;
Program:
      loop
  a1:   noncrit(pub, priv, tm, x);
  a2:   ⟨ if guard(Node[x], priv) then com(Node[x], priv, tm); fi ⟩
      end
Initial conditions Θa : ∀ p:1..P: pcp = a1
Liveness  La : □ ( pcp=a2 ⟶ ◇ pcp=a1 )
```

Fig. 1. Interface $\mathcal{S}_a$

```
CONSTANT
  P = number of processes; N = number of nodes
Shared Variables:
  pub: aType; node: array [1..N+P] of nodeType;
  indir: array [1..N] of 1..N+P;
Private Variables:
  priv: bType; pc: [c1.. c7];
  x: 1..N; mp, m: 1..N+P; tm, tm1: cType;
Program:
      loop
  c1:   noncrit(pub, priv, tm, x);
        loop
  c2:     m := LL(indir[x]);
  c3:     read(node[mp], node[m]);
  c4:     if guard(node[mp], priv) then
  c5:        com(node[mp], priv, tm1);
  c6:        if SC(indir[x], mp) then
                mp := m; tm := tm1; break;
             fi
  c7:     else
             if VL(indir[x]) then break; fi
          fi
        end
      end
Initial conditions Θc :
  (∀ p:1..P: pcp = c1 ∧ mpp=N+p) ∧ (∀ i:1..N: indir[i]=i)
Liveness  Lc : □ ( pcp=c2 ⟶ ◇ pcp=c1)
```

Fig. 2. Lock-free implementation $\mathcal{S}_c$ of $\mathcal{S}_a$

$\mathcal{S}_a$ is shown in Fig. 1, where the following statements are taken as a schematic representation of segments of code:

1. $noncrit(\texttt{ref}\ pub : aType,\ priv : bType;\ \texttt{in}\ tm : cType;\ \texttt{out}\ x : 1..N)$: representing an atomic non-critical activity on variables pub and $priv$ according to the value of tm, and choosing an index x of a shared node to be accessed.
2. $guard(\texttt{in}\ X : nodeType,\ priv : bType)$ a non-atomic boolean test on the variable X of $nodeType$. It may depend on private variable $priv$.
3. $com(\texttt{ref}\ X : nodeType;\ \texttt{in}\ priv : bType;\ \texttt{out}\ tm : cType)$: a non-atomic action on the variable X of $nodeType$ and private variable tm. It is allowed to inspect private variable $priv$.

The action enclosed by angular brackets $\langle \ldots \rangle$ is defined as atomic. The private variable x is intended only to determine the node under consideration, the private variable tm is intended to hold the result of the critical computation com, if executed. By means of Herlihy's methodology, we give a lock-free implementation $\mathcal{S}_c$ of interface $\mathcal{S}_a$ in Fig. 2. In the implementation, we use some other schematic representations of segments of code, which are described as follows:

4. $read(\texttt{ref}\ X : nodeType,\ \texttt{in}\ Y : nodeType)$: a non-atomic read operation that reads the value from the variable Y of $nodeType$ to the variable X of $nodeType$, and does nothing else. If Y is modified during $read$, the resulting value of X is unspecified but type correct, and no error occurs.
5. LL, SC and VL : atomic actions as we defined before.

Typically, we are not interested in the internal details of these schematic commands but in their behavior with respect to lock-freedom. In $\mathcal{S}_c$, we declare P extra shared nodes for private use (one for each process). Array $indir$ acts as pointers to shared nodes. $node[mp.p]$ can always be taken as a "private" node (other processes can read but not modify the content of the node) of process p though it is declared publicly. If some other process successfully updates a shared node while an active process p is copying the shared node to its "private" node, process p will restart the inner loop, since its private view of the node is not consistent anymore. After the assignment $mp := m$ at line $c6$, the "private" node becomes shared and the node shared previously (which contains the old version) becomes "private".

Formally, we introduce N_c as the relation corresponding to command $noncrit$ on $(aType \times bType \times cType,\ aType \times bType \times 1..N)$, P_g as the predicate computed by $guard$ on $nodeType \times bType$, R_c as the relation corresponding to com on $(nodeType \times bType, nodeType \times cType)$, and define

$$\Sigma_a \triangleq (Node[1..N],\ pub) \times (pc,\ x,\ priv,\ tm)^P,$$
$$\Sigma_c \triangleq (node[1..N{+}P],\ indir[1..N],\ pub) \times (pc,\ x,\ mp,\ m,\ priv,\ tm,\ tm1)^P,$$
$$\Pi_a(\Sigma_a) \triangleq (Node[1..N],\ pub), \quad \Pi_c(\Sigma_c) \triangleq (node[indir[1..N]],\ pub),$$
$$\mathcal{N}_a \triangleq \bigvee_{0 \le i \le 2} \mathcal{N}_{ai}\ , \quad \mathcal{N}_c \triangleq \bigvee_{1 \le i \le 7} \mathcal{N}_{ci}\ ,$$

The transitions of the abstract system can be described: $\forall s, t : \Sigma_a,\ p : 1..P$:

$$
\begin{array}{l}
s[\![(\mathcal{N}_{a0})_p]\!]t \triangleq s = t \qquad \text{(to allow stuttering)} \\
s[\![(\mathcal{N}_{a1})_p]\!]t \triangleq pc.p = a1 \ \wedge pc'.p = a2 \ \wedge Pres(\mathcal{V} - \{pub,\ priv.p,\ pc.p,\ x.p\}) \\
\qquad\qquad \wedge\ ((pub,\ priv.p,\ tm.p), (pub,\ priv.p,\ x.p)') \in N_c \\
s[\![(\mathcal{N}_{a2})_p]\!]t \triangleq\ pc.p = a2\ \wedge\ pc'.p = a1\ \wedge\ (P_g(Node[x],\ priv.p) \\
\qquad\qquad \wedge\ ((Node[x],\ priv.p), (Node[x],\ tm.p)') \in R_c \\
\qquad\qquad \wedge\ Pres(\mathcal{V} - \{pc.p,\ Node[x],\ tm.p\}) \\
\qquad\qquad \vee \neg P_g(Node[x],\ priv.p)\ \wedge\ Pres(\mathcal{V} - \{pc.p\})).
\end{array}
$$

The transitions of the concrete system can be described in the same way. Here we only provide the description of the step that starts in $c6$: $\forall s, t : \Sigma_c,\ p : 1..P$:

$$
\begin{array}{l}
s[\![(\mathcal{N}_{c6})_p]\!]t \triangleq pc.p = c6\ \wedge\ (p \in V.indir[x.p] \\
\qquad\qquad \wedge\ pc'.p = c1\ \wedge\ (indir[x.p])' = mp.p\ \wedge\ mp'.p = m.p \\
\qquad\qquad \wedge\ tm'.p = tm1.p\ \wedge\ (V.indir[x.p])' = \emptyset \\
\qquad\qquad \wedge\ Pres(\mathcal{V} - \{pc.p,\ indir[x.p],\ mp.p,\ tm.p,\ V.indir[x.p]\}) \\
\qquad\quad \vee\ \ p \notin V.indir[x.p]\ \wedge\ pc'.p = c2\ \wedge\ Pres(\mathcal{V} - \{pc.p\}))
\end{array}
$$

4.1 Simulation

According to Theorem 2, the verification of a safety property of concrete system $\mathcal{S}_c$ can be reduced to the verification of the corresponding safety property of abstract system $\mathcal{S}_a$ if we can exhibit the existence of a simulation between them.

Theorem 3. *The concrete system $\mathcal{S}_c$ defined in Fig. 2 is simulated by the abstract system $\mathcal{S}_a$ defined in Fig. 1, that is, $\exists \phi : \mathcal{S}_c \preccurlyeq \mathcal{S}_a$.*

Proof: We prove Theorem 3 by providing a simulation. The simulation function ϕ is defined by showing how each component of the abstract state (i.e. state of Σ_a) is generated from components in the concrete state (i.e. state of Σ_c). We define ϕ : the concrete location $c1$ is mapped to the abstract location $a1$, while all other concrete locations are mapped to $a2$; the concrete shared variable $node[indir[x]]$ is mapped to the abstract shared variable $Node[x]$, and the remaining variables are all mapped to the identity of the variables occurring in the abstract system.

The assertion that the initial condition Θ_c of the concrete system implies the initial condition Θ_a of the abstract system follows easily from the definitions of Θ_c, Θ_a and ϕ.

The central step in the proof of simulation is to prove that every atomic step of the concrete system simulates an atomic step of the abstract system. We therefore need to associate each transition in the concrete system with the transition in the abstract system.

It is easy to see that the concrete transition $\mathcal{N}_{c1}$ simulates $\mathcal{N}_{a1}$ and that $\mathcal{N}_{c2}$, $\mathcal{N}_{c3}$, $\mathcal{N}_{c4}$, $\mathcal{N}_{c5}$, $\mathcal{N}_{c6}$ with precondition "$self \notin V.indir[x.self]$", and $\mathcal{N}_{c7}$ with precondition "$self \notin V.indir[x.self]$" simulate a stuttering step $\mathcal{N}_{a0}$ in the abstract system. E.g., we prove that $\mathcal{N}_{c6}$ executed by any process p with precondition "$p \notin V.indir[x.p]$" simulates a stuttering step in the abstract system. By the mechanism of SC, an active process p will only modify its program counter

$pc.p$ from $c6$ to $c2$ when executing $\mathcal{N}_{c6}$ with precondition "$p \notin V.indir[x.p]$". According to the mapping of ϕ, both concrete locations $c6$ and $c2$ are mapped to abstract location $a2$. Since the mappings of the pre-state and the post-state to the abstract system are identical, $\mathcal{N}_{c6}$ executed by process p with precondition "$p \notin V.indir[x.p]$" simulates the stuttering step $\mathcal{N}_{a0}$ in the abstract system.

The proof for the simulations of the remaining concrete transitions is less obvious. Since simulation applies only to transitions taken from a reachable state, we postulate the following invariants in the concrete system $\mathcal{S}_c$:

$$
\begin{array}{ll}
Q1: & (p \neq q \Rightarrow mp.p \neq mp.q) \ \wedge \ (indir[y] \neq mp.p) \\
 & \quad \wedge \ (y \neq z \Rightarrow indir[y] \neq indir[z]) \\
Q2: & pc.p = c6 \ \wedge \ p \in V.indir[x.p] \\
 & \quad \Rightarrow \ ((node[m.p],\ priv.p),\ (node[mp.p],\ tm1.p)) \in R_c \\
Q3: & pc.p = c7 \ \wedge \ p \in V.indir[x.p] \ \Rightarrow \ \neg\, P_g(node[m.p],\ priv.p) \\
Q4: & pc.p \in [c3..c7] \ \wedge \ p \in V.indir[x.p] \ \Rightarrow \ m.p = indir[x.p] \\
Q5: & pc.p \in \{c4,\ c5\} \ \wedge \ p \in V.indir[x.p] \ \Rightarrow \ node[m.p] = node[mp.p] \\
Q6: & pc.p = \{c5,\ c6\} \ \Rightarrow \ P_g(node[mp.p],\ priv.p)
\end{array}
$$

In the invariants, the free variables p and q range over $1..P$, and the free variables y and z range over $1..N$. Invariant $Q1$ implies that, for any process q, $node[mp.q]$ can be indeed treated as a "private" node of process q since only process q can modify that. Invariant $Q4$ reflect the mechanism of the synchronization primitives LL and SC.

With the help of those invariants above, we have proved that $\mathcal{N}_{c6}$ and $\mathcal{N}_{c7}$ executed by process p with precondition "$p \in V.indir[x.p]$" simulate the abstract step $\mathcal{N}_{a2}$ in the abstract system. For reasons of space we refer the interested reader to [7] for the complete mechanical proof. □

4.2 Refinement

Recall that not all simulation relations are refinement mappings. According to the formalism of the reduction, it is easy to verify that ϕ preserves the externally visible state component. For the refinement relation we also need to prove that the simulation ϕ maps behaviors allowed by S_c into behaviors that satisfy S_a's liveness property, that is, $\forall \sigma \in Beh(\mathcal{S}_c) : \mathcal{L}_a(\phi(\sigma))$. Since ϕ is a simulation, we deduce

$$
\begin{array}{rcl}
\sigma \models \mathcal{L}_c & \equiv & \sigma \models \Box(pc = c2 \longrightarrow \Diamond pc = c1) \\
 & \Rightarrow & \sigma \models \Box(pc \in [c2..c7] \longrightarrow \Diamond pc = c1) \\
 & \Rightarrow & \phi(\sigma) \models \Box(pc = a2 \longrightarrow \Diamond pc = a1) \\
 & \equiv & \mathcal{L}_a(\phi(\sigma))
\end{array}
$$

Consequently, we have our main reduction theorem:

Theorem 4. *The abstract system $\mathcal{S}_a$ defined in Fig. 1 is refined by the concrete system $\mathcal{S}_c$ defined in Fig. 2, that is, $\exists \phi : \mathcal{S}_c \sqsubseteq \mathcal{S}_a$.*

The liveness property $\mathcal{L}_c$ of concrete system $\mathcal{S}_c$ can also be proved under the assumption of the strong fairness conditions and the following assumption:

$$\Box\ (\Box pc.p \in [c2..c7] \wedge \Box\Diamond p \in V.indir[x.p] \\ \longrightarrow\ \Diamond(pc.p = c6 \vee pc.p = c7) \wedge p \in V.indir[x.p]).$$

The additional assumption indicates that for every process p, when process p remains in the loop from $c2$ to $c7$ and executes $c2$ infinitely often, it will eventually succeed in reaching $c6$ or $c7$ with precondition "$p \in V.indir[x.p]$".

5 Large object

To reduce the overhead of failing non-blocking operations, Herlihy [13] proposes an exponential back-off policy to reduce useless parallelism, which is caused by failing attempts. A fundamental problem with Herlihy's methodology is the overhead that results from making complete copies of the entire object ($c3$ in Fig. 2) even if only a small part of an object has been changed. For a large object this may be excessive.

We therefore propose two alternatives given in Fig. 3. For both algorithms the fields of the object are divided into W disjoint logical groups such that if one field is modified then other fields in the same group may be modified simultaneously. We introduce an additional field *ver* in *nodeType* to attach version numbers to each group to avoid unnecessary copying. We assume all version numbers attached to groups are positive. As usual with version numbers, we assume that they can be sufficiently large. We increment the version number of a group each time we modify at least one member in the group.

All schematic representations of segments of code that appear in Fig. 3 are the same as before, except

3. com(`ref` X : *nodeType*; `in` g : 1..W, *priv* : *bType*; `out` *tm* : *cType*) : performs an action on group g of the variable X of *nodeType* instead of on the whole object X.
4. *read*(`ref` X : *nodeType*; `in` Y : *nodeType*, g : 1..W) : only reads the value from group g of node Y to the same group of node X.

The relations corresponding to these schematic commands are adapted accordingly.

In the first implementation, *mp* becomes an array used to record pointers to private copies of shared nodes. In total we declare $N * P$ extra shared nodes for private use (one for each process and each node). Note that $node[mp[x].p]$ can be taken as a "private" node of process p though it is declared publicly. Array *indir* continues to act as pointers to shared nodes.

At the moment that process p reads group $i.p$ of $node[m.p]$ (line $l5$), process p may observe the object in an inconsistent state (i.e. the read value is not the current or historical view of the shared object) since pointer $m.p$ may have been redirected to some private copy of the node by some faster process q, which has increased the modified group's version number(in lines $l9$ and $l10$). When process p restarts the loop, it will get higher version numbers at the array *new*, and only needs to reread the modified groups, whose *new* version numbers differ

```
CONSTANT
  P = number of processes; N = number of nodes;
  W = number of groups;
  K = N + N * P;                        (*  II : K = N + P; *)
Type nodeType = record
    val: array [1..W] of valType;
    ver: array [1..W] of posnat;
  end
Shared Variables:
  pub: aType; node: array [1..K] of nodeType;
  indir: array [1..N] of 1..K;
Private Variables:
  priv: bType; pc: [l1..l11];
  x: 1..N; m: 1..K;
  mp: array [1..N] of 1..K;             (*  II : mp: 1..K;  *)
  new: array [1..W] of posnat; old: array [1..W] of nat;
  g: 1..W; tm, tm1: cType; i: nat;
Program:
      loop
  l1:   noncrit(pub, priv, tm, x);
        choose group g to be modified;
        old := node[mp[x]].ver;          (*  II : old := λ (i:1..W): 0; *)
        (*  II : replace all ‘‘mp[x]’’ below by ‘‘mp’’ *)
        loop
  l2:     m := LL(indir[x]);
  l3:     i := 1
  l4:     while i ≤ W do
            new[i] := node[m].ver[i];
            if new[i] ≠ old[i] then
  l5:         read(node[mp[x]], node[m], i); old[i] := 0;
  l6:         if not VL(indir[x]) then goto l2; fi;
  l7:         node[mp[x]].ver[i] := new[i]; old[i] := new[i];
            fi;
            i++;
          end;
  l8:     if guard(node[mp[x]], priv) then
  l9:        com(node[mp[x]], g, priv, tm1); old[g] := 0;
             node[mp[x]].ver[g] := new[g]+1;
  l10:       if SC(indir[x], mp[x]) then
                mp[x] := m; tm := tm1; break;
             fi
  l11:    elseif VL(indir[x]) then break;
          fi
        end
      end
```

Fig. 3. Lock-free implementation I (* implementation II *) for large objects

from their *old* version numbers. Excessive copying can be therefore prevented. Line $l6$ is used to check if the read value of a group is consistent with the version number.

The first implementation is fast for an application that often changes only a small part of the object. However, the space complexity is substantial because $P+1$ copies of each node are maintained and copied back and forth. Sometimes, a trade-off is chosen between space and time complexity. We therefore adapt it to our second lock-free algorithm for large objects (shown in Fig. 3 also) by substituting all statements enclosed by $(*\ldots*)$ for the corresponding statements in the first version. As we did for small objects, we use only one extra copy of a node for each process in the second implementation.

In the second implementation, since the private copy of a node may belong to some other node, a process first initializes all elements of *old* to be zero (line $l1$) before accessing an object, to force the process to make a complete copy of the entire object for the first attempt. The process then only needs to copy part of the object from the second attempt on. The space complexity for our second version saves $(N-1) \times P$ times of size of a node, while the time complexity is more due to making one extra copy of the entire object for the first attempt. To see why these two algorithms are correct, we refer the interested reader to [7] for the complete mechanical proof.

6 Conclusions

This paper shows an approach to verification of simulation and refinement between a lower-level specification and a higher-level specification. It is motivated by our present project on lock-free garbage collection. Using the reduction theorem, the verification effort for a lock-free algorithm becomes simpler since fewer invariants are required and some invariants are easier to discover and easier to formulate without considering the internal structure of the final implementation. Apart from safety properties, we have also considered the important problem of proving liveness properties using the strong fairness assumption.

A more fundamental problem with Herlihy's methodology is the overhead that results from having multiple processes that simultaneously attempt to update a shared object. Since copying the entire object can be time-consuming, we present two enhanced algorithms that avoid unnecessary copying for large objects in cases where only small part of the objects are modified. It is often better to distribute the contents of a large object over several small objects to allow parallel execution of operations on a large object. However, this requires that the contents of those small objects must be independent of each other.

Formal verification is desirable because there could be subtle bugs as the complexity of algorithms increases. To ensure our hand-written proof presented in the paper is not flawed, we use the higher-order interactive theorem prover PVS for mechanical support. PVS has a convenient specification language and contains a proof checker which allows users to construct proofs interactively, to

automatically execute trivial proofs, and to check these proofs mechanically. For the complete mechanical proof, we refer the reader to [7].

References

1. B. Bershad: Practical Considerations for Non-Blocking Concurrent Objects. In Proceedings of the 13th International Conference on Distributed Computing Systems, May 1993.
2. E.H. Jensen, G.W. Hagensen, and J.M. Broughton: A new approach to exclusive data access in shared memory multiprocessors. Technical Report UCRL-97663, Lawrence Livemore National Laboratory, November 1987.
3. E. Clarke, O. Grumberg, and D. Long: Model checking and abstraction ACM Transactions on Programming Languages and Systems 16(5), January 1994.
4. G. Barnes: A method for implementing lock-free data structures. In Proceedings of the 5th ACM symposium on Parallel Algorithms & Architecture, June 1993.
5. Henry Massalin, Calton Pu: A Lock-free Multiprocessor OS Kernel. Technical Report CUCS-005-91, Columbia University, 1991
6. H. Gao, J.F. Groote and W.H. Hesselink.: Efficient almost wait-free parallel accessible dynamic hashtables. Technical Report CS-Report 03-03, Eindhoven University of Technology, The Netherlands, 2003. To appear in the proceedings of IPDPS 2004.
7. http://www.cs.rug.nl/~wim/mechver/LLSCreduction
8. J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors: Stepwise Refinement of Distributed Systems: Models, Formalism, Correctness. Lecture Notes in Computer Science 430. Spinger-Verlag, 1990.
9. Anthony LaMarca: A Performance Evaluation of Lock-free Synchronization Protocols. In proceedings of the thirteenth symposium on priniciples of distributed computing, 1994.
10. L. Lamport: The Temporal Logic of Actions. ACM Transactions on Programming Languages and Systems 16(3), 1994, pp. 872–923.
11. M. Abadi and L. Lamport: The existence of refinement mappings. Theoretical Computer Science, 2(82), 1991, pp. 253–284.
12. Mark Moir: Practical Implementations of Non-Blocking Synchronization primitives. In Proceedings of the sixteenth symposium on principles of Distributed computing, 1997. Santa Barbara, CA.
13. M. P. Herlihy: A methodology for implementing highly concurrent objects. ACM Transactions on Programming Languages and Systems 15, 1993, pp. 745–770.
14. Maurice Herlihy, Victor Luchangco and Mark Moir: The Repeat Offender Problem: A Mechanism for Supporting Dynamic-Sized, Lock-Free Data Structures. In Proceedings of the 16th International Symposium on DIStributed Computing, 2002.
15. Victor Luchangco, Mark Moir, Nir Shavit: Nonblocking k-compare-single-swap. In Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms, 2003, pp. 314-323.
16. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer-Verlag, 1992.

Integrating Cadence SMV in the Verification of UML Software

S. Van Langenhove and A. Hoogewijs

Department of Pure Mathematics and Computer Algebra
Ghent University, Belgium

{Sara.VanLangenhove, Albert.Hoogewijs}@UGent.be

Abstract

The Unified Modeling Language has been widely used in software development. Verifying that a UML model meets the required properties has become a key issue. The verification gives us the opportunity to remove errors, inconsistencies, and misconceptions in the very early stages of software development. Model checking is an important technology of automatic formal verification to ensure the correctness of design specifications. The discussion mainly focuses on the use of Cadence SMV in the design of software systems in UML. In this paper, we present a translation template for UML state charts and for UML protocol conformance into CSMV.

1 Introduction

Modeling and validation, whether formal or ad-hoc, are important steps in system design. Over the last couple of decades, various methods and tools were developed for decreasing the amount of design and development errors. A common component of such methods and tools is the use of formalisms, like the Unified Modeling Language (UML) [10], for specifying the behavior and the requirements of the system. Experience has shown that some of the formalisms, such as UML state charts, are particularly appealing, due to their convenient mathematical properties. In the design phase, state charts are used for the increasingly successful automatic verification, also called model checking [8]. One of the biggest challenges is developing a UML based verification method, based on state charts, to identify and remove errors, inconsistencies, and misconceptions during the design phase and before entering the code phase.

A translation template in the Cadence SMV (CSMV) [7] modeling tool for the state charts is presented. This template allows us to convert arbitrary state charts to the modal logic of CSMV, covering all behavioral model elements as defined in the UML specification like hierarchy, sequentialism, parallelism, non-determinism, priorities, and run-to-completion step semantics. The CSMV model checker is subsequently used to verify the behavioral design. Verifying requirements between communicating objects of different classes using this template is also possible. Communicating objects have their behavior specified in separate state charts. The communication is based on signal and call event exchange, also known as asynchronous and synchronous communication respectively, and their queuing. Additionally, the first step towards the protocol conformance between behavioral state charts and protocol state charts has easily been realized.

The template is built using some of the strengths of the Cadence SMV language. Unfortunately, some technical limitations of CSMV and the UML semantics raise some difficulties. E.g., queuing of events is less trivial as it may seem. The limitations also force us to transform the original state charts into semantic equivalents. More specifically, insertion of additional states and transitions is necessary to cope with these limitations. As a consequence, a number of secondary problems shows up. E.g. we are forced to perform transformations, using temporal logic, on the system requirements, as specified by the designer. To be able to verify the protocol conformance, protocol state charts have to be transformed too, according to a limited number of rules.

In this survey we describe the template structure for a single state chart and for communicating state charts. We define the meaning of protocol conformance and show how the conformance can easily be proved using CSMV. Some bottlenecks during the construction of the templates and the proof of the protocol conformance is discussed. Finally, we give some conclusions and outline future work.

2 Related Work

Recently, there have been some studies in model checking UML state charts. Unfortunately, the proposed methods are far from complete. [3] translates a single UML state chart to the input language of the model checker SPIN. The authors do not believe in model checking multiple communicating state charts due to an imprecise semantics. The actions on transitions are restricted to the generation of only one single output event. [5] transforms multiple state charts to the input language of SPIN as well. The proposed tool is able to model check both single and communicating state charts. The corresponding semantics – which is quite complete – assumes that each state chart runs in a different process. As a consequence the scheduling of the state charts is interleaved. To the best of my knowledge, they avoid the case where multiple objects share the same execution thread[1]. Moreover, call events, used for a synchronous communication, are treated wrongly like any other event. Verifying state chart diagrams is also realized by using other model checkers like the Symbolic Model Verifier SMV ([9, 1]). But here, the run-to-completion semantics is not really followed.

3 Cadence SMV Description

In Cadence SMV, a system is modeled as a state machine. The properties to be verified are given as temporal formulae. The state of the system is determined by the values of a collection of state variables. Available types include booleans, integer ranges, arrays, vectors as well as user defined types, etc. The set of the initial states and the transition relation between the states are defined by assignments to the initial and the next state values of the state variables. Assignments are treated as a system of simultaneous equations meaning that the order in which assignments appear in the program is irrelevant. Invariants and fairness conditions may be added as boolean formulae.

Each system in CSMV is structured into a number of modules. Each module can be instantiated multiple times and may contain instances of other modules. CSMV allows for both synchronous and asynchronous execution of the different modules. The modules communicate with each other using global variables. CSMV has an internal variable for each process/module (called running) that is set equal to true when a transition from that process executes.

In contrast with other versions of CSMV, Cadence SMV provides several additional compositional methods. The methods allow the verification of large, complex systems by reducing the verification problem to smaller problems that can be solved automatically by model checking. Examples of such features include induction, data type reduction, refinement verification, etc.

4 Template structure

UML features state charts based on the widely recognized state chart notation [2]. As an extension of traditional finite automata (see definition 1) by adding hierarchy (= state refinement to contain another automaton), concurrency and communication, UML state charts (see definition 2) are powerful and flexible kinds of state transition diagrams. They give an abstract view of the desired behaviors of an object in its life cycle. This makes an early model checking validation of properties crucial for the design process, since the verification tends to become more expensive during and after the code generation.

Definition 1 *A finite transition system labeled by an alphabet A is a 5-tuple $\mathcal{A} = \langle S, T, \alpha, \beta, \lambda \rangle$ where S is a finite set of states, T is a finite set of transitions, α and β are two mappings from T to S which take each transition t in T to the two states $\alpha(t)$ and $\beta(t)$, respectively the source and the target of the transition t, and where λ is a mapping from T to A taking each transition t to its label $\lambda(t)$.*

Definition 2 *A UML state chart is a 6-tuple $\mathcal{SM} = \langle S, T, E, C, AC, i \rangle$ where S is a finite set of states, E is a finite set of events, C is finite set of conditions, AC is a finite set of actions, $T \subseteq S \times E \times C \times 2^{AC} \times S$ is a finite set of transitions where 2^{AC} denotes the power set of A, and where i is the initial state. (The definition can be extended with mappings to the sources, targets, and labels of the transitions.)*

[1] Here, the *thread of control* represents an abstract notion of control and not an operating system thread.

Model checking, whether in the early design stages or not, of the system requirements, requires the construction of transition relations capturing all the information depicted in the state charts.

In the sequel, we give a general overview of the structure of the relations. It will become clear that most work is required for the analysis of the semantics. Less work is required for the implementation of the transition relation.

4.1 A single state chart

4.1.1 Preliminaries

Model checking a single state chart requires a model whose transition relation follows the state chart execution semantics, also called the run-to-completion step semantics, as close as possible. The execution of a state chart can be observed in terms of *events* accepted and *actions* executed (potentially overlapping). Events received are captured as triggers associated with transitions of the state charts. The spectrum of actions is rather huge and therefore we assume that actions only generate new events or modify attribute values.

Basically, events are dispatched and processed by the state machine one at a time. The processing of a single event by a state machine is known as a *run-to-completion step*. A run-to-completion step (RTC-step) is a sequence of transitions $t_1, t_2, \ldots, t_n$ between two stable configurations, i.e. the set of active states the state machine currently resides in. Intuitively this means that once an event has been dispatched, the system evolves on its own until no more (trigger less) transitions can be taken. Then a new event must be dispatched. An event can only be taken from the pool and dispatched if the processing of the previous event is fully completed. Without affecting the execution semantics, the general structure of a run-to-completion step can be seen as a combination of two closely related phases:

1. The first phase checks whether an instable state configuration evolves on its own. This means that automatic transitions that may fire are identified and executed. Automatic transitions are transitions without a trigger.

2. The second phase checks whether a stable configuration can evolve by dispatching an event from the environment. As like the first phase, transitions that may fire are identified and executed.

Once an event is dispatched, it may result in one or more transitions being enabled for firing. And some of them can be in conflict. This happens when the set of states left by the transitions is not empty. The conflicts are solved using priorities. We call this the *conflict resolution strategy*. A transition has a higher priority if its source state is a substate of the source of the other one. If the conflict cannot be solved using priorities, they may fire non-deterministically. Note that there can be at most one transition being enabled for firing in sequential hierarchical states.

To construct the template we confine ourselves to the following requirement:

Requirement 1 *Avoid flattening the state chart and preserve hierarchy and concurrency in states and their semantics.*

4.1.2 Projection of hierarchy and concurrency into the transition relation

To preserve both hierarchy and concurrency in the transition relation, the state chart is first transformed into an extended hierarchical automaton (EHA) [3, 4]. Hierarchical automata are composed of simple sequential automata related by a *refinement function*. A state is mapped via the refinement function into the set of (parallel) automata that refine it.

Definition 3 *A sequential automaton A is a 4-tuple $(\sigma_A, s^0_A, \lambda_A, \delta_A)$ where σ_A is a finite set of states, s^0_A is the initial state, λ_A is a finite set of labels, and $\delta_A \subseteq \sigma_A \times \lambda_A \times \sigma_A$ is the transition relation.*

Definition 4 *A EHA is a 5-tuple (F, E, ρ, A_0, V), where F is the set of sequential automata with mutually disjoint sets of states, E is a finite set of events, and V is the set of variables. $\rho : \cup_{A \in F} \sigma_A \rightarrow 2^F$ is a refine function, which imposes a tree that satisfies: 1) there exists a unique root automaton $A_0 \in F$, and there exists no state $s \in \cup_{A \in F} \sigma_A$ that $A_0 \in \rho(s)$; 2) every non-root automaton has exactly one ancestor state: $\forall A \in F \ \{A_0\}, \exists_1 s \in \cup_{A' \in F \ \{A\}} \sigma_{A'}, A \in \rho(s)$; 3) there are no cycles: $\forall s \notin \rho^*(s)$.*

Representing each node – a simple sequential automaton – of the tree by a CSMV enumerated variable preserves easily both the hierarchy and the concurrency of the state chart.

```
-- s1, s2, ... sn : direct substates of a sequential composite state
st_seq_state : {s1, s2, ... , sn, [NotActive]}
```

This way, flattening the state chart is avoided, since this sometimes may cause an exponential blow-up of the state space.

4.1.3 Projection of the RTC-step into the transition relation

The power of the branch statement is exploited to define the correct execution order between both phases of the RTC-step in the transition relation. For example, the second branch in the code below is only taken when the state machine is in a stable configuration – represented by $progress\_trigger$. As long as the state chart is in some intermediate and inconsistent situation, represented by $progress\_auto$, the first branch will be taken. Obviously, each branch causes the current active state to change to the transition destination states. They possibly may change attribute values and/or putting new events into the environment. Additionally, the state space is augmented with macro like variables, and the transitions of the state chart are divided into two groups. One group contains all the trigger less transitions while the other group consists of the transitions with a trigger.

```
case {
  progress_auto & ~error : {
     -- phase 1 of run-to-completion
     ...
  };
  progress_trigger & ~error : {
     -- phase 2 of run-to-completion
     ...
  };
  error : {
     -- stuttering due to an error, go to an error state
     ...
  };
  default : {
     -- stuttering due to the fact that no progress can be made
     ...
  };
};
```

To acquire a total transition relation, important for the model checking algorithm [8], the structure is augmented with a stutter rule. This way, finite runs are interpreted as special cases of infinite runs. The stutter rule prevents the model to behave incorrectly at the moment a (in)stable configuration cannot evolve anymore. A configuration may stutter due to two reasons:

1. At the moment an error has occurred, the state configuration is forced to stutter in a so-called *exception configuration*.
2. Due to the lack of progress, the state configuration remains the same. This happens when no transitions can be identified to fire.

4.1.4 Projection of the conflict resolution strategy into the transition relation

The conflict resolution strategy is only needed to define the state transformations. I.e., it is needed to define the state changes of each state variable. Each state variable represents a sequential automaton. Therefore, at any time, there can be at most one transition being enabled for firing. The following code illustrates this clearly enough.

```
choose := { seq_ti ? seq_Ti, seq_tj ? seq_Tj, ... , seq_tm ? seq_Tm }
```

```
next(st_seq_state) := case {
       -- exactly one transition can be identified to fire
                        seq_t1 : target_t1;
                        seq_t2 : target_t2;
                        ...
       -- choose non-deterministically the transition to may fire
                        choose = seq_Ti & seq_ti : target_ti;
                        choose = seq_Tj & seq_tj : target_tj;
       -- a transition higher in the hierarchy can be fired
       -- only if no lower transitions  can be fired
                        seq_thigherk & ~seq_tlowerk : NotActive;
                        ...
       -- no progress in the region but active concurrent state
                        in_conc_state : st_seq_state;
                        default: st_seq_state;
                      }
```

The *choose* variable is used to denote the transition that possibly may fire at the same time. The enabledness conditions are used to build the set of transitions from which a choice is made in a non-deterministic way. Since each concurrent region of a state is represented by a state variable and since the *next* statements are executed simultaneously, real parallellism is achieved for a concurrent state. Later, we explain the way events are handled.

4.2 Active state charts

4.2.1 Preliminaries

The basic elements for modeling concurrent systems are the active objects. An active object maintains its own thread of control and runs concurrently with other active objects based on interleaved execution semantics. Basically, this means that the run-to-completion steps of the different state charts are never executed at exactly the same time.

At any time, the whole system can be regarded as one object by simply modeling the concurrent objects as the concurrent regions of one state chart. Current day, systems are so complex that flat and unstructured state machine descriptions of them will be huge and difficult to analyze. Also, the flattening unnecessarily complicates the behaviors of these relative independent objects due to the semantics about the concurrent states. Thus, to augment the template the following requirement is preserved.

Requirement 2 *Preserve the structure of the multi-object software system. I.e. avoid modeling the concurrent objects of the system simply as concurrent regions of a single state in one state chart.*

Multiple state charts interact with the outside world and communicate with each other. UML provides two types of events to set up a communication between state charts: signals and calls. Signals are events representing an asynchronous communication while call events represent a synchronous communication between state charts. A call is used whenever an object sends out a message and passes the control to the receiver. The receiver changes its state as a result of the message and returns the control to the sender. The caller then waits for the callee before continuing its execution. If a message possibly transfers the control from a sender object to a receiver object without returning the control, then it is a signal. In this case, the sender does not have to wait for the receiver before continuing its execution.

In this section, we assume that there is only one state chart running in each thread.

4.2.2 Projection of asynchronously communicating objects into the transition relation

The asynchronous communication between state charts does not affect the operational semantics of the individual state charts. This is motivated by two reasons. Firstly, an interleaved model of computation is used and secondly, it does not matter when the receiver consumes a signal event. As a consequence, to handle asynchronously communicating state charts easily in CSMV, each thread is interpreted as a module. Moreover, each module covers the transition relation for a single active state chart as defined in section 4.1. To guarantee the interleaved execution, each module is instantiated as a process.

4.2.3 Projection of synchronously communicating objects into the transition relation

The synchronous communication changes a limited part of the run-to-completion step semantics. Since the caller's run-to-completion step is interrupted immediately after sending out a call event, it seems obvious to give a higher priority to calls than to signals. I.e., first an operation call is dispatched, and if this is not possible, then we try to dispatch a signal event. By augmenting the structure defined in section 4.2 with one branch, the second phase of the run-to-completion step semantics is very easily updated.

```
case {
  progress_auto & ~error : {
     -- phase 1 of run-to-completion
     ...
  };
  progress_call & ~error : {
     -- phase 2a of run-to-completion
     ...
  };
  progress_signal & ~error : {
     -- phase 2b of run-to-completion
     ...
  };
  error : {
     -- stuttering due to an error, go to an error state
     ...
  };
  default : {
     -- stuttering due to the fact that no progress can be made
     ...
  };
};
```

At the moment a call event is sent to the callee's state chart, the run-to-completion step of the caller is interrupted or blocked and the corresponding thread falls asleep. In CSMV, block variables are used to denote active and sleeping threads. Additionally, the block variables help us to restrict the running variable of the corresponding modules. The restriction is defined easily using an invariant:

```
INVAR
  thread.running -> (thread.block | activate)
```

When all the processes get stuck, we are obliged to activate their stutter rules to acquire a total transition relation at any time.

4.3 Passive state charts

4.3.1 Preliminaries

Active objects are application objects that own a thread of control. They have controller capabilities. Objects not having their own thread of control are passive objects. By default, subobjects (from a composition) share the thread of their parent object. Therefore, it is not strange that the system under development has several objects (with corresponding state charts) present in a single thread[2]. Nothing in the UML standard prohibits the communication between the state charts of these objects. Obviously, the passive state charts do not run continuously. The run-to-completion steps will be activated at the moment an event is dispatched to a particular object.

4.3.2 Projection into the transition relation

Each CSMV module covers now the transition relation of all the state charts. The main obstacle is to define the transition relation in such a way that the run-to-completion step of one state chart is correctly finished before entering the run-to-completion step of another state chart in the same thread.

[2]It is out of the scope of the current document to define the set of objects running in one thread.

```
last, object_progress: {Object1, ..., ObjectK, NotDef};
init(last) := ObjectX;

object_progress :=
  case {
    last = Object1 & object1_progress_auto: Object1;
                ...
    last = ObjectK & objectK_progress_auto: ObjectK;
    ~(object1_progress_auto) & ... & ~(objectK_progress_auto): NotDef;
    default: { object1_progress_auto ? Object1 ... objectK_progress_auto ? ObjectK};
  };

case {
  object_progress = Object1 & ~error : {
     -- phase 1 of run-to-completion of Object1
     ...
     next(last) := Object1;
  };
  ...
  object_progress = ObjectK & ~error : {
     -- phase 1 of run-to-completion of ObjectK
     ...
     next(last) := ObjectK;
  };
  ...
  object1_progress_call & ~error : {
     -- phase 2a of run-to-completion of Object1
     ...
     next(last) := Object1;
  };
  objectK_progress_call & ~error : {
     -- phase 2a of run-to-completion of ObjectK
     ...
     next(last) := ObjectK;
  };
  object1_progress_signal & ~error : {
     -- phase 2b of run-to-completion of Object1
     ...
     next(last) := Object1;
  };
  ...
  objectK_progress_signal & ~error : {
     -- phase 2b of run-to-completion of ObjectK
     ...
     next(last) := ObjectK;
  };
  error : {
     -- stuttering due to an error, go to an error state
     ...
  };
  default : {
     -- stuttering due to the fact that no progress can be made
     ...
  };
};
```

Two additional variables are used in the template structure to guarantee that the run-to-completion step of a particular object is finished before executing the RTC-step of another object.

1. An additional variable *last* to indicate the last object to which an event is served by the dispatcher.
2. An additional macro like variable *object_progress* that indicates which state chart has to be

evaluated until a stable state configuration is reached. Its value is set to that particular object, with an instable configuration, to which an event was last dispatched. If the last object has a stable configuration then its value is set non-deterministically only if other objects became instable due to the execution of transitions in the last evaluated state chart. If all objects have reached a stable configuration, the dispatcher has to be called once again.

5 Proving Protocol Conformance in CSMV

The upcoming new version of the UML, UML 2.0 [11], introduces *Protocol State Machines (PSMs)* and *Behavioral State Machines (BHMs)*. PSMs express the legal transitions that a classifier can trigger precluding any specific behavioral implementation. BHMs specify behavior of various model elements. They are in fact the already known state charts.

5.1 Definitions

Protocol State Machine In its simplest form a PSM (see definition 5) is a state diagram in standard UML notation whose transitions are triggered by events (call event, signal event, time event, completion event) and do not have actions. They present the possible and permitted transitions of an object by specifying the order in which the events can be consumed and the states through which an object progresses during its life. This way, a PSM captures the *triggering view* of an objects behavior. Otherwise stated, it specifies all the capabilities of a class. Each protocol transition specifies that the associated event can be consumed by an instance in the origin state under the initial condition (pre), and that at the end of the transition, the destination state will be reached under the final condition (post).

Definition 5 *A UML protocol state machine is a 6-tuple* $\mathcal{PSM} = \langle S, T, E, PREC, POSTC, i\rangle$ *where* S *is a finite set of states,* E *is a finite set of events,* $PREC$ *is a finite set of preconditions,* $POSTC$ *is a finite set of postconditions,* $T \subseteq S \times PREC \times E \times POSTC \times S$ *is a finite set of transitions, and where* i *is the initial state. (The definition can be extended with mappings to the sources, targets, and labels of the transitions.)*

Behavioral State Machine BHMs express the behavior of part of a system like ordinary state charts do. The behavioral view of an object is not independent of its triggering view. There are two principal rules to respect during the modeling of the behavior in relation to an existing PSM:

1. The set of states that an object may have during its life are fully defined in its PSM.
2. A behavior transition from state S_1 to state S_2, with the label $event[guard]/actions$ is legal, iff the corresponding PSM defines a protocol transition from state S_1 to state S_2 with the label $[pre]event/[post]$. This means that a behavior transition may exists iff there exists a protocol transition with the same source, target and triggering event.

Note that it is not acquired that every transition in the PSM has some counterpart in the BHM since a PSM defines *what a class can do*. The same is true for the states defined in the PSM.

Protocol Conformance UML explicitly considers protocol conformance. Every rule and constraint specified by the protocol machine (state invariants, pre and post conditions, triggering event) must apply to the behavioral machine. Clearly, the definition of protocol conformance is rather fuzzy; it is not very clear under which circumstances protocol conformance may be declared. Therefore, we give a more formal definition of protocol conformance between a PSM and a BHM (adapted from [6]).

Definition 6 *Let P be a PSM and let B be BHM defined for a class c. B conforms to P with respect to a given initial state iff whenever*

$$s \rightarrow^* s' \overset{event[guard]/actions}{\rightarrow} s''$$

(that is, a transition is triggered from some state in B that is reachable from the initial state of B) we have a corresponding counterpart transition in P

$$s \rightarrow^* s' \overset{[pre]event/[post]}{\rightarrow} s''$$

where both the pre and the post condition evaluate to true.

5.2 The proof

Exploiting the refinement verification provided by CSMV very easily proves the protocol conformance between the two kinds of state machines. Refinement verification is a methodology of verifying that the functionality of an abstract system model (i.e. PSM) is correctly implemented by a low-level implementation (i.e. BHM). The new construct *layer* is introduced for this purpose. The *layer* declaration is in fact a high-level specification, which states that every implementation behavior must be consistent with all the given assignments. The obvious way to prove the protocol conformance, is to define the transition relation of the PSM inside the layer, and to define the transition relation of the BHM outside the layer.

Definition 7 *A behavioral EHA, is an EHA for a behavioral state machine. A protocol EHA, is an EHA for a protocol state machine.*

Note that we do not have to change the definition of an EHA to build a protocol EHA.

Definition 8 *Let P be a PSM and let B be a BHM defined for a class c. Let PE be a protocol EHA and let BE be a behavioral EHA. B conforms to P iff every node in BE conforms to its counterpart node in PE.*

Thus, instead of using a single layer, definition 8 allows us to use several layers – one for each node – to automatically proof the protocol conformance. It is clear that this is much more efficient than using a single layer. The main obstacle to face is the specification of the triggering view for a single node inside a layer such that definition 6 is correctly used in the verification process. Informally, this definition says that the execution of a behavioral transition leads to the execution of some counterpart protocol transition. Thus, small parts of the behavioral RTC-step must be inserted in the layer. Having done so, the conformance is automatically checked by CSMV without any intervention.

```
-- high-level implementation, triggering view
layer node_i_spec: {
  case {
    progress_auto: {
       -- if b_t1 (behavioral transition) is enabled
       -- and all the constraints of p_t1 (protocol transition) are satisfied,
       -- then the behavioral state machine must reach the target-state
       next(node_i) := case {
                         b_t1 & p_t1 : target_p_t1;
                         ...
                         default: node_i;
                       };
    };
    ...
    default : { next(node_i) := node_i; };
  };
}
-- low-level implementation, behavioral view
case {
    progress_auto: {
        next(node_i) := case {
                         b_t1 : target_b_t1;
                         ...
                         default: node_i;
                       };
    };
    ...
    default : { next(node_i) := node_i; };
  };
```

6 Bottlenecks

The CSMV language has several rules for assignments, which sometimes complicate the construction of the transition relation. The rules are important to guarantee that every program is executable. The *single assignment rule* precludes that in each step, every signal is assigned only once. This way, the problem of conflicting definitions is avoided. It is only allowed to assign a value to x or to *next(x)* and *init(x)*, but not both. The circular dependency rule says that combinational loops are illegal. A combinational loop is a cycle of dependencies whose total delay is zero. It is the single assignment rule that is sometime annoying during the construction of the transition relation.

Beside the single assignment rule, one particular part of the UML semantics complicates the construction of the transition relation as well. At the moment the execution of a transition sends out a call event, the run-to-completion step is interrupted and blocked, although the destination state of the transition is not yet reached. Requirement 3 gives us enough information to represent interrupted transitions in the transition relation. An example is given in figure 1.

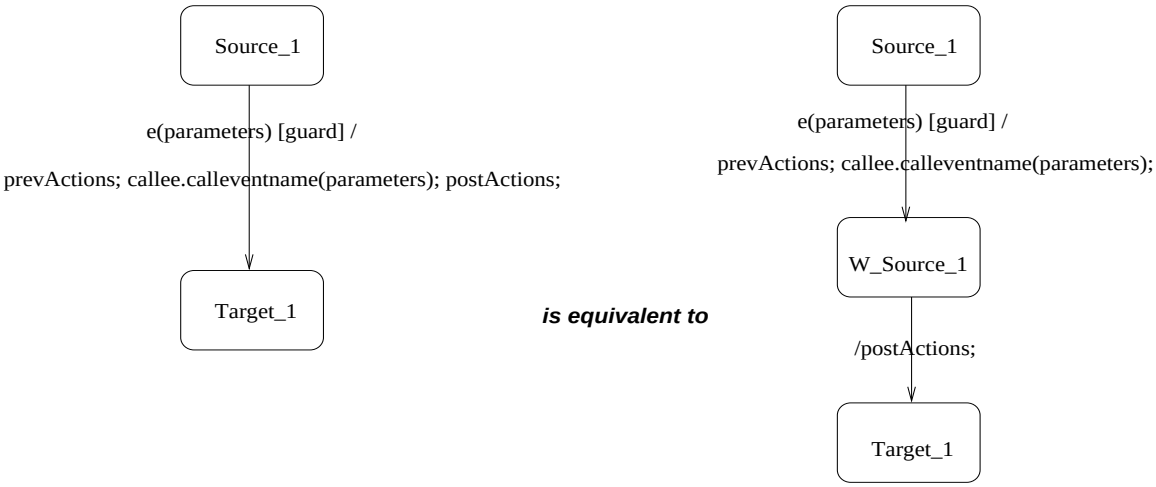


Figure 1: Introduction of additional states

Requirement 3 *Introduce additional states (and of course transitions) to ensure that a run-to-completion always ends in a state and never in the middle of transition as otherwise may happen with synchronous calls.*

6.1 Queuing of events

A run-to-completion step starts by dispatching an event. The role of the dispatcher is to find an event that can be accepted in the current stable configuration of the object. In general, more than one event can be available in the environment. Therefore, they are stored in a *First in First Out* (FIFO) queue. Events generated by actions during an RTC-step cannot be served immediately and therefore they are kept in the queue for later processing. The queue not only stores the events but also forms the basis for the dispatching operation since the dispatcher works on this queue.

How to integrate the queuing of events in the transition relation? The FIFO queue used by the dispatcher is represented as an array with a specific length. The type of the queue is an enumeration of all the events the state chart may react on. An additional *NotDefined* value is used to indicate empty array positions.

```
event_queue : array 0 .. (SIZE - 1) of {ev_1, ... ev_n, NotDefined};
tail : 0 .. (SIZE - 1); -- pointer to the first available position in the queue
```

Assigning an array reference with a variable index counts as assigning every element in the array, as far as the single assignment rule is concerned. The most obvious construction to add elements to the queue with respect to the FIFO order, as illustrated in the code below, is not allowed. This is because CSMV cannot determine at compile time that the two places of the queue are indeed different.

```
case {
  -- when t1 is executed, two new events are generated in a strict order
  t1 : { next(event_queue[tail]) := ev_k;
         next(event_queue[tail + 1]) := ev_l;
       };
  ...
  default: next(event_queue) := event_queue;
};
```

Updating the queue correctly is therefore trickier. We have to make sure that CSMV can determine at compile time that at any time, different array places are filled in. To denote different places we can simply use a loop. The wrong code can be changed into the following:

```
for(i = 0; i < (SIZE); i = i + 1) {
  case {
    -- when t1 is executed, two new events are generated in a strict order
    t1 & tail = i: { next(event_queue[i]) := ev_k;
                     next(event_queue[i + 1]) := ev_l;
                   };
    ...
    default: next(event_queue[i]) := event_queue[i];
  };
};
```

Again, the changed construction to add new elements is not allowed since it is still possible that there are two assignments to the same place of the queue. Thus, it is important that each branch statement inside the loop, updates exactly one assignment and not two like in the code above. The best way to update the queue by adding new events is done in the following manner.

```
for(i = 0; i < (SIZE); i = i + 1) {
  case {
    -- when t1 is executed, two new events are generated in a strict order
    t1 & tail = i: next(event_queue[i]) := ev_k;
    t1 & (tail + 1) = i:  next(event_queue[i + 1]) := ev_l;
    -- when t2 (ev_2) and t3 (ev_3) of different regions execute
    t2 & t3 & tail = i: next(event_queue[i]) := {ev_2, ev_3};
    t2 & t3 & (tail + 1) = i: next(event_queue[i]) :=
                                case {
                                  i > 0 & next(event_queue[i-1]) = ev_2 : ev_3;
                                  default: ev_2;
                                };
    ...
    default: next(event_queue[i]) := event_queue[i];
  };
};
```

The code illustrates also the case of parallel executing transitions. Due to the interleaved execution semantics, these transitions – each belonging to a different region of a concurrent state – can be fired in any order. If they both generate events, the resulting queues will be different. This explains the use of non-deterministic assignments when needed.

6.2 Additional states and the RTC-step

The problem Requirement 3 introduces additional states and transitions. This way, a behavioral specification arises which is different from the original behavioral specification. At any time, both behavioral specifications must be equivalent (see definition 9). Thus, even though there are possibly more states and transitions, both specifications must produce exactly the same results, i.e. exactly the same trace of event delivery order.

Definition 9 *Two behavioral specifications are equivalent if they have the same semantics, meaning that exactly the same information can be derived from their behavior.*

Obviously, the equivalence is automatically guaranteed for a state chart that does not have any concurrent state or with no additional states in the concurrent regions. This is motivated by the fact that the sequential execution of statement stays remained. However, if additional states are inserted into some concurrent regions, the required equivalence is not automatically guaranteed. We will illustrate the problem using figure 2. Here, one region is augmented with an additional state (and also an additional transition). Suppose that in the original state chart transition *T1* and transition *T2* execute concurrently

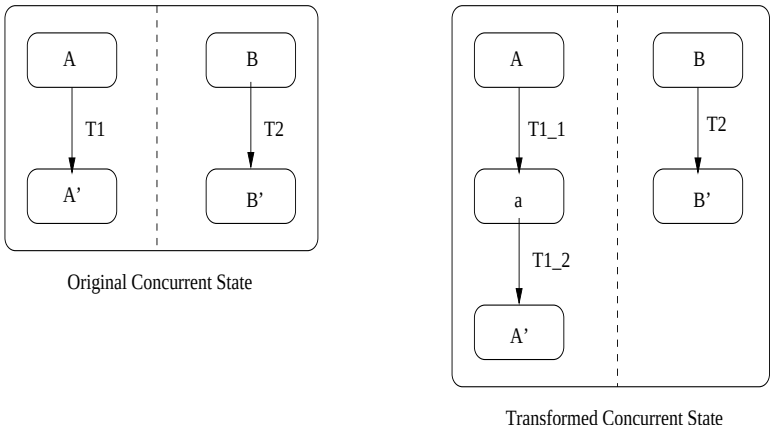


Figure 2: An original and a transformed concurrent state

in the same run-to-completion step. Following the run-to-completion semantics, the active configuration *AB* evolves into configuration *A'B'* while dispatching a single event. In the transformed state chart, the same requirement must hold. Here also, configuration *A'B'* is reached while dispatching a single event. What is the bottleneck? The current structure of the templates is such that in each time step during model checking, real parallelism is achieved in the case concurrent states belongs to an active configuration. If we keep this way of working, transition *T1_1* and transition *T2* execute concurrently and transition *T1_2* might be executed with some other transition leaving the state *B'*. Obviously, if this happens, both the execution order and the event delivery order of both state charts will be different. As a result, the equivalence between both behavioral views is not acquired. To avoid this, we are obliged to interrupt the run-to-completion steps.

The solution Orthogonal regions represent independent states that may concurrently be active with other states. Since they run independently of each other, race conditions (e.g. two regions changing the value of the same variable, one region using the value of a variable that another one is changing) may not occur. This property gives us everything we need to correctly implement the interrupt. If a concurrent region contains states not present in the corresponding concurrent region of the original state chart, then a private queue is attached to the region. At the moment we detect – using a lookahead mechanism – that an active concurrent configuration evolves to a concurrent configuration containing one or more newly introduced states, the interrupt is called. This interrupt has several responsibilities. Firstly, it is capable to achieve real parallelism in the same manner the operational semantics achieves it. In general the following steps are taken:

1. Achieving real parallelism between those transitions leaving states from the active concurrent configuration. This results in reaching a configuration containing newly introduced states in some of the regions. In our example the interrupt executes transition *T1_1* and transition *T2* in parallel.

2. Achieving real parallelism between those transitions leaving newly introduced states. In our example transition *T1_1* will be taken by the interrupt. This step can be repeated several times.

Secondly, although real parallelism is achieved, the event delivery order will be respected since the interrupt is obliged to update only the private queues of the regions. The general queues used to implement the run-to completion semantics are not affected! Only the first step of the interrupt is allowed to make an exception. The first step will remove the event from the corresponding general queue. Last, a lookahead mechanism is again used to detect whether a real configuration (containing

only states present in the original state chart) can be reached. If so, the private queues are merged (several orders) to update the global queues. Now, the interrupt has finished working and a new event can be dispatched.

6.3 Additional states and temporal formulae

To express requirements that are not globally valid (in all states), but temporally or from a certain point onwards, temporal logic (based on a successor relationship on states). CSMV verifies these requirements, through symbolic model checking. When this process gets stuck, CSMV constructs a counter model, which is a path through the model. The counter model is an infinite sequence of states, i.e. one or more states are accessed infinitely often. Due to the introduction of additional states, the verification process has to be guided carefully.

The problem The proposed structure of the transition relation denotes an ordering on the states that allows us to link the proper requirements to the corresponding state(s). Transitions in UML state charts are atomic and cannot be interrupted. Due to requirement 3, it is possible that a transition is broken up into additional transitions and states through the translation to a CSMV model. As a consequence, requirements are verified in these additional states too. This can result in a wrong verification process. To avoid confusion in the formulation and the verification of requirements, we need to transform the temporal formulae so that they cope with possible additional states. This way, additional states are transparent during the verification process. If e.g. the designer wants to know something of a next state, we assume that he means a next state in his behavioral model since he does not know that anything about the additional states. In figure 3 the next state of state $S1$ is state $S5$ and not the additional state D.

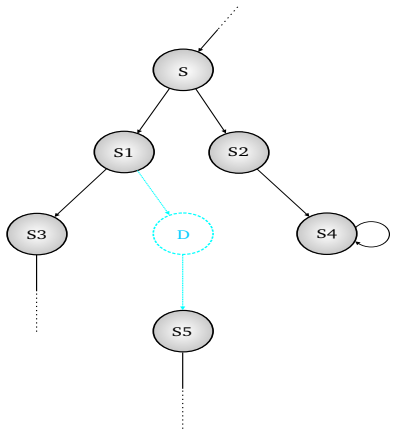


Figure 3: Execution trace with additional states

The solution The most obvious solution is to use a *clock operator* to make the additional states transparent for the verification process. The clock operator allows applying a formula only to states satisfying some condition. Unfortunately, only the *ForSpec* language from Intel and the *Accelera PSL* language (a.k.a., Sugar 2.0) both have such a construct. Since CSMV does not support this, the formulae have to be translated in such a way that the same result is achieved as with the clock operator. In fact, we change the interpretation of the formulae. Obviously, the designer does not see the translations. For that matter, counterexample traces has to be processed as well to eliminate the additional states. In doing so we make this template preserve the behavioral semantics of the objects. Lets take a look to some transformations.

$EF\varphi$ transformation $EF\varphi$ states that it is possible (by following a suitable execution) to have φ some day. If we keep this interpretation, it is allowed that φ holds in an additional state of some execution

path. To avoid this, we make sure that φ is not verified in such additional states. Therefore we are allowed to define the following equivalence:

$$EF\varphi \equiv EF(\varphi \wedge \neg additional)$$

$EG\varphi$ transformation $EG\varphi$ states that there exists an execution along which φ always holds, i.e. in every state of the execution. Additional states represent not completed transitions. Since in UML it is forbidden to interrupt a transition, it is also forbidden to verify φ in the intermediate states because φ can be true or false in these states. The equivalence

$$EG\varphi \equiv EG(\varphi \vee additional)$$

makes sure that an intermediate state is correctly skipped. The equivalence says that everything holds in the additional states along the execution.

$EX\varphi$ transformation $EX\varphi$ states that from the current state, it is possible in one step to reach a state satisfying φ. Transformation of this formula is a little more tricky.

$$EX\varphi \equiv EX(E(additional\ U\ (\neg additional \wedge \varphi)))$$

The property would be true only if from the current state CSMV visits zero or more additional states until CSMV reaches a non-additional state in which φ holds. The equivalence is best understood by using the fixed-point definition of the until operator.

$$E[\phi\ U\ \psi] \equiv \psi \vee (\phi \wedge EXE[\phi\ U\ \psi])$$

Using this equivalence, we first verify if the next state is an non-additional state in which φ holds. If this is not the case, the next state can be an additional state for which the process is repeated. If the next state is not an additional state, then the next state of another execution is examined.

6.4 Additional states and the protocol conformance

Introducing additional states (transitions) in a BHM leads to introducing additional states (transitions) in the corresponding PSM. This is inevitable for proving the protocol conformance. Fortunately, it is done in a slightly more efficient way. In contrast with behavioral transitions, exploiting the *state list* feature can reduce the total amount of additional protocol states and protocol transitions. This is not possible for the behavioral transitions since we have to respect the execution semantics. An example of such a transformation is given in figure 4.

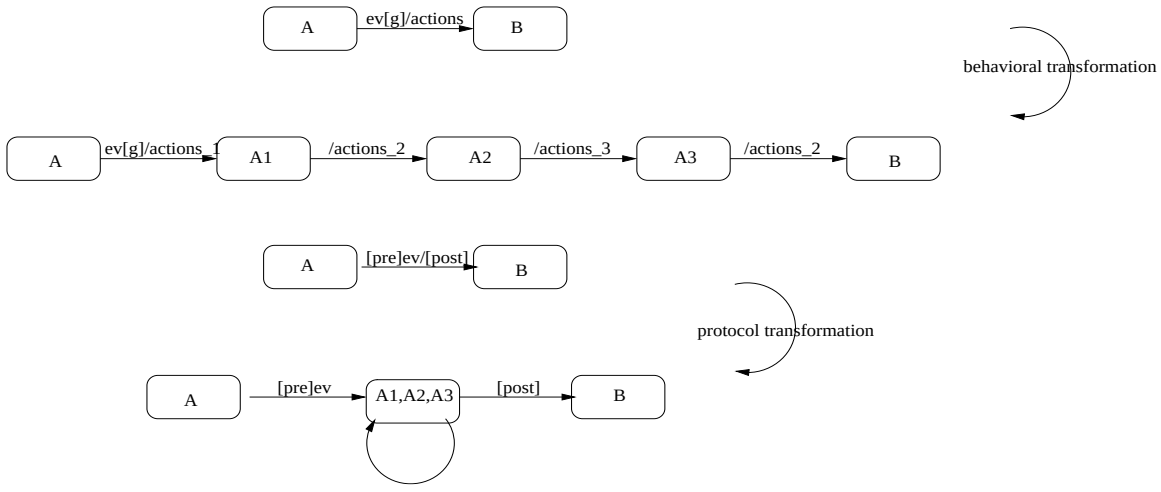


Figure 4: Behavioral/Protocol Transformations

7 Conclusion and future work

Using the strengths of the Cadence SMV language and after solving the bottlenecks, the template makes way for the automatic translation of UML state charts. In particular, we will use the standardized translation of UML in XML to fill out the parameters of our template. With such an automatic translation available, UML not only becomes a standard for system development, but also portal to formal verification. Hence, UML is tuned into a more powerful and interesting tool, particularly for engineers and programmers.

Because of the characteristics such as concurrency and hierarchy, verification of state charts still faces the state explosion problem in model checking. To deal with this problem, a few researchers attempt to reduce the state space of model checking with the method of program slicing. Currently, we try to generalize and to optimize the slicing algorithm presented in [12]. Also, we will use the slicing algorithm to reduce the state space during the protocol conformance proof.

Acknowledgments

The research activities that have been described in this paper are funded by Ghent University (BOF/GOA project B/03667/01 IV1) and the Prof. Dr. Wuytack Fund.

References

[1] Tanuan M. C. Automated Analysis of Unified Modeling Language (UML) Specifications. Master's thesis, University of Waterloo, 2001.

[2] Harel D. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[3] Latella D., Majzik I., and Massink M. Automatic Verification of a behavioural subset of UML Statechart Diagrams using the SPIN Model-checker. *Formal Aspects of Computing*, 11(6):637–664, 1999.

[4] Wei D., Ji W., Xuan Q., and Zh-Chang Qi. Model Checking UML Statecharts. *Eighth Asia-Pacific Software Engineering Conference (APSEC'01)*.

[5] Lilius J. and Paltor P. I. vUML: A Tool for Verifying UML Models. *Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, 1999. 0-7695-0415-9.

[6] Tenzer J. and Stevens P. Modelling recursive calls with uml state diagrams. *Proceedings of Fundamental Approaches to Software Engineering, LNCS*, 2621, 2003.

[7] McMillan K. `http://www-cad.eecs.berkeley.edu/~kenmcmil/smv`.

[8] Clarke E. M., Grumberg O., and Peled D. A. *Model Checking*. The MIT Press, 2002. 0-262-03270-8.

[9] Clarke E. M. and Heinle W. Modular Translation of Statecharts to SMV. Master's thesis, Carnegie Mellon University, 2000.

[10] OMG. `www.omg.org`.

[11] U2 Partners. Unified Modeling Language 2.0 proposal. `http://www.u2-partners.org`.

[12] Ji W., Wei D., and Zhi-Chang Q. Slicing Hierarchical Automata for Model Checking UML Statecharts. In George C. and Miao H., editors, *Formal Methods and Software Engineering, 4th International Conference on Formal Engineering Methods, ICFEM 2002 Shanghai, China, October 21-25, 2002, Proceedings*, volume 2495 of *Lecture Notes in Computer Science*, pages 435 – 446. Springer, 2002.

Theorem proving as a tool in a programmer's toolkit

Arthur van Leeuwen
Universiteit Utrecht

June 21, 2004

Abstract

These days a good number of proof systems to choose from exists. However, incorporating proof technology into other systems is hard, as the existing proof systems are normally not designed for that purpose. This article looks at proof technology as a component of other systems. This point of view leads to several interesting research questions, as well as an interesting software engineering challenge.

1 Introduction

Proof technology has progressed considerably, from its humble beginnings up to the applications to industrial problems we see nowadays. Many tools that incorporate proof technology have been developed in the meantime. Almost all of these tools are designed with a particular goal in mind, such as implementing a particular logic. This focus has resulted in lower priorities for other concerns, such as the software engineering aspects of the implementation.

Fortunately, the engineering aspects have not been completely lost sight of, as work by Boulton [6], Hutter [12] and Harrison [9] shows. There is another software engineering aspect as well, that of using the technology originally developed for proof tools as components of other software. A well known example is the metalanguage of the LCF prover, ML, which has been further developed independently of the theorem prover itself [14, 18]. Another, somewhat more narrowly applicable example can be found in BDD libraries [13, 17].

More recently, Dennis et al. developed the Prosper toolkit, which allows the use of a HOL-like proof engine as a component in other software [8, 7]. However, not much more research has been done into further reuse of technology originally developed for proving in other software.

The present article will therefore delve into using technology developed for proof tools as a component of other software. To that end we will examine the components of current proof tools and the technology contained therein. Given that base we will look into what components may be reused and what the prerequisites for such reuse are.

2 Proof tool components

In this article we will look at proof technology in general, thereby ignoring most differences between theorem provers and model checkers. The reason for this approach is that the technologies used in theorem provers and model checkers only partially differ and many of the components overlap. As the components themselves are of interest us, the differences are mostly irrelevant.

All proof tools consist of several components. Most important among these is ofcourse the mechanization of the formal system the proof tool builds upon. Another component is the user interface. Furthermore, many proof tools come with more or less extensive libraries of theorems and/or models. Finally, there are proof supporting components, such as decision procedures.

The mechanization of the formal system at the base of a proof tool is often the most interesting component. It consists of a mechanization of the abstract syntax of the formal system, as well as access methods to build and inspect objects in that abstract syntax, according to the rules of the formal system. Furthermore, it may contain methods that perform transformations on the abstract syntax, encoding the rules of the formal system.

Such a mechanization is useful in many situations. Every application of reasoning can benefit from it. Examples are inferencing databases [4], software agents, and also adaptive hypermedia systems [5]. The challenge is in formalizing the terms and inference rules used in such an application. Logical frameworks such as Isabelle [16] and MetaPRL [10] show that this can be done, efficiently. They are rarely used for these purposes, however.

On top of the mechanization of the formal system a proof tool has to provide some user interface. This user interface may be as spartan as a batch interface, accepting a file with a formulation of a concrete theorem or model as its input and writing a file with the results of checking the theorem or model as its output. It may also be as luxurious as a graphical user interface allowing full graphical or mathematical notation and direct interaction. In all cases however, there is a concrete syntax in which the input is specified and the output displayed.

Methods of dealing with concrete syntax are well known, and known to be useful. These components of proof tools are normally built analogous to similar components of compilers. Software engineers can therefore easily look to the field of compilers if they want to make use of such technology.

Of greater interest is the library of work that has been done in the formal system. Many proof tools come with rather extensive libraries of mathematical results formulated and proven in the formal system the proof tool is based upon. It is these libraries that turn the proof tool from a theoretically useful tool into a practical one, as almost all practical applications depend on having more than just a logic to model with.

There is a correlation between the popularity of a proof tool and the size of its library. This correlation works in two directions. First, a popular proof tool has more users and these build more libraries. Second, a proof tool with an extensive library is more attractive, and attracts even more users. One of the great advantages of for example the Mizar system [3] is its extensive library of formalized mathematics. However, these libraries are normally tied to a particular proof tool. While normally possible, it is usually difficult, tedious work to port a library from one proof tool to another.

Last but not least there are the proof supporting components such as decision

procedures or implementations of unification algorithms. These help the user of the proof tool to get to the desired results. It is these that provide much of the reason for the existence of the diversity in proof tools, as one formal system may well be more suited to a particular method of supporting finding a proof than another.

3 Technology

The different components of a proof tool all have their own technological issues associated with them. We will shortly highlight a number of these.

The mechanization usually consists of an embedded language used to build terms and other objects. These may be represented internally as abstract syntax trees, but they may just as well be represented as more complex datastructures, such as tries [11]. In some systems, such as HOL [1], this embedded language is directly accessible, whereas in others it is only used as the backend of the parser for the concrete syntax in which the input is specified.

Reusing a mechanization of a formal system can be done by taking the embedded language and all supporting code of a proof tool and using that as a component of a different system. If the embedded language is directly accessible this is much easier than if it is not. Furthermore, proof tools that have the term language exposed usually also have it documented, thereby facilitating reuse even further. In the case of HOL this is exemplified by the PROSPER toolkit. Its core proof engine is virtually identical to the core of HOL.

The advantage to lifting the mechanization out of the proof tool and reusing it in a different program is that it is the most direct way to include a formal system. However, in doing so one may lose the connection between the mechanized system in the proof tool and that in the new program. This implies that any development of the mechanization in the proof tool, such as bugfixes to it, needs to be lifted seperately as well. Furthermore, it invites divergent development, which may make reuse of libraries of already formalized knowledge harder.

Reusing libraries of formalized knowledge is hard enough in itself, without further complication. Not only may proof tools wildly vary in the logics they accept, the concrete syntaxes those logics take can also differ. A glaring example is in the HOL family of systems. The syntaxes of HOL, HOL Light [2] and Isabelle/HOL are different, even if the logic is the same. Therefore, libraries of math formalized in one do not carry over directly to the other. Another problem in handling libraries is that the scripts representing an existing formalization may well break on a new version of the exact same proof tool, due to modifications in the proof supporting components.

As hard as reusing libraries is, there are still possibilities of doing it. As long as the logic used for the library is comparable to or easily embeddable in the logic used in a new system, porting the library could conceivably be automated [15]. One way of doing this is in expanding all automated search used in the scripts of the library to proof steps that can be directly embedded. Another is in exporting complete proof objects at the level of the logic itself, and then importing these. These approaches are complex, but not impossible. The situation is worse if the logics are not similar. How to automate translating results stated in one logic to statements in another is unknown, if the logics are sufficiently dissimilar.

The final component to look into is the proof support. Much of this is in implementation of e.g. search algorithms. These may well be suited to other tasks than proof support. As long as the implementations of such algorithms are made generic enough it is easy to apply them to other problem domains, in other programs. However, as higher order generic programming is not very common, most implementations are tied to the particular data structures used in a proof tool.

Beyond the algorithms used to support e.g. proof search, proof tools may also provide systems for dealing with stacks of proof goals, libraries of theorems, etc. As these are geared to supporting ease of use of the proof tool, reuse of these is not normally relevant; a system using proof technology may not even need user-interaction.

Another approach to reuse is to take the entire proof tool as a component. This entails running the proof tool as a seperate program under control of the other system, and communicating with it through for example its standard input and output. The drawback of this approach is that it needs a method of dealing with the concrete syntax of the formal system within the main system, as there is no way to directly access the abstract objects inside the proof tool. This also implies that many advantages of the internal representation within the proof tool are lost to the other system. However, as the entire proof tool is available, some of these drawbacks may be mitigated by the proof supporting components of the proof tool.

Using the entire proof tool as a component solves other issues as well. As the tool is not broken up into its constituent components the library of results already obtained using the tool can still be directly used. The same holds for all proof support that is present in and for the tool. Furthermore, once the infrastructure for dealing with one proof tool as a component has been set up, it is fairly straightforward to add other proof tools as components as well. This is the approach PROSPER took, resulting in a library that encapsulated said infrastructure.

4 Conclusions

As is clear, proof technology is a fertile area for software engineering. There are many opportunities for reusing components of proof tools, either in other proof tools, or in tools that make use of properties that can be formally proven. However, the existing proof tools are not normally designed with such reuse in mind.

It is still possible to benefit from the technology developed for proof tools in other software. This reuse is non-trivial, however, at every level that one looks at. The problems that occur are mostly of a software-engineering nature, but some of them are more fundamental.

This non-triviality points to an interesting research field. Among others, the PROSPER project has gone some way into that field, but much is still left to be explored. Most notably research into the reuse of libraries of formalized knowledge may lead to highly useful results.

References

[1] Hol 4 kananaskis 2. http://hol.sourceforge.net/.

[2] Hol light – a small and idealistic, yet fairly powerful, theorem prover. http://www.cl.cam.ac.uk/users/jrh/hol-light.

[3] The mizar project. http://mizar.org/.

[4] The PARKA project. http://www.cs.umd.edu/projects/plus/Parka/.

[5] Matteo Baldoni, Cristina Baroglio, and Viviana Patti. Applying logic inference techniques for gaining flexibility and adaptivity in tutoring systems. In *HCII*, 2003.

[6] Richard Boulton. *Efficiency in a Fully-Expansive Theorem Prover*. PhD thesis, University of Cambridge, 1993.

[7] Graham Collins and Louise A. Dennis. System description: Embedding verification into microsoft excel. In *Conference on Automated Deduction*, pages 497–501, 2000.

[8] Louise Dennis, Graham Collins, Michael Norrish, Richard Boulton, Konrad Slind, Graham Robinson, Mike Gordon, and Tom Melham. The prosper toolkit. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 78–92, 2000.

[9] John Harrison. Hol done right. Unpublished draft, 1995. http://www.cl.cam.ac.uk/users/jrh/papers/holright.html.

[10] Jason Hickey, Aleksey Nogin, Robert L. Constable, Brian E. Aydemir, Ely Barzilay, Yegor Bryukhov, Richard Eaton, Adam Granicz, Alexei Kopylov, Christoph Kreitz, Vladimir N. Krupski, Lori Lorigo, Stephan Schmitt, Carl Witty, and Xin Yu. Metaprl – a modular logical environment. In *Theorem Proving in Higher Order Logics*, 2003.

[11] Thomas Hillenbrand. CITIUS ALTIUS FORTIUS: lessons learned from the theorem prover WALDMEISTER. In *First-order Theorem Proving*, 2003.

[12] Dieter Hutter. Deduction as an engineering science. In *Proceedings of the International Workshop on First-Order Theorem Proving, 2003*, Electronic Notes in Theoretical Computer Science, vol. 86, no. 1, 2003.

[13] David E. Long. The design of a cache-friendly bdd library. In *Proceedings of the 1998 International Conference on Computer-Aided Design (ICCAD'98)*, 1998.

[14] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML - Revised*. The MIT Press, May 1997.

[15] Pavel Naumov, Mark-Oliver Stehr, and Jose Meseguer. The hol/nuprl proof translator – a practical approach to formal interoperability, 2001.

[16] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[17] John Whaley. Javabdd – high-performance java binary decision diagram library, 2003. http://javabdd.sourceforge.net/.

[18] Xavier Leroy (with Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon). The objective caml system, release 3.07, Sep 2003.

C-CoRN, the Constructive Coq Repository at Nijmegen

Luís Cruz-Filipe[1,2], Herman Geuvers[1], and Freek Wiedijk[1]

[1] NIII, Radboud University of Nijmegen
[2] Center for Logic and Computation, Lisboa
lcf|herman|freek@cs.kun.nl

Abstract. We present C-CoRN, the Constructive Coq Repository at Nijmegen. It consists of a library of constructive algebra and analysis, formalized in the theorem prover Coq. In this paper we explain the structure, the contents and the use of the library. Moreover we discuss the motivation and the (possible) applications of such a library.

1 Introduction

A repository of formalized constructive mathematics [5] in the proof assistant Coq [12] has been constructed over the last five years at the University of Nijmegen. This is part of a larger goal to design a computer system in which a mathematician can do mathematics.

At this moment we don't have a complete idea of what such a system should look like and how it should be made; through foundational and experimental research we want to contribute ideas and results that support the development of such a system in the future.

One of the things that is very important for such a 'mathematical assistant' to be used is the availability of a large and usable library of basic results. But such a library shouldn't just be a huge collection of proved results (including the 'proof scripts' that are input for the proof assistant to carry out the proof). In our view, a library of formalized mathematics should be:

Accessible: one should be able to get a fairly fast overview of what's in it and where to find specific results;
Readable: once one has come down to the basic objects like definitions, lemmas and proofs, these should be presented in a reasonable way;
Coherent: results about a specific theory should be grouped together and theories extending others should be defined as such;
Extensible: it should be possible to include contributions from other researchers.

How can one make such a (large) coherent library of formalized mathematics? Ideally, this should also be independent of the Proof Assistant one is working with, but right now we don't know how to do that. Several other projects deal with this question. The Mowgli project [1] aims at devising system independent tools for presenting mathematics on the web. The OpenMath [9] and OMDoc

[24] standards aim at exchanging mathematics across different mathematical applications, which is also one of the aims of the Calculemus project [7]. This may eventually lead to ways of sharing mathematical libraries in a semantically meaningful way that preserves correctness, but that's not possible yet (an exception is NuPRL which can use HOL results [28].)

So, to experiment with creating, presenting and using such a library, one has to stick to one specific theorem prover, and already there many issues come up and possible solutions can be tested. We have chosen to use the Coq proof assistant, because we already were familiar with it and because we were specifically interested in formalizing constructive mathematics.

This paper first describes the backgrounds of C-CoRN: its history and motivation. Then we describe the structure of the repository as it is now and the methodology that we have chosen to develop it. Finally we discuss some applications and future developments.

2 History

The C-CoRN repository grew out of the FTA-project, where a constructive proof of the Fundamental Theorem of Algebra was formalized in Coq. This theorem states that every non-constant polynomial f over the complex numbers has a root, i.e., there is a complex number z such that $f(z) = 0$.

One of the main motivations for starting the FTA-project was to create a library for basic constructive algebra and analysis, to be used by others. Often, a formalization is only used by the person that created it (or is not used further at all!), whereas an important added value of formalizing mathematics – in our view – is to create a joint computer based repository of mathematics. For the FTA-project, this meant that we didn't attempt to prove the theorem as fast as possible, but that in the proving process we tried to formalize the relevant notions at an appropriate level of abstraction, so they could be reused.

An important condition for the successful use of a library of formalized mathematics is to have good documentation of the code. There are two main purposes of documentation:

1. to show to the world what has been formalized via a 'high level' presentation of the work (in our case that would be a LaTeX document giving a mathematical description of the formalized theory);
2. to help the interested outsider to extend (or change or improve or vary on) the formalized theory.

For (1) one wants to produce a LaTeX document that 'goes along' with the formalization. This may be generated from the formalization (but it is not quite clear whether it is at all possible to generate something reasonably, and mathematically abstract from the very low level formal proof code). Alternatively – and this is the approach followed in the FTA-project –, this LaTeX file may be created in advance and then used as a reference for the proof to formalize. The goal of the FTA-project was to formalize an *existing* proof and not to redo the

mathematics or 'tailor' the mathematics toward the proof assistant. This meant that the LaTeX document we started from was an original constructive proof of FTA, described in [21], with lots of details filled in to ease the formalization process. The same approach has been followed throughout the rest of C-CoRN: existing mathematical proofs and theories were formalized, so the (high level) mathematical content corresponds to an existing part of a book or article.

For (2), some simple scripts were created in the FTA-project to be able to extract from the Coq input files a useful documentation for outsiders interested in the technical content. However, this was pretty *ad hoc* and not very satisfactory, and it was changed in C-CoRN, as described in Section 5.

After the FTA-project was finished, i.e., after the theorem had been formally proved in Coq, it was not yet clear that it had been successful in actually creating a usable library, because all people working with the library until then were part of the project. The only way to test this would be by letting outsiders extend the library. This is not too easy: due to the fact that we have tactics implemented in ML (e.g. to do equational reasoning), one cannot use the standard image of Coq and has to build a custom image first. Therefore, the first real test only came when the first author of this paper started as a new Ph.D. student to formalize constructive calculus (leading to the Fundamental Theorem of Calculus) in Coq. The FTA library turned out to be very usable. Most importantly, there was almost no need to restructure the library or redefine notions, implying that most of the basic choices that were made in the FTA-project worked. (Of course, the basic library was extended a lot, with new results and new definitions.) Hereafter, the library was re-baptized to C-CoRN, the Constructive Coq Repository at Nijmegen, since the FTA and the work of the FTA-project had become only a (small) part of the formalized mathematics.

Since then, several people, both working in Nijmegen and outside, have consulted, used and contributed to C-CoRN. These have as a rule found its structure (including notations, automation facilities, documentation) quite useful.

3 Why C-CoRN?

Formalizing mathematics can be fun. In the process of formalizing, one discovers the fine structure of the field one is working with and one gains confidence in the correctness of the definitions and the proofs. In addition to this, formalizing mathematics can also be useful. We indicate some of its possible uses:

Correctness guaranteed: the formalized mathematics is checked and therefore the proofs are guaranteed to be correct for all practical purposes. This can be vital in the realm of software or system correctness, where one wants to be absolutely sure that the mathematical models and the results proved about them are correct;

Exchange of 'meaningful' mathematics: that the mathematics is formalized means that it has a structure and a semantics within the Proof Assistant. So a mathematical formula or proof is not just a string of symbols, but it has a structure that represents the mathematical meaning and its building

blocks have a definition (within the Proof Assistant). These can in principle be exploited to generate meaningful documents or to exchange mathematics with other applications;

Finding mathematical results: based on the semantics and the structure of the formalized mathematics, it should be possible to find results easier. Querying based on the (meaningful) structure is already possible, but more semantical querying would be welcome.

The potential uses of formalized mathematics only really become available if one can share the formalization and let others profit from it, e.g. by making it possible for them to study it, extend it or use it for their own applications or further development. A key requirement for this is that the formalized mathematics be presented. Ordinary (non-computer-formalized) mathematical results are published in articles, books or lecture notes and are in that way shared with other mathematicians or users of mathematics. Giving a good presentation of formalized mathematics in practice, though, turns out to be quite hard. There are various reasons for this:

Idiosyncrasies of the Proof Assistant: when talking to a Proof Assistant, things have to be stated in a specific way, so the system understands it: definitions have to be given in a specific format, proofs have a specific form, etc. Moreover, each Assistant has its own logical foundations (e.g. set theory, type theory or higher order logic), making it easy to express some concepts (e.g. inductive definitions in type theory) and hard to express others (e.g. subsets in type theory). Because of this, mathematical theory will be defined in a specific way that best fits the idiosyncrasies of the system at hand. When presenting the formal mathematics, one would like to abstract from these 'arbitrary' choices;

Verbosity of formalized mathematics: to make the Proof Assistant understand (or be able to verify) what the user means, a lot of details have to be given. By itself, this is unavoidable (and fine, because we really want the mathematics to be verified and that doesn't come for free). But in the presentation phase, one wants to abstract from these finer details and 'view' the mathematics at a higher level. This is not so easy to achieve;

Access to the formalized mathematics: how does one find a result in the library, how does one get an overview of the material? One can query the library with syntactic means, searching a formula of a specific form, as described in [22]. This is helpful, but can a useful result still be found if it occurs in the library in disguise? Also, if a user knows that a specific lemma appears in the library, she will want to apply it, which in a Proof Assistant is done by using its name. But what is the name of (the proof of) a lemma? One probably doesn't want to clutter up the names in the presentation of the math, so 'logical' naming conventions come in handy.

To be able to really understand and work on these points, one needs a 'testbed' to experiment with. This was one of the reasons to start C-CoRN: to have a library of mathematics that people can really contribute to, read and work with.

Of course, such libraries already exist. The prominent one is the Mizar Mathematical Library, MML. However, Mizar was not good for experimenting with a library of formalized mathematics, because we wouldn't have control over the library: one can't just install Mizar and formalize a large library on top of the existing one. Soon the system gets slow and one has to submit the work to the Mizar library to have it integrated. Moreover, the Mizar system itself is not open in the sense that one can program one's own tools (e.g. for automation or documentation) on top of it[3]. Another important reason not to choose Mizar was that we were aiming at a library of *constructive* mathematics. For all these reasons, Coq was a good choice, with the drawback that there was only a very small initial library to begin with. (There are many 'Coq contributions', but they do not form one coherent library as discussed in Section 8.)

The main reason for working constructively is that we want to use our library, apart from studying how it conducts as a repository, to study the connections between 'conceptual' (abstract) mathematics and 'computational' (concrete) mathematics. The first (conceptual math) deals with e.g. the proof of (and theory development leading to) the Fundamental Theorem of Algebra, while the second (computational math) deals with an actual representation of the reals and complex numbers and the actual root finding algorithm that the FTA exhibits. In this paper we will not elaborate on this any further, but just point out that this was an important motivation for choosing to work with Coq. At present work is being done in program extraction from the C-CoRN library, and this relies heavily on the fact that the library is constructive.

4 Contents

C-CoRN includes at present formalizations of significant pieces of mathematics. In this section, we will give an overview of the main results which are in the library. So far everything in the library has been formalized constructively. Although we do not exclude adding non-constructive theorems to the library, working constructively has some added value, as indicated in the previous section.

The C-CoRN library is organized in a tree-like fashion. This structure agrees with the dependencies among the mathematics being formalized. In Figure 1 the directory structure of C-CoRN can be seen.

At the bottom of the library are the tactics files and the Algebraic Hierarchy, developed in the framework of the FTA-project. Most of the tactics are to make equational reasoning easier, see [20]. In the Algebraic Hierarchy, the most common algebraic structures occurring in mathematics (e.g. monoids, rings, (ordered) fields) are formalized in a cumulative way and their basic properties are proved. For reasons which will be discussed in Section 5, this formalization proceeds in an abstract way, described in detail in [18]. Furthermore, the hierarchy

[3] Right now, C-CoRN is not open either: we want to have 'central control' over the library. But the point we want to make here is that the Proof Assistant Coq, on which C-CoRN is based, *is* open.

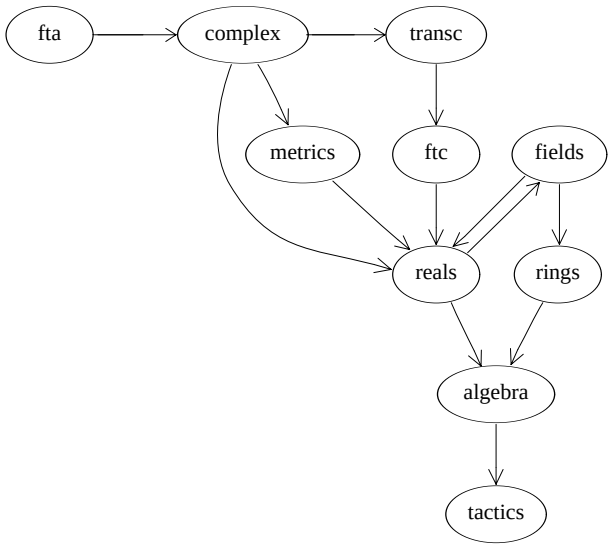


Fig. 1. Directory structure of C-CoRN

is built in such a way that more complex structures are instances of simpler ones, i.e. all lemmas which have been proved e.g. for groups are inherited by all ordered fields.

Real number structures are defined as complete Archimedean ordered fields. The C-CoRN library includes not only a concrete model of the real numbers (namely, the standard construction as Cauchy sequences of rationals) but also a formal proof that any two real number structures are equivalent, and some alternative sets of axioms that can be used to define them. Thanks to these results, it makes sense to work with a generic real number structure rather than with any concrete one. This part of the library is described in detail in [17].

Among generic results about real numbers included in the library we point out the usual properties of limits of Cauchy sequences and the Intermediate Value Theorem for polynomials, which allows us in particular to define n^{th} roots of any nonnegative number.

At this point the library branches in various independent directions.

One of the branches consists of the rest of the original FTA-library, which contains the definition of the field of complex numbers and a proof of the Fundamental Theorem of Algebra due to M. Kneser; this work is discussed in [21]. Other important results in this part of the library include the definition of important operations on the complex numbers (conjugation, absolute value and n^{th} roots) and their properties.

The second main branch deals with a development of real analysis following [4]. Here, properties of real valued functions are studied, such as continuity, differentiability and integrability. Several important results are included, among which Rolle's Theorem, Taylor's Theorem and the Fundamental Theorem of Cal-

culus. Also, this segment of the library includes results allowing functions to be defined via power series; as an application, the exponential and trigonometric functions are defined and their fundamental properties are proved. Logarithms and inverse trigonometric functions provide examples of function definition via indefinite integrals.

A separate branch of the library, currently in its initial stage of development, deals with topological and metric spaces. At present this part of the library is very small; it includes simple properties of metric spaces, as well as a proof that the complex numbers form a metric space.

The sizes of the different parts of the C-CoRN library are shown in Figure 2. This data does not include the files dealing with metric spaces, as these are still

Description	Size (Kb)	% of total
Algebraic Hierarchy (incl. tactics)	533	26.4
Real Numbers (incl. Models)	470	23.3
FTA (incl. Complex Numbers)	175	8.7
Real Analysis (incl. Transc. Fns.)	842	41.6
Total	2020	100

Fig. 2. Contents and size of C-CoRN (input files)

in an early stage of development, nor those dealing with applications to program extraction, which will be discussed in Section 6.

5 Methodology

In order to successfully pursue the goal of formalizing a large piece of mathematics, it is necessary to work in a systematic way. In this section we look at some of the general techniques that are used to make the development of the C-CoRN library more fluent.

We will focus on four main aspects:

Documentation: In order to be usable, a library needs to have a good documentation that allows the user to quickly find out exactly what results have been formalized, as well as understand the basic notations, definitions and tactics.

Structuring: Another important issue is the structure of the library. We feel that lemmas should be somehow grouped according to their mathematical content rather than to any other criterion; e.g. all lemmas about groups should be put together in one place, all lemmas about order relations in another, and so on. A related aspect is how to name lemmas. Experience shows that following some simple rules can make the process of looking for a particular result both easier and faster.

Abstract approach: C-CoRN aims at generality. This suggests that mathematic structures (e.g. real numbers) be formalized in an abstract way rather

than by constructing a particular example and working on it. We will examine some of the consequences of this style of working.

Automation: Finally, any successful theorem-proving environment must have at least some automation, otherwise the proving process quickly becomes too complex. We give an overview of the specific tactics that were developed for C-CoRN and show how they help in the development of the library.

Documentation

Providing a good documentation for the formalized library in parallel with its development was a central preoccupation from the beginning of the FTA-project. In fact, having a human-readable description of what has been formalized can be very useful in communicating not only content but also ideas, notations and even some technical aspects of the formalization process.

Such a documentation should at any given moment reflect the state of the library, and as such should be intrinsically linked to the script files. (This is also the idea behind Knuth's 'Literate Programming'. Aczel and Bailey use the term 'Literate Formalization' for this method applied to formalized mathematics [3].) At present, Coq provides a standard tool, called `coqdoc` (see [16]), that automatically generates postscript and `html` documentation from the Coq input files. Additional information can be introduced in the documentation via comments in the script file.

The C-CoRN documentation is presently produced using `coqdoc`. It includes all definitions, axioms and notation as well as the statements of all the lemmas in the library, but no proofs: being meant as *documentation*, rather than *presentation* of the library, we feel that the presence of long and incomprehensible proof scripts in the documentation would undermine its purpose. For the same reason, tactic definitions are omitted from the documentation, but not their description: although the actual code is not presented, the behavior of the existing C-CoRN specific tactics is explained as well as how and when they can be used.

In the `html` version, hyperlinks between each occurrence of a term and its definition allow the users to navigate easily through the documentation, being able to check quickly any notion they are not familiar with.

Structuring

There are several ways that the lemmas and files in a library of formalized mathematics can be organized. The current trend in most major systems, as discussed in Section 8, seems to be adding individual files to the library as independent entities, and seldom if ever changing them afterward (except for maintenance).

However, C-CoRN is intended as a growing system upon which new formalizations can be made. The approach above described directly conflicts with this purpose; for it typically leads to dispersion of related lemmas throughout the library and unnecessary duplication of work.

For this reason, we feel that this is not the best way to proceed. In C-CoRN, lemmas are organized in files according to their statements, and files are distributed in directories according to their subjects. Thus, different areas of mathematics appear in different directories and different subjects within one area will be different files in the same directory. Of course, this requires central control over the repository: after an extension, the library has to be reconsidered to put the definitions and lemmas in the 'right' place. This may become problematic if many files are contributed within a short time.

No part of the library is, strictly speaking, immutable: new lemmas can be added at any time to existing files, if they are felt to belong there. In this way, new lemmas then become immediately available to other users. In practice, though, the lower in the tree structure of Figure 1 a file is, the less often it will be changed.

Coupled with this method of working, the documentation system described above makes looking for a particular statement a simpler process than in most of the systems the authors are acquainted with. But in addition to this, naming conventions are adopted throughout C-CoRN that allow experienced users to find a specific lemma even quicker without needing to consult the documentation. These naming conventions, however, are too specific to be explainable in a short amount of space. The interested reader can find them throughout the C-CoRN documentation.

Abstract Approach

One finds two approaches to formalizing algebraic operations. On the one hand one just has concrete types for various number structures, like the natural numbers, the integers, the real numbers and the complex numbers, and for each of those one defines a separate set of arithmetical operations. On the other hand – which is the approach that is followed in C-CoRN, as described in [18] – one can have a hierarchy of the commonly appearing algebraic structures in mathematics, such as groups, rings, fields and ordered fields, and then instantiate these to specific number structures. In this approach the theory of the real numbers will not refer to a specific type of real numbers, but just to a type of 'real number structure', which later can be instantiated to a concrete model.

This second approach has advantages and disadvantages. An advantage is that the theory that is developed for a certain structure is maximally reusable. For example the group properties can be reused for the integers, the rational numbers, the real numbers, polynomials, vectors in a vector space, and so on. In our abstract approach each of these structures will be just an instance of the already existing algebraic type of groups, and the laws that were proved for it will be immediately available. In the first approach the same theory has to be developed over and over again, every time a new structure with the same algebraic properties is defined.

Another advantage is that the same notation will be used for the same algebraic operation. This is especially useful in a system that has no overloading, like Coq. For instance, in the first approach one has different additions on natural

numbers, integers, real numbers, while in C-CoRN all of these are simply written as `(x[+]y)`.

A third advantage is that the development of the theory will more closely follow the development of algebra in a mathematical textbook.

The main disadvantage of the abstract approach is that the terms that occur in the formalization are usually much bigger, because they have to refer to the specific structure used. Also, because of the hierarchy of the types of algebraic structures, there will be functions needed in the terms to get to the right kind of algebraic structure. However this is not a problem for the user, since all these operations are implicit: the specific structure is generally an implicit argument, while the functions that map algebraic structures are coercions. On the other hand, internally these terms are still big, so it slows down the processing of the formalization by Coq.

Another slight disadvantage of this approach is that sometimes proofs can be less direct than in the case that all functions are concretely defined. This also affects program extraction. For instance, if one knows that one is dealing with the rational numbers, a proof might be possible that gives a much better extracted program. In the case that one has to give a proof from an abstract specification, this optimization might not be available.

Automation

An important part of the C-CoRN library consists in tools designed to aid in its own development. Together with definitions, notations and lemmas, several automated tactics are defined throughout C-CoRN.

These tactics vary in complexity and in their underlying mechanism. Thus, there are several tactics based on Coq's `Auto` mechanism, which simply performs Prolog-style depth-first search on a given collection of lemmas. Each tactic is designed to work within a specific subject, such as reasoning equationally in different algebraic structures (`Algebra`) or proving continuity of real-valued functions (`Contin`).

Other tactics base themselves on the principle of reflection to tackle wider classes of problems in a more uniform and more efficient way. We mention `Rational`, described in detail in [18], which provides proofs of equalities in rings or fields, but can solve a much larger class of goals than `Algebra`; and `Deriv`, described in [13], a reflective tactic which can prove goals of the form $f' = g$ when f and g are real-valued (partial) functions. Although tactics based on reflection are usually more powerful than those based on `Auto`, they are also more time-consuming when the goals are simple, and usually cannot infer as much information from the context as the latter.

Finally, an interface for equational reasoning is also provided via the `Step` tactic. This tactic allows the user to replace a goal of the form $R(a, b)$, where R is a relation and a and b have appropriate types, by either $R(c, b)$ or $R(a, c)$, where c is a parameter given by the user. This tactic looks through a database of lemmas that state extensionality of (various types of) relations, and chooses

the one which applies to R. Then it applies either `Algebra` or `Rational` to prove the equational side condition generated by the lemma.

Presently the `Step` tactic has been generalized to work in a much wider domain than that of C-CoRN, and it is expected to be included in the standard distribution of Coq in a near future.

6 Applications

Besides the direct interest of formalizing mathematics *per se*, there are some interesting applications that are either being explored at present or are planned for the near future.

One of the consequences of working constructively, and therefore without any axioms, is that, according to the Curry-Howard isomorphism, every proof is an algorithm. In particular, any proof term whose type is an existential statement is also an algorithm whose output satisfies the property at hand.

In Coq there is an extraction mechanism available which readily transforms proof terms into executable ML-programs (see [25]). Marking techniques are used to significantly reduce the size of extracted programs, as most of the information in the proofs regards *correctness* rather than *execution* of the algorithm and can thus safely be removed. In [14] it is described how this extraction mechanism was used to obtain, from the formalized proof of the Fundamental Theorem of Algebra, an algorithm that computes roots of non-constant polynomials. At the time of writing the extracted program is too complex and does not produce any output in a reasonable amount of time; but the same method has been used to produce a *correct* program that can compute 150 digits of e in little over one minute.

Of course, the performance of these extracted programs can in no way compete with that of any existing computer algebra system. However, we feel that in situations where correctness is more important than speed, program extraction may very well be successfully used.

7 Future Developments

There are presently a number of different directions in which we would like to see C-CoRN extended in a near future.

One goal is to extend the library by adding new branches of mathematics to the formalization, or by building upon existing ones. In particular, the following areas are considered important:

Complex Analysis: presently there exist a usable algebraic theory of complex numbers and a formalization of one-variable real calculus. These provide a basis upon which a formalization of complex analysis can be built;

Basic Topology: there are no general topology results yet available in C-CoRN; a development of the elementary properties of topological spaces would not

only extend the library, but would probably make it possible to unify different parts of the library where instances of the same general lemmas are proved for specific structures;

Metric Spaces: similarly, several of the properties of the absolute value operation on real numbers and its correspondent on complex numbers are in fact instances of properties which can be proved for any distance function on a metric space. We hope that the small development currently existing in C-CoRN will enable us to prove these and other similar results in a more uniform manner;

Number Theory: on a different line, number theory seems to be a subject where an attempt at formalization could be very successful, since Coq is a system where it is for example very easy to use induction techniques. Furthermore, the preexistence of a library of real analysis would make it much easier to prove results which require manipulating specific integrals;

Group Theory: this is also a subject that would be interesting to explore in C-CoRN. Although we have built an algebraic hierarchy which includes monoids, groups and Abelian groups among its inhabitants, so far most of the development has been done only when at least a ring structure is available. Formalizing important results of group theory would be an important test ground for the usability and power of the algebraic hierarchy.

On a different note, we would like to develop applications of C-CoRN. There are currently plans to do this in two different ways:

Program extraction: the new extraction mechanism of Coq, described in [25], has made it possible to extract and execute programs from the C-CoRN library, as has been explained in [15]. However, the results so far have been slightly disappointing. Recent work has shown that much improvement may be obtainable, and we hope to pursue this topic;

Education: a formalization of basic algebra and analysis should not only be useful for additional formalizations (by researchers) but also for students, who can use it as course material. This addresses a different audience, to which the material has to be explained and motivated (using lots of examples). We believe that a formalization can be useful as a starting point for an interactive set of course notes, because it gives the additional (potential) advantages that all the math is already present in a formal way (with all the structure and semantics that one would want to have) and that one can let students actually work with the proofs (varying upon them, making interactive proof exercises). In the IDA project (Interactive course notes on Algebra, see [10]), a basic course in Algebra has been turned into an interactive course, using applets to present algebraic algorithms. Other experience, especially on the presentation of proofs in interactive course notes, is reported in [6]. We want to investigate C-CoRN as a basis for such a set of course notes.

For usability it is very important to have good automation tactics and powerful (or at least helpful) user interfaces. Along these lines, we have some concrete plans:

Equational reasoning: the present-day `Rational` tactic has as its main drawback that it only works for fields, and cannot use any specific properties which happen to hold in concrete fields such as the real numbers. A variant of `Rational` which works on rings has also been built, but it is not totally satisfactory as it is simply a copy of the original code for `Rational` with some bits left out. We would like to generalize this tactic so that the same program would work at the different levels of the algebraic hierarchy, recognizing at every stage which properties were available.

Dealing with *in*equalities: it would also be nice to have a similar tactic to reason about inequalities in ordered structures.

8 Related Work

The Coq system is distributed with a basic standard library. There is quite some duplication between what one finds there and what we offer in C-CoRN.

In particular the theory of real numbers by Micaela Mayero [26] is part of the standard library. This duplication extends to the tactics: what there is called the `Field` tactic is the `Rational` tactic in C-CoRN. However, the two theories of real numbers are quite different. Mayero's reals are classical and based on a set of axioms that constructively cannot be satisfied, while the C-CoRN reals are constructive and also have various concrete implementations. Another difference is that in the Mayero reals division is a total function which is always defined (although it is unspecified what happens when one divides by 0), which is not an option in a constructive setting. In C-CoRN, division can only be written when one knows that the denominator is apart from 0. This means that it gets three arguments, of which the third is a proof of this apartness. This difference also shows in the tactics `Field` and `Rational`. The first generates proof obligations about denominators, while the second does not need to do this, because this information already is available in the terms.

Besides the standard library, all significant Coq formalizations are collected in an archive of contributions. From the point of view of the Coq project, C-CoRN is just one of these contributions, although it is currently a considerable part of this archive. The contributions of Coq have hardly any relation to each other. There is no effort to integrate the Coq contributions into a whole, like we tried to do with the C-CoRN library. Everyone uses the standard library, but hardly anyone uses any of the other contributions.

Apart from Coq [12], there are several other systems for formalization of mathematics that have a serious library. The most important of these are: Mizar [27], HOL98 [31] and HOL Light [23], Isabelle/Isar [29], NuPRL/MetaPRL [11] and PVS [30].

The largest library of formalized mathematics in the world is the library of the Mizar system, which is called Mizar Mathematical Library or MML. To give an idea of the size of this library: the source of the C-CoRN repository is about 2 megabytes, the sources of the Coq standard library together with the Coq contributions are about 25 megabytes, while the source of MML is about

55 megabytes. (Of course these sizes are not completely meaningful, as a Coq encoding of a proof probably has a different length from a Mizar encoding of the same proof. Still it is an indication of the relative sizes of the libraries of both systems.)

Some of the theorems that are the highlights of C-CoRN are also proved in MML, like the Fundamental Theorem of Algebra and the Fundamental Theorem of Calculus.

Unlike the Coq contributions, the MML is integrated into a whole: all Mizar articles use all the other articles that are available. So our goals with C-CoRN are similar to that of MML. However, there also are some differences. First, the MML is classical, while almost all of C-CoRN currently is constructive. More importantly, although the MML is revised all the time to make it more coherent mathematically, until recently theorems were never moved. In C-CoRN related theorems are in the same file, but in MML a theorem could potentially be found anywhere. Recently the Encyclopedia of Mathematics in Mizar project (or EMM) has been started to improve this situation, but it is still in an early stage. Another difference is that in C-CoRN arithmetic is formalized in an abstract style, as discussed above. In the MML both the abstract and concrete styles are available, but the majority of the formalizations use the latter.

Mizar users are encouraged to submit their formalizations to the MML. Mizar is not designed to allow large libraries that are separate from the MML: the performance of the system degrades if one tries to do this. When Mizar users submit their work to MML they sign a form to give up the copyright to their work, so that it can be revised if necessary by the Mizar developers. An approach on how to integrate work by others into C-CoRN still has to be developed. It will need to address the issues that (1) we want to discourage the development of libraries on top of C-CoRN that are not integrated into it, and (2) we want to be free to revise other people's work without getting conflicts over this.

The other systems mentioned above have a library similar to that of Coq. These libraries also have similarities to the C-CoRN library. For instance, in the HOL Light library both the Fundamental Theorem of Algebra and the Fundamental Theorem of Calculus are proved. The Isabelle, NuPRL and PVS libraries also contain proofs of the Fundamental Theorem of Calculus. A comparison between these proofs and the one in C-CoRN can be found in [13].

Of course the C-CoRN repository is the only serious constructive library available today.

References

1. A. Asperti and B. Wegner. MOWGLI – A New Approach for the Content Description in Digital Documents. In *Proc. of the 9th Intl. Conference on Electronic Resources and the Social Role of Libraries in the Future*, volume 1, Autonomous Republic of Crimea, Ukraine, 2002.
2. Andrea Asperti, Bruno Buchberger, and James Davenport, editors. *Mathematical Knowledge Management, 2nd International Conference, MKM 2003*, volume 2594 of *LNCS*. Springer, 2003.

3. A. Bailey. *The machine-checked literate formalisation of algebra in type theory.* PhD thesis, University of Manchester, 1998.
4. Errett Bishop. *Foundations of Constructive Analysis.* McGraw-Hill Book Company, 1967.
5. Constructive Coq Repository at Nijmegen. `http://c-corn.cs.kun.nl/`.
6. Paul Cairns and Jeremy Gow. A theoretical analysis of hierarchical proofs. In Asperti et al. [2], pages 175–187.
7. The CALCULEMUS Initiative. `http://www.calculemus.net/`.
8. Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack, editors. *Types for Proofs and Programs, Proceedings of the International Workshop TYPES 2000*, volume 2277 of *LNCS*. Springer, 2001.
9. O. Caprotti, Carlisle D.P., and Cohen A.M. The OpenMath Standard, version 1.1, 2002. `http://www.openmath.org/cocoon/openmath/standard/`.
10. Arjeh Cohen, Hans Cuypers, and Hans Sterk. *Algebra Interactive!* Springer, 1999.
11. Robert L. Constable et al. *Implementing Mathematics with the Nuprl Development System.* Prentice-Hall, NJ, 1986.
12. The Coq Development Team. *The Coq Proof Assistant Reference Manual, Version* 7.2, January 2002. `http://pauillac.inria.fr/coq/doc/main.html`.
13. Luís Cruz-Filipe. Formalizing real calculus in Coq. Technical report, NASA, Hampton, VA, 2002.
14. Luís Cruz-Filipe. A constructive formalization of the Fundamental Theorem of Calculus. In Geuvers and Wiedijk [19], pages 108–126.
15. Luís Cruz-Filipe and Bas Spitters. Program extraction from large proof developments. In *Theorem Proving in Higher Order Logics, 16th International Conference, TPHOLs 2000*, LNCS, pages 205–220. Springer, 2003.
16. Jean-Christophe Filliâtre. *CoqDoc: a Documentation Tool for Coq, Version* 1.05. The Coq Development Team, September 2003. `http://www.lri.fr/~filliatr/coqdoc/`.
17. Herman Geuvers and Milad Niqui. Constructive reals in Coq: Axioms and categoricity. In Callaghan et al. [8], pages 79–95.
18. Herman Geuvers, Randy Pollack, Freek Wiedijk, and Jan Zwanenburg. The algebraic hierarchy of the FTA Project. *Journal of Symbolic Computation, Special Issue on the Integration of Automated Reasoning and Computer Algebra Systems*, pages 271–286, 2002.
19. Herman Geuvers and Freek Wiedijk, editors. *Types for Proofs and Programs*, volume 2464 of *LNCS*. Springer-Verlag, 2003.
20. Herman Geuvers, Freek Wiedijk, and Jan Zwanenburg. Equational reasoning via partial reflection. In *Theorem Proving in Higher Order Logics, 13th International Conference, TPHOLs 2000*, volume 1869 of *LNCS*, pages 162–178. Springer, 2000.
21. Herman Geuvers, Freek Wiedijk, and Jan Zwanenburg. A constructive proof of the Fundamental Theorem of Algebra without using the rationals. In Callaghan et al. [8], pages 96–111.
22. Ferruccio Guidi and Irene Schena. A query language for a metadata framework about mathematical resources. In Asperti et al. [2], pages 105–118.
23. John Harrison. *The HOL Light manual (1.1)*, 2000. `http://www.cl.cam.ac.uk/users/jrh/hol-light/manual-1.1.ps.gz`.
24. M. Kohlhase. OMDoc: Towards an Internet Standard for the Administration, Distribution and Teaching of mathematical Knowledge. In *Proceedings of Artificial Intelligence and Symbolic Computation*, LNAI. Springer-Verlag, 2000.
25. Pierre Letouzey. A new extraction for Coq. In Geuvers and Wiedijk [19], pages 200–219.

26. Micaela Mayero. *Formalisation et automatisation de preuves en analyses réelle et numérique*. PhD thesis, Université Paris VI, December 2001.
27. M. Muzalewski. *An Outline of PC Mizar*. Fond. Philippe le Hodey, Brussels, 1993. `http://www.cs.kun.nl/~freek/mizar/mizarmanual.ps.gz`.
28. P. Naumov, M.-O. Stehr, and J. Meseguer. The HOL/NuPRL Proof Translator: A Practical Approach to Formal Interoperability. In R.J. Boulton and P.B. Jackson, editors, *The 14th International Conference on Theorem Proving in Higher Order Logics*, volume 2152 of *LNCS*, pages 329–345. Springer-Verlag, 2001.
29. T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
30. N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *The PVS System Guide*. SRI International, December 2001. `http://pvs.csl.sri.com/`.
31. Konrad Slind. *HOL98 Draft User's Manual*. Cambridge University Computer Laboratory, January 1999. `http://hol.sourceforge.net/`.

Well-foundedness of the recursive path ordering in Coq

Nicole de Kleijn, Adam Koprowski, and Femke van Raamsdonk
Department of Computer Science, Vrije Universiteit
Amsterdam, The Netherlands
ndkleijn,akoprow,femke@cs.vu.nl

June 2004

Abstract

The recursive path ordering due to Dershowitz is one of the important methods to prove termination of first-order term rewriting. The generalization to the higher-order case is due to Jouannaud and Rubio. This work is concerned with a formalization in Coq of the proof of well-foundedness of both the first- and the higher-order version of the recursive path ordering. The proof does not rely on Kruskal's tree theorem.

1 Introduction

Background. A term rewriting system is terminating if all its reduction sequences are finite. Termination of term rewriting is in general undecidable, but especially for first-order term rewriting many techniques have been developed to prove termination in particular cases. Obviously, a term rewriting system is terminating if it can be shown that for every rewrite step $s \to t$ we have $s > t$ in some well-founded ordering on the set of terms.

An important syntactical technique to prove termination of first-order term rewriting is the one using the recursive path ordering (RPO) defined by Dershowitz [2]. The starting point of the definition of RPO is a well-founded ordering on the set of function symbols. Then two terms are compared by first comparing their head-symbols, and then recursively comparing their arguments. We consider here only the case where the arguments are compared as multisets, but there are other possibilities, such as comparing them as lexicographically ordered lists. RPO turns out to be a reduction ordering, which means that in order to prove $s > t$ for every rewrite step $s \to t$, it is sufficient to prove $l > r$ for every rewrite rule $l \to r$. RPO is one of the important termination methods that is suitable for automation. A crucial point in the justification of the termination method using RPO is the proof of its well-foundedness. Originally this proof uses Kruskal's tree theorem.

For higher-order term rewriting, where terms are simply typed λ-terms, not so many termination methods exist. A bottleneck in the generalization of RPO to the higher-order case was the fact that there is no suitable extension of Kruskal's tree theorem for higher-order terms. A breakthrough was the introduction of the higher-order recursive path ordering (HORPO) by Jouannaud and Rubio [4]. A crucial point is that the proof of well-foundedness of HORPO does not rely on Kruskal's tree theorem, but uses instead the computability proof method due to Tait and Girard. The specialization to the first-order case yields a new proof of well-foundedness of RPO, that proceeds by induction on the structure of terms, which is independently also presented in the work by Persson [11].

Contribution. The present work is concerned with a formalization of well-foundedness of RPO and HORPO in Coq. On the one hand, this work can be seen as a contribution to the development of the theory for extending the proof assistant Coq with rewriting: if we want to add the possibility of calculating using term rewrite rules, then we need a formal proof of the termination of those rules. On the other hand, this work might contribute to the study of automatically proving termination of term rewriting: several tools for proving termination of term rewriting have been developed, and it would be interesting to be able to feed their output to a proof assistant such as Coq or Isabelle to verify the correctness of the output.

History and status of the work. This work is based on two master's thesis projects. The first one was carried out by Nicole de Kleijn [6] at the Vrije Universiteit in the period February 2003 - August 2003. This project is concerned with a formalization of the well-foundedness of the first-order recursive path ordering. The second project is carried out by Adam Koprowski [7], also at the Vrije Universiteit, in the period February 2004 - August 2004. This project is concerned with a formalization of the well-foundedness of the higher-order recursive path ordering, in the version with syntactic pattern matching (not modulo $\beta\eta$).

The work is still in progress. The current status of the formalization of well-foundedness of HORPO is that is consists of around 5000 lines of code in Coq version 8.0. The main hypotheses concern the computability properties. The formalization of well-foundedness of RPO is done in Coq version 7.3.1 and parts of it still need to be transferred to version 8. The present extended abstract is a first presentation of the combined projects and will be worked out in more detail later on.

Related work. Persson [11] introduces the notion of recursive path relation which generalizes the recursive path ordering by not requiring the underlying relations to be orderings. He presents a constructive proof of

well-foundedness of a general form of recursive path relations. This proof is very similar to, and independently obtained, of the specialization to the first-order case of the proof of well-foundedness of HORPO by Jouannaud and Rubio [4]. The proof in [11] is extracted from the classical proof using a minimal bad sequence argument by using open induction due to Raoult [13]. Persson also presents an abstract formalization of well-foundedness of recursive path relations in the proof-checker Agda. The main difference between the work by Persson and the current work is the level of abstraction: here we are much more concrete. Another difference is of course the use of Agda instead of Coq.

Leclerc [8] presents a formalization in Coq of well-foundedness of RPO with the multiset ordering. The Coq script consists of about 250 pages and hence is not presented in full detail in the reference. The focus is on giving upper bounds for descending sequences in RPO. There are quite some differences between the work by Leclerc and the current work. Most datatypes are represented differently, and also the current development is like the one by Persson substantially shorter.

Murthy [9] formalizes a classical proof of Higman's lemma, a specific instance of Kruskal's tree theorem, in a classical extension of Nuprl 3. The classical proof is due to Nash-Williams and uses a minimal bad sequence argument. The formalized classical proof was automatically translated into a constructive proof using Friedman's A-translation. The formalization is very big: around 10 megabytes.

Berghofer [1] presents a constructive proof of Higman's lemma in Isabelle. The constructive proof is due to Coquand and Fridlender.

Organization of the paper. In this paper we focus on the formalization of the well-foundedness of the higher-order version of the recursive path ordering, since the first-order case can be obtained as a specialization. Several variations of the higher-order version of the recursive path ordering exist, see [5, 12]. The formalization deals so far only with the most simple version, where HORPO is defined for algebraic-functional systems (AFSs) with a simple typing discipline. An important point here is that we do not work modulo β, and matching is syntactic. Further, we consider the version of HORPO without the computable closure.

To start with, the definition of algebraic-functional systems (AFSs) with a simple typing discipline is given. Along the way we comment on some aspects of the formalization. Then the definition of HORPO, and the proof of its well-foundedness are presented. The definition and the well-foundedness proof use a part of the theory of finite multisets, which is formalized in [7].

2 Higher-order term rewriting

There are several definitions of higher-order rewriting. They fall into two categories: the ones where rewriting is roughly defined modulo β, and the ones where plain rewriting is used, that is, matching is syntactic. Definitions in the first category are the combinatory reduction systems (CRSs) introduced by Klop, the similar class of expression reduction systems (ERSs) introduced independently by Khasidashvili, and the higher-order rewrite systems (HRSs) defined by Nipkow. In the second category are the algebraic-functional systems (AFSs) introduced by Jouannaud and Okada [3].

In the formalization we restrict attention to the case of AFSs with a simple type discipline, like in the first definition of HORPO. In this section we recall the main ingredients of the definition of these AFSs.

Types. To start with, we assume a non-empty set of *base types* with decidable equality. From the set of base types, simple types are built inductively according to the following grammar:

$$A, B ::= a \mid A \to B$$

with a a base type. Simple types, shortly called types, are written as $A, B, C, \ldots$.

For the definition of HORPO, we use the following equivalence relation on the set of simple types: A and B are equivalent if they have the same arrow structure. That is, the equivalence relation $\equiv$ on the set of types is defined as the smallest relation that satisfies the following two clauses:

1. we have $a \equiv b$ if a and b are both base types, and
2. we have $A \to A' \equiv B \to B'$ if $A \equiv A'$ and $B \equiv B'$.

For instance, $\mathsf{nat} \to \mathsf{bool} \equiv \mathsf{natlist} \to \mathsf{nat}$.

In the formalization, simple types are defined as an inductive type, and the equivalence on the set of types is a fixed point definition.

Pre-terms. We assume a countably infinite set of *variables* written as $x, y, z, \ldots$. Also, a non-empty set of *function symbols* is assumed, which is supposed to be disjoint from the set of variables. Every function symbol comes equipped with a unique type. We assume a decidable equality relation on the set of function symbols.

Pre-terms, also called raw terms, are built from variables, function symbols, abstraction and simple types, and application. The set of *pre-terms* is defined inductively according to the following grammar:

$$M, N ::= x \mid f \mid \lambda x : A.\, M \mid @(M, N)$$

The terminology 'pre-term' has nothing to do with being in β-normal form or not; it means that no typing relation is taken into account yet.

In the formalization, De Bruijn indices are used to avoid the α-conversion problems. So a variable is a natural number, and an abstraction is of the form $\lambda^A M$. Pre-terms are defined as an inductive type. In the paper, we use named variables for the sake of readability.

The typing relation. An *environment* is a set of declarations of the form $x : A$ with all variables distinct. A pre-term M is said to be *typable* if we can derive $\Gamma \vdash M : A$ for some environment Γ and some type A. A typable pre-term is called a *term*. The typing relation is defined by the following clauses:

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : A \to B}$$

$$\frac{f : A}{\Gamma \vdash f : A} \qquad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash @(M, N) : N}$$

Given an environment, a term has a unique type.

Because of the De Bruijn notation, an environment in the formalization is just a list of types. The typing relation

```
Typing : Env -> Preterm -> SimpleType -> Set
```

is defined inductively, taking as inputs an environment, a pre-term, and a type. Finally, a term is a record consisting of an environment Γ, a pre-term M, a type A and a typing derivation $\Gamma \vdash M : A$.

There is a slight difference between the definition of pre-terms and terms as used here, and the one in [4]. There a function symbol has besides a type also an arity, expressing the number of arguments it should get. A function symbol f with arity 3 can be used to build a term of the form $f(a, b, c)$. If this term has type $D \to E$, then it can be used to form the application $@(f(a, b, c), d)$ of type E. Here and in the formalization we do not use functional application but only the binary application written as $@$. So a function symbol $f : A \to B \to C \to D \to E$ can for instance be used to build the terms $@(f, a)$, $@(@(f, a), b)$, and $@(@(@(f, a), b), c)$.

Substitution. A *substitution* is a finite mapping from variables to terms written as $\{x_1 \mapsto M_1, \ldots, x_n \mapsto M_n\}$. In the formalization, the definition of substitution is quite involved. It deals for instance with the lifting of indices.

Rewriting. The focus in the formalization so far is on well-foundedness of the higher-order recursive path ordering. For the definition of the ordering and the proof of its well-foundedness it is in fact not necessary to know

the rewrite relation of AFSs. But of course HORPO is defined with the aim to prove termination of rewriting, so we continue here by explaining shortly the definition of the rewrite relation. This part is not present in the formalization so far.

The definition of rewriting in an AFS is as follows. First, the rewrite relation $\to_R$ is induced by a set of rewrite rules. A *rewrite rule* is of the form $\Gamma \vdash l \to r : A$, where both $\Gamma \vdash l : A$ and $\Gamma \vdash r : A$ are derivable. We have a *rewrite step* $M \to_R N$ if there is a position p in M such that $M|_p$ is of the form $l\sigma$. Further, $N = M[p \leftarrow r\sigma]$, that is, N is obtained from M by replacing the subterm $l\sigma$ at position p by the subterm $r\sigma$. We do not give all formal definitions here; they are mainly standard. Note that we do not mention typing compatibility issues. An important thing is that matching is *syntactic*.

Besides the algebraic rewrite rules that induce the rewrite relation $\to_R$, there is also the rule for β-reduction:

$$@(\lambda x : A.\, M, N) \to_\beta M\{x \mapsto N\}$$

The rewrite relation of interest is the union of $\to_R$ and the rewrite relation $\to_\beta$ induced by the β-reduction rule. This rewrite relation is denoted by $\to_{R\beta}$ or also shortly by $\to$.

We conclude the description of AFSs by giving some examples of rewriting systems. In the examples we will sometimes write $f(M_1, \ldots, M_n)$ instead of $@(\ldots @(f, M_1) \ldots M_n)$. This is just a notation meant to increase readability. We also use the convention that functional variables are written in upper case.

Example 2.1.

1. First we consider the example for map. We assume two base types: nat and $\mathsf{natlist}$. The following signature is used:

$$\begin{array}{lcl} \mathsf{nil} & : & \mathsf{natlist} \\ \mathsf{cons} & : & \mathsf{nat} \to \mathsf{natlist} \to \mathsf{natlist} \\ \mathsf{map} & : & \mathsf{natlist} \to (\mathsf{nat} \to \mathsf{nat}) \to \mathsf{natlist} \end{array}$$

   The rewrite rules for map are the following:

$$\begin{array}{lcl} \mathsf{map}(\mathsf{nil}, Z) & \to & \mathsf{nil} \\ \mathsf{map}(\mathsf{cons}(h, t), Z) & \to & \mathsf{cons}(@(Z, h), \mathsf{map}(t, Z)) \end{array}$$

   Here $Z : \mathsf{nat} \to \mathsf{nat}$ is a variable for a function on natural numbers, and $h : \mathsf{nat}$ and $t : \mathsf{natlist}$ are variables for the head and the tail.

   As we will see later, this rewrite system can be proved to be terminating using HORPO.

2. The second example is Gödel's recursor for natural numbers. We assume base types nat and A. The following signature is used:

$$\begin{array}{lcl} \mathsf{0} & : & \mathsf{nat} \\ \mathsf{s} & : & \mathsf{nat} \to \mathsf{nat} \\ \mathsf{rec} & : & \mathsf{nat} \to A \to (\mathsf{nat} \to A \to A) \to A \end{array}$$

The rewrite rules for rec are the following:

$$\begin{array}{lcl} \mathsf{rec}(\mathsf{0}, y, Z) & \to & y \\ \mathsf{rec}(\mathsf{s}(x), y, Z) & \to & @(Z, x, \mathsf{rec}(x, y, Z)) \end{array}$$

Here $Z : \mathsf{nat} \to A \to A$ is a functional variable.

Also this AFS can be shown to be terminating using HORPO.

3. Finally, we consider an example from [3]. We assume a base type A, and the following signature:

$$f : A \to A \to A$$

There is one rewrite rule for the symbol f:

$$f(@(Z, x), x) \to f(@(Z, x), @(Z, x))$$

The rewrite relation induced by this rewrite rule and the rule for β-reduction is not terminating:

$$\begin{array}{ll} f(@(\lambda x : A.\, x, y), y) & \to \\ f(@(\lambda x : A.\, x, y), @(\lambda x : A.\, x, y)) & \to \\ f(@(\lambda x : A.\, x, y), y) & \to \\ \ldots & \end{array}$$

3 The higher-order recursive path ordering

In this section we give the definition of HORPO for AFSs with simple types. This is the first definition of HORPO as presented in [4]. A more powerful version of HORPO using also the notion of computable closure is presented also in that paper. Later on, HORPO has been extended in several ways, as presented in [5, 12]. In this paper and in the formalization, we only consider the first, most simple version.

We assume a well-founded ordering on the set of function symbols, called a precedence, denoted by $\triangleright$. We use again the notation $f(M_1, \ldots, M_n)$. Also, we use the notation $@(M_1, \ldots, M_n)$ with just one instead of $n - 1$ application symbols. Given an environment Γ, the type of a term M is unique and denoted by $\mathsf{type}(M)$.

Definition 3.1. Suppose that $\Gamma \vdash M : A$ and $\Gamma \vdash N : A'$ for some environment Γ, and for types A and A' with $A \equiv A'$. Then we have $\Gamma \vdash M \succ N$ if this can be inferred using the following clauses:

1. $M = f(M_1, \ldots, M_k)$ for some $k \geq 0$
   there exists $i \in \{1, \ldots, k\}$ such that $M_i \succeq N$

2. $M = f(M_1, \ldots, M_k)$ for some $k \geq 0$
   $N = g(N_1, \ldots, N_l)$ for some $l \geq 0$
   $f \triangleright g$
   for all $j \in \{1, \ldots, l\}$ we have:
   if $\mathsf{type}(M) \equiv \mathsf{type}(N_j)$ then $M \succ N_j$,
   if $\mathsf{type}(M) \not\equiv \mathsf{type}(N_j)$, then
   there exists $i \in \{1, \ldots, k\}$ such that $M_i \succeq N_j$

3. $M = f(M_1, \ldots, M_k)$ for some $k \geq 1$
   $N = f(N_1, \ldots, N_k)$
   $\{\{M_1, \ldots, M_k\}\} \succ_{mul} \{\{N_1, \ldots, N_k\}\}$

4. $M = f(M_1, \ldots, M_k)$ for some $k \geq 0$
   $N = @(N_1, \ldots, N_l)$ for some $l \geq 2$
   for all $j \in \{1, \ldots, l\}$ we have:
   if $\mathsf{type}(M) \equiv \mathsf{type}(N_j)$ then $M \succ N_j$,
   if $\mathsf{type}(M) \not\equiv \mathsf{type}(N_j)$, then
   there exists $i \in \{1, \ldots, k\}$ such that $M_i \succeq N_j$

5. $M = @(M_1, M_2)$
   $N = @(N_1, N_2)$
   $\{\{M_1, M_2\}\} \succ_{mul} \{\{N_1, N_2\}\}$

6. $M = \lambda x : B.\, M_0$
   $N = \lambda x : B'.\, N_0$
   $M_0 \succ N_0$

$\diamond$

A few remarks concerning this definition:

- This is the definition that is formalized.
- Two terms can only be compared using HORPO if they have in the same environment equivalent types. We assume this, also if at most places we write only $M \succ N$.
- We use $\{\{\ldots\}\}$ to denote a multiset. The extension of $\succ$ to multisets is denoted by $\succ_{mul}$. More comments on this issue can be found below.

- A first difference with the definition as given in [4] is that we have one clause less: in case of two terms starting with the same head-symbol, we compare the arguments only by the multiset extension of $\succ$. In the original definition there is yet another clause, were the arguments are put in lists which are compared using the lexicographic extension of $\succ$. We do not consider this clause in the formalization. (In the original definition of RPO due to Dershowitz, the lexicographic clause is not present. The lexicographic version of RPO is introduced in an unpublished note by Kamin and Lévy.)

- The second (and last) difference with the definition as given in [4] is that the statement

  for all $j \in \{1, \ldots, l\}$ we have:
  if $\mathsf{type}(M) \equiv \mathsf{type}(N_j)$ then $M \succ N_j$,
  if $\mathsf{type}(M) \not\equiv \mathsf{type}(N_j)$, then
  there exists $i \in \{1, \ldots, k\}$ such that $M_i \succeq N_j$

  as in the definition above is deterministic. This is a slight change compared to the original definition, where a non-deterministic statement is given. It is remarked already there that this non-deterministic statement can be replaced by the deterministic statement that we use here. Clearly the deterministic version is more suitable for a formalization.

- Note that in clause 5 for application M_1 can only be compared with N_1, and M_2 can only be compared with N_2 for typing reasons. Hence there are three possibilities (in fact we need to know the definition of the multiset extension in order to see this): $M_1 \succ^* N_1$ and $M_2 = N_2$, or $M_1 = N_1$ and $M_2 \succ^* N_2$, or $M_1 \succ^* N_1$ and $M_2 \succ^* N_2$, with $\succ^*$ the transitive closure of $\succ$.

- In fact, for typing reasons in clause 4 we have $k \geq 1$.

- It can be shown that clause 5 for application is redundant in the case that M_1 is a function symbol. This is used in the formalization.

- It is crucial that only terms of equivalent types are compared using HORPO. However, note that the precedence does not take the types of the function symbols into account; it is possible that function symbols of different types are related in the precedence.

- The first three clauses of the definition together give the original first-order definition of the recursive path ordering as introduced by Dershowitz [2]. To make this slightly more precise we need to represent a first-order term rewriting system as an AFS which is not very hard:

take one base type a (representing the set of terms), and give a function symbol of arity n the type $a \to \ldots \to a \to a$ with in total $n+1$ times a.

In the following example we show that the rewrite rules for map and for the recursor rec can be oriented using HORPO.

Example 3.2.

1. First we consider the example for map. We use the precedence $\mathsf{map} \triangleright \mathsf{cons}$.

   We have $\mathsf{map}(\mathsf{nil}, Z) \succ \mathsf{nil}$ by an application of clause 1.

   In order to show $\mathsf{map}(\mathsf{cons}(h,t), Z) \succ \mathsf{cons}(@(Z,h), \mathsf{map}(t,Z))$, we apply clause 2. Then two things need to be shown: first we need to show that $\mathsf{map}(\mathsf{cons}(h,t), Z) \succ @(Z,h)$ (note that $\mathsf{natlist}$ and nat are equivalent types), and second that $\mathsf{map}(\mathsf{cons}(h,t), Z) \succ \mathsf{map}(t,Z)$. The first holds by an application of clause 4. The second holds by an application of clause 3 because $\mathsf{cons}(h,t) \succ t$.

2. Second we consider the example for rec.

   We have $\mathsf{rec}(\mathsf{0}, y, Z) \succ y$ by an application of clause 1.

   In order to show $\mathsf{rec}(\mathsf{s}(x), y, Z) \succ @(Z, x, \mathsf{rec}(x,y,Z))$ we apply clause 4. Then three things remain to be shown. First, we have $Z \succeq Z$. Second, we have $\mathsf{rec}(\mathsf{s}(x), y, Z) \succ x$ by an application of clause 1. Note that the types A and nat are equivalent. Third, we have $\mathsf{rec}(\mathsf{s}(x), y, Z) \succ \mathsf{rec}(x,y,Z)$ by an application of clause 3 because $\mathsf{s}(x) \succ x$.

In the formalization, the definition of HORPO is formalized by mutually dependent inductive types. To start with, there is the inductive definition of HORPO that takes into account the environment and the types of the terms to be compared. Depending on the definition of HORPO are the extension of HORPO to multisets of terms (used in clause 3 of the definition above) and the reflexive closure (in the definition above written as $\succeq$) of HORPO. There is a mutual dependency between the definition of HORPO and the definition of a relation that is called pre-HORPO in the formalization. The clauses in the definition of pre-HORPO correspond to the clauses of the definition given above. Pre-HORPO does not take the environments and types of the terms into account. It uses another inductive definition, which formalizes the statement concerning the arguments used both in clause 2 and in clause 4, which in its turn relies on the definition of HORPO.

HORPO is used to show well-foundedness of the algebraic rewrite rules of an AFS. Besides those rules, there is also the rule for β-reduction, and the rewrite relation we are interested in is the union of the algebraic rewrite

relation $\rightarrow_R$ and β-reduction $\rightarrow_\beta$. The well-foundedness proof is hence concerned with the union $\succ \cup \rightarrow_\beta$.

The main goal of the formalization so far is to prove well-foundedness of $\succ \cup \rightarrow_\beta$. It is not (yet) concerned with issues as proving that $\succ$ is transitive, or stable under application of substitutions and contexts. We do not consider those issues in this paper either.

3.1 Finite multisets

The definitions of HORPO and RPO use the extension of those relations to finite multisets. In the recursive call of the definitions, when the multiset extension is used, it is not yet known whether RPO and HORPO are orderings. As a matter of fact, many versions of HORPO are not transitive, and hence are not orderings. So the definitions of HORPO and RPO use the multiset extension of a relation which is not necessarily transitive. In this subsection we briefly comment on such multiset extensions, which form a substantial part of the formalization presented in [7].

We assume a set X and a relation $>$ on X. The intuition of a finite multiset over X is that it is a finite set with repeated elements, or a finite list where the order is irrelevant. More formally, a *finite multiset* over X is a mapping $m : X \rightarrow \mathbb{N}$ with $m(x) \neq 0$ for only finitely many elements of X. We denote a multiset m using $\{\{\ldots\}\}$ and repeating each element x exactly $m(x)$ times. For instance, $\{\{1, 1, 2\}\}$ is the multiset m over $\mathbb{N}$ with $m(1) = 2$, and $m(2) = 1$, and $m(n) = 0$ for all other n. The operations multiset union, denoted by $\cup$, element, denoted by $\in$, are defined as usual. Equality on multisets is denoted by $=$. The empty multiset is denoted by $\emptyset$.

In the formalization, finite multisets are first rendered as an abstract datatype. Then a representation using finite lists is given.

The definition of *multiset reduction*, denoted by $\Rightarrow$, is as follows: we have $m \Rightarrow n$ if

$$\begin{aligned} &m = m' \cup \{\{x\}\} \\ &n = m' \cup n' \\ &\forall y \in n' : x > y \end{aligned}$$

The multiset extension of $>$, denoted by $>_{mul}$, is defined as the transitive closure of $\Rightarrow$.

An alternative definition, here denoted by $>>$, of the multiset extension of $>$ is as follows: we have $m >> n$ if

$$\begin{aligned} &m = p \cup m' \text{ with } m' \neq \emptyset \\ &n = p \cup n' \\ &\forall y \in n' \, \exists x \in m' : x > y \end{aligned}$$

The two definitions can be proved to be equivalent if the underlying relation $>$ is transitive. This is done in the formalization. If the underlying relation $>$ is not transitive, then the two definitions are not necessarily the same.

Consider for instance the relation $>$ defined by $a > b$ and $b > c$, but where $a > c$ does not hold. We have $\{\{a\}\} \Rightarrow \{\{b\}\}$ and $\{\{b\}\} \Rightarrow \{\{c\}\}$ and hence $\{\{a\}\} >_{mul} \{\{c\}\}$. However, $\{\{a\}\} >> \{\{c\}\}$ does not hold.

Not all versions of HORPO are transitive. Therefore in the development transitivity is not assumed. Hence it is relevant to know which definition of the multiset extension is used: it is the first one, where the multiset extension is defined as the transitive closure of the multiset reduction. The version of HORPO used in the formalization is transitive; however, this is issue is not considered in the formalization.

If $>$ is not transitive, then from $m >_{mul} n$ we cannot conclude that for every $y \in n$ there is a $x \in m$ such that $x > y$. See for instance the example given above, where we have $\{\{a\}\} >_{mul} \{\{c\}\}$, but not $a > c$. We do have the weaker property that from $m >_{mul} n$ we can conclude that for every $y \in n$ there is a $x \in m$ such that $x >^* y$, with $>^*$ the transitive closure of $>$. This property is used in the proof of well-foundedness.

The formalization contains several properties of finite multisets. The most important one, which is used to justify the well-founded induction in the proof of the key lemma for the well-foundedness of $\succ \cup \rightarrow_\beta$, is the following: if all elements of a finite multiset m are well-founded with respect to $>$, then m is well-founded with respect to $>_{mul}$. Also: if $>$ is well-founded on X, then $>_{mul}$ is well-founded on the set of finite multisets over X. The proof that is formalized proceeds mainly by well-founded induction and is due to Buchholz [10]. It is already formalized in Isabelle and HOL.

4 Well-foundedness of HORPO

In this section we present the main steps of the proof of well-foundedness of $\succ \cup \rightarrow_\beta$ due to Jouannaud and Rubio [4]. The proof makes use of the computability predicate due to Tait and Girard with respect to the relation $\succ \cup \rightarrow_\beta$. Some standard properties of computability are used. In the formalization, they are assumed as hypotheses.

The specialization of the well-foundedness proof to the case of first-order RPO is also independently given by Persson [11]. There the proof is obtained from the classical proof using a minimal bad sequence argument by means of the principle of open induction due to Raoult [13]. The outline of the classical proof (for the first-order case, considering only $\succ$) is as follows: Suppose that it is not the case that all $\succ$-sequences are finite. Then there is an infinite $\succ$-sequence. Then there is an infinite *minimal* sequence in the sense that for every step $M \succ N$ in the infinite minimal sequence any alternative $M \succ N'$ with N' a subterm of N would yield only finite sequences. The constructive part of the classical proof consists of showing by well-founded induction that all minimal sequences are finite. Then from the contradiction we conclude that all $\succ$-sequences are finite. Actually it is not necessary

to consider the subterm relation in the definition of minimal sequence; any well-founded ordering works.

We continue by presenting the key lemma in the well-foundedness proof. We use the notion of computability with respect to $\succ \cup \to_\beta$, and some properties of computability which are standard and not mentioned here explicitly. For the first-order case, a similar lemma is proved. The difference in the statement is that it is concerned with well-founded, not computable, terms. The main difference in the proof is that it deals with fewer cases.

Lemma 4.1. Suppose that $M_1, \ldots, M_k$ are computable for $k \geq 0$. Then $f(M_1, \ldots, M_k)$ is computable.

Proof. The proof proceeds by well-founded induction on the pair $(f, \{\{M_1, \ldots, M_k\}\})$, lexicographically ordered by the precedence on function symbols, and the multiset extension of $\succ \cup \to_\beta$ to finite multisets of computable (and hence well-founded with respect to $\succ \cup \to_\beta$) elements. Because both components are well-founded, also the lexicographic product is well-founded. We denote this ordering in the proof by $>$.

We proceed by showing that all successors of $f(M_1, \ldots, M_k)$ with respect to $\succ \cup \to_\beta$ are computable. Because $f(M_1, \ldots, M_k)$ is neutral, this yields that it is computable itself. We suppose $f(M_1, \ldots, M_k) \succ \cup \to_\beta N$.

0. Suppose that N is obtained by a β-reduction step in one of the M_i: $M_i \to_\beta M_i'$. Then $N = f(\ldots, M_i', \ldots)$ and $(f, \{\{\ldots, M_i, \ldots\}\}) > (f, \{\{\ldots, M_i', \ldots\}\})$. By the induction hypothesis, N is computable.

1. Suppose that N is obtained by an application of clause 1 of HORPO. Then $M_i \succeq N$ for some $M_i \in \{M_1, \ldots, M_k\}$. Since M_i is by assumption computable, by a computability property also its successor N is computable.

2. Suppose that N is obtained by an application of clause 2. Then $N = g(N_1, \ldots, N_l)$ with the conditions of clause 2. The idea is to apply the induction hypothesis using $(f, \{\{M_1, \ldots, M_k\}\}) > (g, \{\{N_1, \ldots, N_l\}\})$. Therefore we need to show that all N_i are computable with respect to $\succ \cup \to_\beta$. For N_i with the same type as M, we have $M \succ N_i$. Then computability of N_i follows by adding an inner induction hypothesis on the size of N. For N_i with another type than M, we have $M_j \succeq N_i$ for some j, and then computability of N_i follows by a computability property, since a successor of a computable term is computable.

3. Suppose that N is obtained by an application of clause 3. Then $N = f(N_1, \ldots, N_k)$ with $\{\{M_1, \ldots, M_k\}\} \succ_{mul} \{\{N_1, \ldots, N_k\}\}$. The idea is to apply the induction hypothesis using $(f, \{\{M_1, \ldots, M_k\}\}) > (f, \{\{N_1, \ldots, N_k\}\})$. Therefore we need to show that all N_i are computable. For every N_i there are two possibilities. Either we have

$N_i = M_j$ for some j. In that case N_i is computable by assumption. Or we have $M_j \succ^* N_i$. In that case computability of N_i follows from the computability property stating that the successor of a computable term is computable.

4. Suppose that N is obtained by an application of clause 4. Then $N = @(N_1, \ldots, N_l)$. We need to show that all N_i are computable. This is done is the same way as for case 2.

We conclude that all successors with respect to $\succ \cup \rightarrow_\beta$ of M are computable. Because M is neutral this yields that M is computable. $\diamond$

The formalization of the key lemma is quite hard. Now the main theorem to be formalized is the following.

Theorem 4.3. Let M be a term and let γ be a computable substitution. Then M^γ is computable.

Proof. The proof proceeds by induction on the size of M.

1. If M is a variable, so $M = x$, then $M^\gamma = \gamma(x)$ and hence by assumption computable.
2. If M is a function symbol, so $M = f$, then $M^\gamma = f$ which is computable by Lemma 4.1.
3. If M is an abstraction, so $M = \lambda x : A.\, M_0$, then it is sufficient to show that $M_0^\gamma\{x \mapsto N\}$ is computable for any computable N. If N is computable, then $\delta = \gamma \cup \{x \mapsto N\}$ is also computable. By the induction hypothesis, M_0^δ is computable. This yields that M^γ is computable.
4. If M is an application, so $M = @(M_1, M_2)$, then by induction M_1^γ and M_2^γ are both computable. This yields that M^γ is computable.

$\diamond$

In the formalization, once the key lemma is there, the abstraction clause is the most difficult. For the moment a hypothesis is added stating that if a substitution is computable, then also its lifting is computable. From this result, computability and hence well-foundedness with respect to $\succ \cup \rightarrow_\beta$ follows by using it with the identity substitution.

5 Concluding remarks

There are several obvious possibilities for extending the formalization: consider properties of HORPO such as transitivity, consider also the computability properties (this would be quite some work), and consider a variant of HORPO that applies to rewriting modulo β as in HRSs.

References

[1] S. Berghofer. A constructive proof of Higman's lemma in Isabelle. In S. Berardi, M. Coppo, and F. Damiani, editors, *Proceedings of the International Workshop Types for Proofs and Programs (TYPES 2003)*, volume 3085 of *LNCS*, pages 66–82. Springer, 2004.

[2] N. Dershowitz. Orderings for term rewriting systems. *Theoretical Computer Science*, 17(3):279–301, 1982.

[3] J.-P. Jouannaud and M. Okada. A computation model for executable higher-order algebraic specification languages. In *Proceedings of the 6th annual IEEE Symposium on Logic in Computer Science (LICS '91)*, pages 350–361, Amsterdam, The Netherlands, July 1991.

[4] J.-P. Jouannaud and A. Rubio. The higher-order recursive path ordering. In *Proceedings of the 14th annual IEEE Symposium on Logic in Computer Science (LICS '99)*, pages 402–411, Trento, Italy, July 1999.

[5] J.-P. Jouannaud and A. Rubio. Higher-order recursive path orderings 'à la carte'. `http://www.lix.polytechnique.fr/Labo/Jean-Pierre.Jouannaud/biblio.html`, 2003.

[6] N. de Kleijn. Well-foundedness of RPO in Coq. Master's thesis, Vrije Universiteit, Amsterdam, The Netherlands, August 2003.

[7] A. Koprowski. Well-foundedness of the higher-order recursive path ordering in Coq. Master's thesis, Vrije Universiteit, Amsterdam, The Netherlands, August 2004. To appear.

[8] F. Leclerc. Termination proof of term rewriting systems with the multiset path ordering: A complete development in the system Coq. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proceedings of the 2nd International Conference on Typed Lambda Calculi and Applications (TLCA'95)*, volume 902 of *LNCS*, pages 312–327, Edinburgh, UK, April 1995. Springer.

[9] C. Murthy. *Extracting constructive content from classical proofs.* PhD thesis, Cornell University, New York, USA, 1990.

[10] T. Nipkow. An inductive proof of the wellfoundedness of the mulitset order. `http://www4.informatik.tu-muenchen.de/~nipkow/misc/index.html`, October 1998. A proof due to W. Buchholz.

[11] H. Persson. *Type Theory and the Integrated Logic of Programs.* PhD thesis, Göteborg University, Göteborg, Sweden, May 1999.

[12] F. van Raamsdonk. On termination of higher-order rewriting. In A. Middeldorp, editor, *Proceedings of the 12th International Conference on Rewriting Techniques and Applications (RTA '01)*, pages 261–275, Utrecht, The Netherlands, May 2001.

[13] J.-C. Raoult. Proving open properties by induction. *Information Processing Letters*, 29:19–23, 1988.

TORPA:
Termination of Rewriting Proved Automatically

H. Zantema

Department of Computer Science, TU Eindhoven

P.O. Box 513, 5600 MB, Eindhoven, The Netherlands

e-mail h.zantema@tue.nl

Abstract

The tool TORPA (Termination of Rewriting Proved Automatically) can be used to prove termination of string rewriting systems (SRSs) fully automatically. The underlying techniques include semantic labelling, polynomial interpretations, recursive path order, the dependency pair method and match bounds of right hand sides of forward closures.

1 Introduction

Consider a finite string over $\{a, b\}$ and only one rule: if $aabb$ occurs in the string than it may be replaced by $bbbaaa$. The goal is to prove *termination*: prove that application of this rule starting on a finite string cannot go on forever. This is a surprisingly hard problem for which only ad hoc proofs were available until recently [16]. A set of such string replacement rules is called a string rewriting system (SRS) or semi-Thue system. String rewriting can be seen as a particular case of term rewriting. In this paper we describe the tool TORPA by which termination of many SRSs including this example $\{aabb \to bbbaaa\}$ can be proved fully automatically.

In the last few decades many techniques have been developed for proving termination of rewriting. In the last years work in this area concentrates on proving termination automatically: the development of tools by which a rewrite system can be entered and by which fully automatically a termination proof is generated. A strong impulse in the development of these tools was given by the competition on the International Workshop on Termination in Valencia in June 2003. The second was hold in May 2004, where TORPA turned out to be the best tool in the category of string rewriting.

The tool TORPA has been developed by the author. After having ideas in mind for years actual implementation started in July 2003. Earlier versions of

TORPA have been described in [14] (TORPA1.1) and in [15] (TORPA1.2). The present version is TORPA1.3.

All versions only work on string rewriting. On the one hand this is a strong restriction compared to general term rewriting. On the other hand string rewriting is a natural and widely accepted paradigm with full computational power.

There are many small SRSs from a wide variety of origins for which termination is proved fully automatically by TORPA within a fraction of a second. For some of them all other techniques for (automatically) proving termination seem to fail.

The main feature of TORPA is that an SRS is given as input and that TORPA generates a proof that this SRS is terminating or non-terminating. This proof is given in text. It is given in such a way that any human familiar with the basic techniques used in TORPA as they are described here, can read and check the proof. The five basic techniques used for proving termination are

- polynomial interpretations ([9]),
- recursive path order ([4]),
- dependency pairs ([1]),
- RFC-match-bounds ([6]), and
- semantic labelling ([12]).

Polynomial interpretations, recursive path order and RFC-match-bounds are direct techniques to prove termination, while semantic labelling and dependency pairs are techniques for transforming an SRS to another one in such a way that termination of the original SRS can be concluded from (relative) termination of the transformed SRS. Originally, semantic labelling was the most significant transformation used in TORPA. For very small SRSs, in particular single rules, RFC-match-bounds provide the most powerful technique. In this paper we describe how the given five techniques are applied, and give some examples of proofs as they are generated by TORPA. For more examples and a full exposition of the theory including proofs of the theorems, but excluding the most recent part on RFC-match-bounds, we refer to [14].

Other tools like TTT ([8]), CiME ([3]) and AProVE ([5]) combine dependency pairs and path orders, too, and apply them much more involved than we do. They turn out to be the best tools for proving termination of term rewriting of the moment. However, applied to SRSs all of these tools are weaker than TORPA, as was shown in the competition of termination tools of the 7th International Workshop on Termination in May 2004.

A completely different approach is RFC-match-boundedness as introduced in [6], and implemented in the tool Matchbox by Johannes Waldmann, see [10]. However, Matchbox only involves techniques related to match-boundedness, and

for typically hard examples like $aabb \to bbbaaa$ TORPA is much more efficient than the Matchbox version from before January 2004 when the authors of [6] were informed about the heuristics implemented in TORPA.

Recently our approach for RFC-match-boundedness was also implemented in AProVE, by which the score of AProVE in the 2004 competition for the category string rewriting was nearly as high as TORPA's score: 87 termination proofs for AProVE and 88 for TORPA.

TORPA is freely available in two versions:

- A full executable version written in Delphi with a graphical user interface, including facilities for editing SRSs. This runs directly in any Windows environment.
- A command line version written in standard Pascal to be used on other platforms, or for running batches. This version was used for the competition. Also a LINUX executable is available.

Downloading is done from

`http://www.win.tue.nl/~hzantema/torpa.html`

where also some more detailed information is given. The present version is version 1.3. In the earlier version 1.1 (October 2003) RFC-match-boundedness was not implemented. Version 1.2 (January 2004) was the first one including RFC-match-boundedness. In version 1.3 (May 2004) the techniques were improved slightly, and facilities for proving non-termination were added.

The structure of this paper is as follows. First we give preliminaries of string rewriting and relative termination. Then in five consecutive sections we discuss each of the basic techniques. Next we describe how a search for non-termination is executed. In the final section we give conclusions.

To distinguish text generated by TORPA from the text of the paper the text generated by TORPA is always given in `typewriter font`.

2 Preliminaries

A string rewrite system (SRS) over an alphabet Σ is a set $R \subseteq \Sigma^+ \times \Sigma^*$. Elements $(l, r) \in R$ are called *rules* and are written as $l \to r$; l is called the left hand side (lhs) and r is called the right hand side (rhs) of the rule. In TORPA format the arrow $\to$ is written by the two symbols `->`. A string $s \in \Sigma^*$ rewrites to a string $t \in \Sigma^*$ with respect to an SRS R, written as $s \to_R t$ if strings $u, v \in \Sigma^*$ and a rule $l \to r \in R$ exist such that $s = ulv$ and $t = urv$.

An SRS R is called *terminating* (strongly normalizing, $\mathsf{SN}(R)$) if no infinite sequence $t_1, t_2, t_3, \ldots$ exists such that $t_i \to_R t_{i+1}$ for all $i = 1, 2, 3, \ldots$. An SRS R is called *terminating relative to* an SRS S, written as $\mathsf{SN}(R/S)$, if no infinite sequence $t_1, t_2, t_3, \ldots$ exists such that

- $t_i \to_{R \cup S} t_{i+1}$ for all $i = 1, 2, 3, \ldots$, and
- $t_i \to_R t_{i+1}$ for infinitely many values of i.

The notation R/S is also used for the rewrite relation $\to_S^* \cdot \to_R \cdot \to_S^*$; clearly $\mathsf{SN}(R/S)$ coincides with termination of this rewrite relation. By definition $\mathsf{SN}(R/S)$ and $\mathsf{SN}(R/(S \setminus R))$ are equivalent. Therefore we will use the notation $\mathsf{SN}(R/S)$ only for R and S being disjoint. In writing an SRS $R \cup S$ for which we want to prove $\mathsf{SN}(R/S)$ we write the rules of R by $l \to r$ and the rules of S by $l \to= r$. In TORPA format the arrow $\to=$ is written by the three symbols `->=`. The rules from R are called *strict* rules; the rules from S are called *non-strict* rules. Clearly $\mathsf{SN}(R/\emptyset)$ and $\mathsf{SN}(R)$ coincide.

Our first theorem is very fruitful for stepwise proving (relative) termination.

Theorem 1 *Let R, S, R' and S' be SRSs for which*

- *$R \cup S = R' \cup S'$ and $R \cap S = R' \cap S' = \emptyset$,*
- *$\mathsf{SN}(R'/S')$ and $\mathsf{SN}((R \cap S')/(S \cap S'))$.*

Then $\mathsf{SN}(R/S)$.

Theorem 1 is applied in TORPA as follows. In trying to prove $\mathsf{SN}(R/S)$ it is tried to split up $R \cup S$ into two disjoint parts R' and S' for which $R' \neq \emptyset$ and $\mathsf{SN}(R'/S')$. If this succeeds then the proof obligation $\mathsf{SN}(R/S)$ is weakened to $\mathsf{SN}((R \cap S')/(S \cap S'))$, i.e., all rules from R' are removed. This process is repeated as long as it is applicable. If after a number of steps $R \cap S' = \emptyset$ then $\mathsf{SN}((R \cap S')/(S \cap S'))$ trivially holds and the desired proof has been given.

Next we consider reversing strings. For a string s write s^{rev} for its reverse. For an SRS R write $R^{\mathsf{rev}} = \{\ l^{\mathsf{rev}} \to r^{\mathsf{rev}} \mid l \to r \ \in \ R\ \}$.

Lemma 2 *Let R and S be disjoint SRSs. Then $\mathsf{SN}(R/S)$ if and only if $\mathsf{SN}(R^{\mathsf{rev}}/S^{\mathsf{rev}})$.*

Lemma 2 is strongly used in TORPA: if $\mathsf{SN}(R/S)$ has to be proved then all techniques are not only applied on R/S but also on $R^{\mathsf{rev}}/S^{\mathsf{rev}}$.

3 Polynomial Interpretations

The ideas of (polynomial) interpretations go back to [9, 2]. First we give the underlying theory for doing this for string rewriting.

Let A be a non-empty set and Σ be an alphabet. Let ϵ denote the empty string in Σ^*. If $f_a : A \to A$ has been defined for every $a \in \Sigma$ then $f_s : A \to A$ is defined for every $s \in \Sigma^*$ inductively as follows:

$$f_\epsilon(x) = x, \quad f_{as}(x) = f_a(f_s(x)), \quad \text{for every } x \in A, a \in \Sigma, s \in \Sigma^*.$$

Theorem 3 *Let A be a non-empty set and let $>$ be a well-founded order on A. Let $f_a : A \to A$ be strictly monotone for every $a \in \Sigma$, i.e., $f_a(x) > f_a(y)$ for every $x, y \in A$ satisfying $x > y$. Let R and S be disjoint SRSs over Σ such that $f_l(x) > f_r(x)$ for all $x \in A$ and $l \to r \in R$, and $f_l(x) \geq f_r(x)$ for all $x \in A$ and $l \to r \in S$. Then $\mathsf{SN}(R/S)$.*

In the general case this approach is called *monotone algebras* ([11, 13]). In case A consists of all integers $> N$ with the usual order for some number N, and the functions f_a are polynomials this approach is called *polynomial interpretations*.

In TORPA only three distinct polynomials are used: the identity, the successor $\lambda x \cdot x + 1$, and $\lambda x \cdot 10x$. For every symbol a one of these three polynomials is chosen, and then it is checked whether using Theorem 3 gives rise to $\mathsf{SN}(R/S)$ for some non-empty R, where R consists of the rules for which '$>$' is obtained. If so, then by using Theorem 1 the proof obligation is weakened and the process is repeated until no rules remain, or only non-strict rules. As a first example consider the two rules $ab \to ba, a \to= ca$, i.e., $\mathsf{SN}(R/S)$ has to be proved where R consists of the rule $ab \to ba$ and S consists of the rule $a \to ca$. Now TORPA yields:

```
Choose polynomial interpretation:
a: lambda x.10x,  b: lambda x.x+1,  rest identity
remove: ab -> ba
Relatively terminating since no strict rules remain.
```

Here for f_a and f_b respectively $\lambda x \cdot 10x$ and the successor are chosen. Since $f_{ab}(x) = f_a(f_b(x)) = 10(x+1) > 10x + 1 = f_b(f_a(x)) = f_{ba}(x)$ for every x indeed the first rule may be removed due to Theorem 3, and relative termination may be concluded due to Theorem 1.

Checking whether $f_l(x) > f_r(x)$ or $f_l(x) \geq f_r(x)$ for all x for some rule $l \to r$ is easily done due to our restriction to linear polynomials. However, for n distinct symbols there are 3^n candidate interpretations, which can be too big. Therefore a selection is made: only choices are made for which at least $n - 2$ symbols have the same interpretations. In this way the number of candidates is quadratic in n. Attempts to prove (relative) termination are done both for the given SRS and its reverse. For instance, for the single rule $ab \to baa$ no polynomial interpretation is possible (not even an interpretation in $\mathbf{N}$ as is shown in [11]), but for its reverse TORPA easily finds one. TORPA applied on the three rules $a \to fb$, $bd \to cdf$, $dc \to adfd$ yields

```
Reverse every lhs and rhs of the system and choose polynomial
interpretation:    f: identity, d: lambda x.x+1, rest lambda x.10x
remove: dc -> adfd
Choose polynomial interpretation   a: lambda x.x+1,  rest identity
remove: a -> fb
Choose polynomial interpretation   b: lambda x.x+1,  rest identity
remove: bd -> cdf
Terminating since no rules remain.
```

4 Recursive Path Order

Recursive path order is an old technique too; it was introduced by Dershowitz [4]. Restricted to string rewriting it means that for a fixed order $>$ on the finite alphabet Σ, called the *precedence*, there is an order $>_{rpo}$ on Σ^* called *recursive path order*. The main property of this order is that if $l >_{rpo} r$ for all rules $l \to r$ of an SRS R, then R is terminating. This order $>_{rpo}$ has the following defining property: $s >_{rpo} t$ if and only if s can be written as $s = as'$ for $a \in \Sigma$, and either

- $s' = t$ or $s' >_{rpo} t$, or
- t can be written as $t = bt'$ for $b \in \Sigma$, and either
  - $a > b$ and $s >_{rpo} t'$, or
  - $a = b$ and $s' >_{rpo} t'$.

For further details we refer to [13]. To avoid branching in the search for a valid precedence in TORPA a slightly weaker version is used. On the other hand, the basic order is also used in combination with removing symbols and reversing. For details see [14]. As an example we give TORPA's result on the single rule $abc \to bacb$:

```
Terminating by recursive path order with precedence:   a>b  b>c
```

5 Dependency Pairs

The technique of dependency pairs was introduced in [1] and is extremely useful for automatically proving termination of term rewriting. Here we only use a mild version of it, without explicitly doing argument filtering or dependency graph approximation. It turns out that often the same reduction of the problem caused by these more involved parts of the dependency pair technique is done by applying our versions of labelling and polynomial interpretations.

For an SRS R over an alphabet Σ let Σ_D be the set of *defined symbols* of R, i.e., the set of symbols occurring as the leftmost symbol of the left hand side of a rule in R. For every defined symbol $a \in \Sigma_D$ we introduce a fresh symbol $\overline{a}$. TORPA follows the convention that if a is a lowercase symbol then its capital version is used as the notation for $\overline{a}$. Write $\overline{\Sigma} = \Sigma \cup \{\overline{a} \mid a \in \Sigma_D\}$. The SRS $DP(R)$ over $\overline{\Sigma}$ is defined to consist of all rules of the shape $\overline{a}l' \to \overline{b}r''$ for which $al' = l$ and $r = r'br''$ for some rule $l \to r$ in R and $a, b \in \Sigma_D$. Rules of $DP(R)$ are called *dependency pairs*. Now the main theorem of dependency pairs reads as follows.

Theorem 4 *Let R be any SRS. Then $\mathsf{SN}(R)$ if and only if $\mathsf{SN}(DP(R)/R)$.*

It is used in TORPA as follows: if proving $\mathsf{SN}(R)$ does not succeed by the earlier techniques, then the same techniques are applied for trying to prove $\mathsf{SN}(DP(R)/R)$ or $\mathsf{SN}(DP(R^{\mathsf{rev}})/R^{\mathsf{rev}})$. In fact the desire for being able to do so was one of the main reasons to generalize the basic methods to relative termination and design TORPA to cover relative termination.

Applying TORPA on the two rules $ab \to c, c \to ba$ yields:

```
Dependency pair transformation:
  ab ->= c
  c ->= ba
  Ab -> C
  C -> A
```

followed by a simple proof by polynomial interpretations.

6 RFC-match-bounds

A recent very elegant and powerful approach for proving termination of string rewriting is given in [6]. The strongest version is proving match bounds of right hand sides of forward closures, shortly RFC-match-bounds. Here we present the main theorem as it is used in TORPA; for the proof and further details we refer to [6]. For an SRS R over an alphabet Σ we define the SRS $R_{\#}$ over $\Sigma \cup \{\#\}$ by $R_{\#} = R \cup \{\, l_1\# \to r \mid l \to r \in R \wedge l = l_1 l_2 \wedge l_1 \neq \epsilon \neq l_2 \,\}$. For an SRS R over an alphabet Σ we define the infinite SRS $\mathsf{match}(R)$ over $\Sigma \times \mathbf{N}$ to consist of all rules $(a_1, n_1) \cdots (a_p, n_p) \to (b_1, m_1) \cdots (b_q, m_q)$ for which $a_1 \cdots a_p \to b_1 \cdots b_q \in R$ and $m_i = 1 + \min_{j=1,\ldots,p} n_j$ for all $i = 1, \ldots, q$.

Theorem 5 *Let R be an SRS and let $N \in \mathbf{N}$ such that for all rhs's $b_1 \cdots b_q$ of R and all $k \in \mathbf{N}$ and all reductions*

$$(b_1, 0) \cdots (b_q, 0)(\#, 0)^k \;\to^*_{\mathsf{match}(R_{\#})}\; (c_1, n_1) \cdots (c_r, n_r)$$

it holds that $n_i \leq N$ for all $i = 1, \ldots, r$. Then R is terminating.

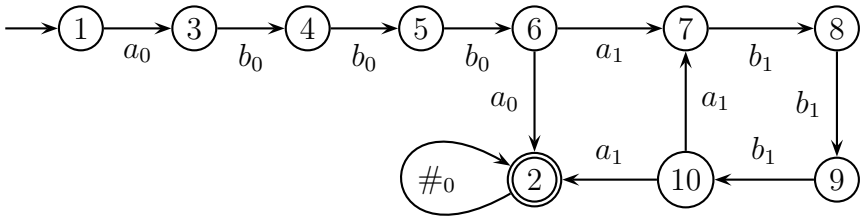
The minimal number N satisfying the condition in Theorem 5 is called the corresponding *match bound*. The way to verify the condition of Theorem 5 is to construct a *certificate*, being a finite automaton M over the alphabet $(\Sigma \cup \{\#\}) \times \mathbf{N}$, where Σ is the alphabet of R, satisfying:

- for every rhs $b_1 \cdots b_q$ of R and every $k \in \mathbf{N}$ the automaton M accepts $(b_1, 0) \cdots (b_q, 0)(\#, 0)^k$, and
- M is closed under $\mathsf{match}(R_{\#})$, i.e., if M accepts v and $v \to_{\mathsf{match}(R_{\#})} u$ then M accepts u too.

The pair $(a, k) \in (\Sigma \cup \{\#\}) \times \mathbf{N}$ will shortly be written as a_k, and the number k is called the *label* of this pair. It is easy to see that if a (finite) certificate M

has been found then for N being the biggest label occurring in M the condition of Theorem 5 holds. Hence the only thing to be done for proving termination by this approach is finding a certificate. All of these observations are found in [6]; the only new contribution of TORPA is the much more powerful heuristic of searching for such a certificate.

As an example we consider the single rule $aba \to abbba$. None of the earlier techniques works, but TORPA yields the following certificate:



Rather than giving such a picture TORPA yields a list of all transitions, containing all information about the automaton. Indeed this automaton is a certificate: it accepts $a_0b_0^3a_0\#_0^k$ for every k, and it is closed under $\mathsf{match}(R_\#)$, hence proving termination. For instance, it accepts $a_0b_0^3a_0\#_0$ which rewrites to $a_0b_0^3a_1b_1^3a_1$ by the rule $a_0\#_0 \to a_1b_1^3a_1$ of $\mathsf{match}(R_\#)$, also accepted by the automaton.

TORPA tries to construct a certificate if the earlier techniques fail, both for the SRS and its reverse. This construction starts by an automaton exactly accepting $(b_1, 0)\cdots(b_q, 0)(\#, 0)^k$ for every rhs $b_1\cdots b_q$ of R and every $k \in \mathbf{N}$. Then for every path in the automaton labelled by a lhs of $\mathsf{match}(R_\#)$ it is checked whether there exists a path between the same two nodes labelled by the corresponding rhs. If not, then such a path has to be constructed. Here the heuristic comes in. If a path from n_1 to n_2 has to be constructed labelled by the string au for $a \in (\Sigma \cup \{\#\}) \times \mathbf{N}$ and $u \in ((\Sigma \cup \{\#\}) \times \mathbf{N})^*$, then it is checked whether a node n exists for which a path from n to n_2 exists labelled by u. If so, then an edge from n_1 to n is added labelled by a, if not, then a completely fresh path from n_1 to n_2 is constructed labelled by au. This process is repeated until either no edges need to be added or overflow occurs. In the first case the resulting automaton is a certificate by construction, proving termination. Overflow occurs if the automaton contains 800 edges.

In the above example nodes 7, 8, 9, 10 are added for making a path from 6 to 2 labelled by $a_1b_1^3a_1$. Then the edge from 10 to 7 is added for making a path from 10 to 2 labelled by $a_1b_1^3a_1$, and then nothing is to be added any more. This very simple heuristic was found after trying many other and more involved heuristics that turned out to be much less powerful.

Termination of the single rule $aabb \to bbbaaa$ is easily proved by this approach, yielding exactly the same automaton as given in [6] having 42 nodes and match bound 4. However, there it was found after an extensive process on intermediate automata of thousands of nodes, while in our approach no intermediate automata exceeding the final result occurred. The main difference is that in [6] an exact

automaton is computed while TORPA computes an approximation. However, for nearly all examples both automata coincide.

7 Semantic Labelling

The technique of semantic labelling was introduced in [12]. Here we restrict to the version for string rewriting in which every symbol is labelled by the value of its argument. For this version we present the theory for relative termination; TORPA only applies this for (quasi-)models containing only two elements. This approach grew out from [7]. In fact this was how TORPA started: as a tool for semantic labelling. In the first versions indeed semantic labelling was the core of TORPA. However, due to the power of RFC-match-bounds and the laborious nature of labelling, in the present version 1.3 it was chosen to apply semantic labelling only if the other techniques fail. First we summarize the theory.

Fix a non-empty set A and maps $f_a : A \to A$ for all $a \in \Sigma$ for some alphabet Σ. Let f_s for $s \in \Sigma^*$ be defined as before. Let $\overline{\Sigma}$ be the alphabet consisting of the symbols a_x for $a \in \Sigma$ and $x \in A$. The *labelling function* $\mathsf{lab} : \Sigma^* \times A \to \overline{\Sigma}^*$ is defined inductively as follows:

$$\mathsf{lab}(\epsilon, x) = \epsilon, \quad \mathsf{lab}(sa, x) = \mathsf{lab}(s, f_a(x))a_x, \qquad \text{for } s \in \Sigma^*, a \in \Sigma, x \in A.$$

For an SRS R define $\mathsf{lab}(R) = \{\ \mathsf{lab}(l, x) \to \mathsf{lab}(r, x) \mid l \to r \in R, x \in A\ \}$.

Theorem 6 *Let R and S be two disjoint SRSs over an alphabet Σ. Let $>$ be a well-founded order on a non-empty set A. Let $f_a : A \to A$ be defined for all $a \in \Sigma$ such that*

- *$f_a(x) \geq f_a(y)$ for all $a \in \Sigma, x, y \in A$ satisfying $x > y$, and*
- *$f_l(x) \geq f_r(x)$ for all $l \to r \in R \cup S, x \in A$.*

Let Dec be the SRS over $\overline{\Sigma}$ consisting of the rules $a_x \to a_y$ for all $a \in \Sigma, x, y \in A$ satisfying $x > y$. Then $\mathsf{SN}(R/S)$ if and only if $\mathsf{SN}(\mathsf{lab}(R)/(\mathsf{lab}(S) \cup \mathsf{Dec}))$.

In case the relation $>$ is empty the set A together with the functions f_a for $a \in \Sigma$ is called a *model* for the SRS, otherwise it is called a *quasi-model*. It is called a model since then for every rule $l \to r$ the interpretation f_l of l is equal to the interpretation f_r of r. Note that $\mathsf{Dec} = \emptyset$ in case of a model. On the four rules $aal \to laa, raa \to aar, bl \to bar, rb \to lb$ TORPA may yield:

```
Apply labelling with the following interpretation in {0,1}:
  a: lambda x.1-x
  l: constant 1
  r: constant 1
  b: constant 1
and label every symbol by the value of its argument.

This interpretation is a model.

Labelled system:
   a0  a1  l0   -> l0  a1  a0
   a0  a1  l1   -> l1  a0  a1
   r0  a1  a0   -> a0  a1  r0
   r1  a0  a1   -> a0  a1  r1
   b1  l0   -> b0  a1  r0
   b1  l1   -> b0  a1  r1
   r1  b0   -> l1  b0
   r1  b1   -> l1  b1

Choose polynomial interpretation   b1 : lambda x.x+1,  rest identity
remove: b1  l0   -> b0  a1  r0
remove: b1  l1   -> b0  a1  r1
Choose polynomial interpretation   r1 : lambda x.x+1,  rest identity
remove: r1  b0   -> l1  b0
remove: r1  b1   -> l1  b1
Choose polynomial interpretation:
r1 : lambda x.10x,  rest lambda x.x+1
remove: r1  a0  a1   -> a0  a1  r1
Choose polynomial interpretation:
a1 : lambda x.10x,  rest lambda x.x+1
remove: a0  a1  l1   -> l1  a0  a1
Choose polynomial interpretation:
r0 : lambda x.10x,  rest lambda x.x+1
remove: r0  a1  a0   -> a0  a1  r0
Choose polynomial interpretation:
a0 : lambda x.10x,  rest lambda x.x+1
remove: a0  a1  l0   -> l0  a1  a0
Terminating since no rules remain.
```

by which both rules are removed. In the notation of Theorem 6 this means that $A = \{0,1\}$, $f_a(x) = 1 - x$ and $f_b(x) = f_l(x) = f_r(x) = 1$ for $x \in A$, $R = \{aal \to laa, raa \to aar, bl \to bar, rb \to lb\}$, $S = \mathsf{lab}(S) = \mathsf{Dec} = \emptyset$. Since $\mathsf{lab}(aal, x) = a_0 a_1 l_x$ and $\mathsf{lab}(laa, x) = l_x a_{1-x} a_x$ for $x = 0, 1$, the first two rules of the labelled system $\mathsf{lab}(R)$ are as indicated. As is shown, the termination proof is given by proving termination of $\mathsf{lab}(R)$ by simple polynomial interpretations.

Now we describe how such a proof is found by TORPA. For $A = \{0,1\}$ for every symbol a there are four possibilities for $f_a : A \to A$: $f_a = \lambda x \cdot x$, $f_a = \lambda x \cdot 0$, $f_a = \lambda x \cdot 1$, $f_a = \lambda x \cdot 1 - x$. Up to renaming $A = \{0,1\}$ admits only two strict orders $>$: $> = \emptyset$ and $> = (1,0)$. For the first one (the model case) for all symbols a all four interpretations for f_a are allowed, and the only restriction is

that $f_l = f_r$ for all rules $l \to r \in R \cup S$. For the second order (the quasi-model case) for all symbols a only the first three interpretations for f_a are allowed, since $f_a = \lambda x \cdot 1 - x$ does not satisfy the requirement that $f_a(x) \geq f_a(y)$ for $x > y$. On the other hand, now the restriction on the rules is weaker: rather than $f_l(x) = f_r(x)$ it is only required that $f_l(x) \geq f_r(x)$ for all rules $l \to r \in R \cup S$ and $x \in A$.

In TORPA first the model approach is tried for random choices of the functions f_a until the model requirements hold. Then polynomial interpretations and recursive path order are applied on the labelled systems. If this succeeds the desired proof is generated, otherwise the whole procedure is repeated. There is a basic maximal number of attempts to be done. The default of this number is 100. Subsequent attempts to prove termination by TORPA may yield different solutions, due to the use of the random generator.

In case this first series of attempts was not yet successful a similar procedure is applied for quasi-models. For both the model case and the quasi-model case everything is done twice while no solution is found: once for R/S and once for $R^{\mathsf{rev}}/S^{\mathsf{rev}}$. On the four rules $a \to bc$, $ab \to ba$, $dc \to da$, $ac \to ca$ TORPA yields:

```
Reverse every lhs and rhs of the system.
Apply labelling with the following interpretation in {0,1}:
  a: identity             c: identity
  b: constant 0           d: constant 1
and label every symbol by the value of its argument.
This is a quasi-model for  1 > 0.
```

and for the resulting labelled system a termination proof is given by polynomial interpretations. We do not know any other way to prove termination of this SRS.

In case some attempt to prove termination of a labelled system fails, but applying polynomial interpretations succeeds in removing some rules, then in the default setting of TORPA it is checked whether after removing all labels a strict subset of the original SRS is obtained. If so, then the whole procedure starts again on this smaller SRS. In this way the techniques in TORPA may be combined in complicated termination proofs.

As an example we consider the SRS consisting of the five rules

$$\begin{array}{lcl} f0 & \to & s0 \\ d0 & \to & 0 \\ ds & \to & ssdps \\ fs & \to & dfps \\ ps & \to & e \end{array}$$

in which s, d and f describe successor, doubling and the exponential function, respectively. For this SRS TORPA proceeds as follows. First the first rule is removed by choosing a polynomial interpretation. Then a labelling is found, by which after removing rules by polynomial interpretations and next removing all labels only the last three rules of the original system remain. Next, the

dependency pair transformation is applied on the reversed system of these three rules. For this system a quasi-model labelling is found. After removing rules by polynomial interpretations and again removing all labels, a system is found for which the ultimate termination proof is generated by finding a labelling for the third time. Finding the full proof requires one or two seconds.

8 Detecting non-termination

In case the search for a termination is not successful, in TORPA it is tried to find a proof for non-termination. This is done by generating a directed graph in which the nodes are labelled by strings in such a way that only an edge is allowed from a node labelled by a string u to a node labelled by a string u' if u rewrites to $u_1u'u_2$ for (possibly empty) strings u_1, u_2. It is easy to see that if this graph is cyclic then a infinite derivation of the following shape exists:

$$u \to^+ vuw \to^+ vvuww \to^+ vvvuwww \to^+ \cdots.$$

Such a derivation is called *looping*. TORPA generates such a graph of limited size and then checks whether this graph is cyclic using the same algorithm as was used for recursive path order. Details of this approach and comparison with other approaches are matter of current research.

As an example we apply TORPA on the single rule $ab \to bbaa$:

```
Non-terminating; looping derivation starting in: aab
```

Indeed, for $u = aab$ a looping derivation of the above shape is found in which $v = bb$ and $w = aa$:

$$u = aab \to abbaa \to bbaabaa = vuw.$$

9 Conclusions

For many small SRSs TORPA automatically finds a termination proof, but finding a human proof allowing any presently known technique seems to be a really hard job. Usually, the generated proofs are not more than a few pages of text, including many details. For people familiar with the underlying theory verifying the generated proofs is always feasible. However, this may be very boring, and redundant since the proofs are correct by construction.

Due to the extension in January 2004 by RFC-match-bounds, the present version 1.3 is much stronger than the earlier version 1.1 described in [14].

Most techniques used in TORPA also apply for term rewriting rather than string rewriting. Hence a natural follow up will be a version of TORPA capable of proving termination of term rewriting.

References

[1] T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.

[2] A. Ben-Cherifa and P. Lescanne. Termination of rewriting systems by polynomial interpretations and its implementation. *Science of Computer Programming*, 9:137–159, 1987.

[3] E. Contejean, C. Marché, B. Monate, and X. Urbain. The CiME rewrite tool. Available at `http://cime.lri.fr/`.

[4] N. Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17:279–301, 1982.

[5] J. Giesl et al. Automated program verification environment (AProVE). Available at `http://www-i2.informatik.rwth-aachen.de/AProVE/`.

[6] A. Geser, D. Hofbauer, and J. Waldmann. Match-bounded string rewriting. Technical Report 2003-09, National Institute of Aerospace, Hampton, VA, 2003. Submitted for publication in a journal.

[7] J. Giesl and H. Zantema. Liveness in rewriting. In R. Nieuwenhuis, editor, *Proceedings of the 14th Conference on Rewriting Techniques and Applications (RTA)*, volume 2706 of *Lecture Notes in Computer Science*, pages 321–336. Springer, 2003.

[8] N. Hirokawa and A. Middeldorp. Tsukuba termination tool. In R. Nieuwenhuis, editor, *Proceedings of the 14th Conference on Rewriting Techniques and Applications (RTA)*, volume 2706 of *Lecture Notes in Computer Science*, pages 311–320, 2003.

[9] D.S. Lankford. On proving term rewriting systems are noetherian. Technical report MTP 3, Louisiana Technical University, 1979.

[10] J. Waldmann. Matchbox: a tool for match-bounded string rewriting. In V. van Oostrom, editor, *Proceedings of the 15th Conference on Rewriting Techniques and Applications (RTA)*, volume 3091 of *Lecture Notes in Computer Science*, pages 85–94. Springer, 2004.

[11] H. Zantema. Termination of term rewriting: Interpretation and type elimination. *Journal of Symbolic Computation*, 17:23–50, 1994.

[12] H. Zantema. Termination of term rewriting by semantic labelling. *Fundamenta Informaticae*, 24:89–105, 1995.

[13] H. Zantema. Termination. In *Term Rewriting Systems, by Terese*, pages 181–259. Cambridge University Press, 2003.

[14] H. Zantema. Termination of string rewriting proved automatically. Technical Report CS-report 03-14, Eindhoven University of Technology, 2003. Submitted, available via `http://www.win.tue.nl/inf/onderzoek/en_index.html` .

[15] H. Zantema. TORPA: termination of rewriting proved automatically. In V. van Oostrom, editor, *Proceedings of the 15th Conference on Rewriting Techniques and Applications (RTA)*, volume 3091 of *Lecture Notes in Computer Science*, pages 95–104. Springer, 2004.

[16] H. Zantema and A. Geser. A complete characterization of termination of $0^p1^q \to 1^r0^s$. *Applicable Algebra in Engineering, Communication and Computing*, 11(1):1–25, 2000.