Formal Transformational Reasoning about Reactive Systems in the Theorem Prover LAMBDA*

Jürgen Bohn †
University of Oldenburg, Computer Science Department
PB 2503, 26111 Oldenburg, Germany

August 15, 1994

Abstract

Computer aided transformational reasoning about reactive systems requires the formalization of a suitable specification language SL, its semantics, and transformation rules for refining specifications stepwise into implementations. This paper presents the embedding of a sophisticated semantic model in the higher order logic of LAMBDA[FFHM92]. The model originates from the ESPRIT Basic Research Action "ProCoS" (Provably Correct Systems) [Bjø89]. It allows a detailed description of the reactive behaviour of synchronously communicating systems. The embedding in a theorem prover makes it possible to prove semantic system properties. Rules for the transformation of syntactically given specifications are formally proved correct in the semantics. The application of such rules is demonstrated by a case study. Here we present the stepwise transformation of a shared register specification into a system of distributed, synchronously communicating subsystems, which logically imply the behaviour of the original specification. Some work is done on the design of automatic tactics, which verify application conditions of transformation rules. The necessary interaction during the system design is reduced nearly to the point that one transformation step coincides with one tactical rule application.

1 Introduction

We present an embedding of the transformational approach to the design of reactive systems as developed in the ProCoS project into the higher order logic of LAMBDA. The language supported by LAMBDA is very similar to ML, a functional language of higher order with polymorphic types, but extended by the standard operators of higher order logic. The system offers some support for formalizations by automatically generating rule sets from much shorter descriptions of data types and functions. It has a graphical interface, provides a browser over rule sets, mouse driven windows for selection and manipulation of expressions, and a generic pretty printer. This text requires no special knowledge of LAMBDA, so we give no introduction to the system. For more information on LAMBDA itself see [FFHM92], [MF91].

There is no special type for sets in LAMBDA. We have implemented sets, as usual, as characteristic functions, but use a function *set: 'a list* $\rightarrow$ *('a* $\rightarrow$ *{TRUE,FALSE})*, which maps lists to sets, for concrete given sets. Of course, we have formalized all operators on

*This research was partially supported by the German Ministry of Research and Technologies (BMFT) as part of the project KORSO (Korrekte Software) under grant no. 01 IS 203 N.

†E-mail: Juergen.Bohn@Informatik.Uni-Oldenburg.DE

sets, but pretty print some of them to improve readability. E.g. $(a \cup b)$ stands for `unS(a, b)` and $(a \subseteq b)$ for `subS(a, b)`. We have updated LAMBDA's pretty printer for our specification language SL, but the specifications presented below are also slightly pretty printed by hand.

2 Specifying synchronously communicating systems

When describing the communicative behaviour of distributed systems we specify

1. which sequences of (synchronous) communications are allowed,
2. under which conditions on the internal state of the system a communication is possible, and
3. what is the effect of a communication on the internal system state.

We use regular expressions over channel names for the description of all possible communication sequences. As an example think of a register that stores one value and may be accessed by a single writer and several readers. The register communicates through directed channels with its environment. It gets input from the writer over a channel W and returns an acknowledgement signal on a channel A. For every reader it has an output channel T_i and a request channel R_i.

The register example is introduced by L. Lamport as case study for interprocess communication through shared registers [Lam86]. A detailed study of this example in our transformational approach can be found in [OR93].

The interface of the register is built by the mentioned directed channels. Allowable communication sequences require that every input on W is followed by an acknowledgement and every request on R_i is answered on T_i. This is specified together with the interface by the following *trace assertions*

$$
\begin{array}{ll}
\text{spec} & \text{input } W \text{ of } V \\
& \text{output } A \text{ of } \mathit{Signal} \\
& \text{input } R_1, \ldots, R_n \text{ of } \mathit{Signal} \\
& \text{output } T_1, \ldots, T_n \text{ of } V \\
& \\
& \text{trace } \{W,A\} \text{ in } \mathit{pref(W.A)^*} \\
i \in 1..n : & \text{trace } \{R_i, T_i\} \text{ in } \mathit{pref(R_i.T_i)^*}
\end{array}
$$

V is a variable for any type of values. There are no additional restrictions on the enabling of communications. Independent from the internal register state communications may happen in sequences given by the trace assertions. So it remains to describe the effect of each single communication. In our specification language SL this is done by *communication assertions* of the form

$$\text{com } ch \text{ read } RL \text{ write } WL \text{ when } enable \text{ then } effect.$$

A communication on channel *ch* is allowed, if the enable predicate is fulfilled in the current state. The effect of a communications to the (local and global) system variables is specified in the effect predicate. The symbol $@ch$ refers to the communicated value, v to the initial value of the variable v and v' to the final value of v, i.e. its value after the communication. All variables used in the enable and effect predicate have to be declared with the correct access mode in the read list RL and the write list WL of variable names.

Trivial components such as the enable condition *true* in the register specification are omitted. Here we introduce a local variable v which stores the register value. A Boolean variable m indicates whether the register is in a write phase. For the return values to the readers we require at least, that the content of the register is given, if no write phase overlaps the phase between a reader request and its answer. In the case of overlapping reading and writing phases nothing is said about the returned value.

$$
\begin{array}{rll}
 & \text{var } v \text{ of } V & \\
 & \text{var } m, c_1, \ldots, c_n \text{ of } Bool & \\
 & \text{com } W & \text{write } \{v, m, c_1, \ldots, c_n\} \\
 & & \text{then } m' \wedge \ v' = @W \wedge \ \bigwedge_{i=1}^{n} \neg c_i' \\
 & \text{com } A & \text{write } \{m\} \\
 & & \text{then } \neg m' \\
i \in 1..n : & \text{com } R_i & \text{write } \{c_i\} \text{ read } \{m\} \\
 & & \text{then } c_i' = \neg m \\
i \in 1..n : & \text{com } T_i & \text{read } \{c_i, v\} \\
 & & \text{then } c_i \Rightarrow \ (@T_i = v)
\end{array}
$$

The variables c_i indicate, whether the register has to return the correct value to a reader. If a write phase starts (with a communication on W) then m is set to *true*, the received value is stored in v and all c_i are set to *false*. After the writing is acknowledged (via A) the variable m is set *false*. A request on R_i initializes the variable c_i. If c_i is true until the end of a reading phase, then the correct register value v is sent on T_i.

3 Observable behaviour of communicating systems

Here we introduce the (logical) variables which we use to define the observable behaviour of communicating systems and the types of these variables.

A system may be ready for a communication, what we call stable. Or it may be terminated, i.e. not ready for any communication with its environment. Additionally one distinguishes the case that the system performs infinite internal communications. Then the system is called divergent. Summing up we define as global system states (where `term` abbreviates terminated and `diver` means divergent):

```
STATUS = stable | term | diver;
```

A system communicates with its environment via directed channels. We observe the sequences (lists) of synchronous communications which a system may perform in the form of traces. Each trace element is a pair of a channel name and the communicated value:

```
type TRACES = (CH_NAME * VALUE) list;
```

Each communication may change the state of the system variables. We need to describe the values of the variables before and after each communication by initial and final environments, which map variable names to values.

```
type ENV = V_NAME -> VALUE;
```

Usually a system is only ready for a communication on a subset of its channels. The readiness for a communication may depend on the values of the system variables. Therefore we observe the set of ready channels.

```
type READY_SET = CH_NAME SET;
```

The total behaviour of a synchronously communicating system is described by a behaviour predicate. This is formalized as a function in the following variables:

- `st` for the global system status
- `tr` holds the sequence of communications
- `Rdy` gives the ready set, if the system is stable
- `initial` and `final` of type `ENV` describe the internal state
- `typeOfVar` and `typeOfChan` map variable and channel names to types;

The type of such a behaviour is

```
type behaviour = STATUS * TRACES * READY_SET
                 * ENV * ENV * (V_NAME -> TYPES) * (CH_NAME -> TYPES);
```

whereas behaviour predicates are formalized as Boolean functions

```
type 'a PRED = 'a -> {TRUE,FALSE}
type behPred = behaviour PRED
```

4 Example: semantics of simple specifications

In this section we show the behaviour predicate `S_Spec` of specifications without local variables and local channels. Then some basic predicates used by `S_Spec` and properties of these predicates are shown.

A specification contains at least an interface, trace assertions and communication assertions. As behaviour of such specifications we define the following predicate:

```
fun S_Spec (D, TA ,CA) :behPred = fn (st,tr,Rdy,initial,final,tyV,tyCh) =>
      tcp D (st,tr,Rdy,initial,final,tyV,tyCh)
      ∧ st == stable
      ∧ elemTrLan(map fst tr,D,TA)
      ∧ (exists l_fin:ENV.
          TR (D,CA) (st,tr,Rdy,initial,l_fin,tyV,tyCh)
          ∧ (forall ch.  (ChanS D) ch
                ->> (Rdy ch == ((WH(ch,CA)) l_fin
                               ∧ elemTrLan((map fst tr)@[ch],D,TA) )) )
          )
```

The more informal meaning of this predicate is as follows: The specification shall fulfill some type conditions (`tcp`) on variables and channels of the interface `D`. The status of the system is stable. The trace `tr` projected to its first components (channel names) is an element of the regular language, defined by the trace assertions `TA`. There exists a local environment `l_fin`, i.e. variable values, such that a transition relation `TR` is fulfilled and for each channel `ch` in the interface `D` it holds, that `ch` is in the ready set, if and only if firstly the enable predicates for this channel are true in the environment `l_fin`, and secondly the trace extended by this channel `ch` is also an element of the defined regular language.

Remember, that sets are formalized as characteristic functions. Testing, whether `ch` is an element of `Rdy` means to apply the characteristic functions `Rdy` to `ch`. `(ChanS D)` yields a function which is true for `ch`, if `ch` is a channel in the interface `D`.

We can also specify systems, which may terminate. Specifying divergent systems makes no sense. In most cases only stable systems are developed, since it is required from a reactive systems, that it is always ready for interactions with its environment.

4.1 The type condition predicate `tcp`

We will now discuss some details. At least every behaviour predicate `P` shall fulfill for an appropriate interface `D` the type condition predicate `tcp` in our model.

```
P implies tcp(D);
```

We have proved for all interesting syntactic systems that their semantics fulfill this implication by structural induction over the constructors of systems. The type condition predicate `tcp` guarantees that

- the trace `tr` contains only channels from the interface and communicated values which respect the types of the channels;
- the ready set `Rdy` is a subset of the interface channels;
- the types of the variables and their initial and final values fit together.

The formalization of these properties is straightforward. `tcp` does not depend on the status of the system. The argument `st` is only given for compositional reasons, i.e. `tcp` applied to an interface has type `behPred`.

4.2 The transition relation `TR`

The transition relation `TR` describes the progress of the system inductively on the observed trace. It depends on the interface `D` and the specified behaviour in the communication assertions `CA`. If the trace is empty (`[]`) we require that for all interface variables the type function `tyV` is set correctly and that no observed variable changes its value, i.e. the initial and final values are equal.

```
TR (D:INTERFACE,CA:commAss) (st,[],Rdy,initial,final,tyV,tyCh) =
     forall v.  (VarS D) v
            ->> (typeOfVal(initial v) == (tyV v)) ∧ initial v == final v
```

For a communication trace starting with a communication `h` followed by `tr1` we define

```
TR (D:INTERFACE,CA:commAss) (st,(h::tr1),Rdy,initial,final,tyV,tyCh) =
    (forall v.  (VarS D) v ->> (typeOfVal(initial v) == (tyV v)))
    ∧ CommL((h::tr1),D) tyCh
    ∧ ( exists ch.  (ChanS D) ch
        ∧ (WH(ch,CA)) initial
        ∧ exists inter.
          (forall v.  (VarS D) v
                      ->> ((Wch(ch,CA)) v ∨ (inter v == initial v)))
          ∧ (exists m.  h == (ch,m) ∧ (TH(ch,CA))(m,initial,inter))
          ∧ TR (D,CA) (st,tr1,Rdy,inter,final,tyV,tyCh)
      )
```

The first two lines guarantee the correct typing of the interface variables and trace channels and communicated values. Then it is required that there exists a channel `ch` in the interface, whose enable conditions are fulfilled in the initial environment (`(WH(ch,CA)) initial`). There exists an additional intermediate environment `inter`, such that all variables are either write variables of the channel `ch` (`(Wch(ch,CA)) v`) or the intermediate and the initial value are the same. Further there exists a communicated value `m` and the communication `h` contains `m` paired with the channel. The effect predicates of `ch` are fulfilled with the communication value `m` and the mentioned initial and intermediate values of variables. Last it is required that the tail `tr1` of the trace belongs to the transition relation started in the intermediate environment instead of the initial one.

The following theorems demonstrate, how theorems in LAMBDA look like. A theorem is a sequent with a list of hypothesis separated by the $-sign, a turnstile ⊢ and one conclusion. Theorems are rules without premises. Rules contain a list of sequents as premises and one sequent as conclusion.

The definition of `TR` allows to prove some simple statements easily. For example : although `TR` is defined for compositional reasons as function in `st` and `Rdy`, it does not depend on these variables :

```
  TR (D,CA) (st,tr,Rdy,ini,fin,tyV,tyCh)
⊢ TR (D,CA) (st',tr,Rdy',ini,fin,tyV,tyCh)
```

Some properties require, that only syntactically correct specifications are used. For example all read and write variables in communication assertions have to be correctly given in the interface and there are only communication assertions for channels, which occur in the interface. The conditions are formalized in the predicate `ProperCA`.

```
fun ProperCA(D:INTERFACE,CA:commAss) =
        (caReadS CA ⊆ VarS D)
      ∧ (caWriteS CA ⊆ WriteS D) ∧ (caChanS CA ⊆ ChanS D)
```

The initial and final environments are interchangeable, if they coincide on the interface variables, and the communication assertions are proper :

```
   forall v.  VarS D v ->> initial v == ini v $ ProperCA (D,CA) $
   forall v.  VarS D v ->> final v == fin v $
   TR (D,CA) (st,tr,Rdy,initial,final,tyV,tyCh)
 ⊢ TR (D,CA) (st,tr,Rdy,ini,fin,tyV,tyCh)
```

Another proved theorem for proper communication assertions is, that all read only variables hold their values in TR.

4.3 Synchronous decomposition

Of more interest is the next result. If we want to decompose a specification into two (or more) parallel subspecifications, we have to split the transition relation.

```
   paraUn (D1,D2) $ ProperCA (D1,CA1) $ ProperCA (D2,CA2)
 ⊢ TR (D1 ∪ D2, CA1 @ CA2) (st,tr,Rdy,initial,final,tyV,tyCh) ==
   ( CommL (tr,D1 ∪ D2) tyCh
   ∧ TR (D1,CA1) (st,(tr ↓ ChanS D1),Rdy,initial,final,tyV,tyCh)
   ∧ TR (D2,CA2) (st,(tr ↓ ChanS D2),Rdy,initial,final,tyV,tyCh) )
```

paraUn formalizes that there are no common write variables in two interfaces. This allows the union of interfaces of parallel systems. (tr $\downarrow$ chs) filters all communications over a set chs of channels from a trace tr. The main result for the parallel decomposition of specifications is the following rule:

```
   paraUn(D1,D2) $ ProperCA (D1,CA1) $ ProperCA (D2,CA2) $
   ProperTA (D1,TA1) $ ProperTA (D2,TA2)
 ⊢ Beh (SPEC (D1 ∪ D2,TA1 @ TA2, CA1 @ CA2))
   == (Beh (SYN[ (SPEC (D1,TA1,CA1)), (SPEC (D2,TA2,CA2)) ]))
```

Beh computes the behaviour predicate of a system. This theorem makes it possible to prove a transformation rule on syntactic systems. As example we use a generalization for the synchronous decomposition of the register specification at the end of the text.

5 Other SL components

In the register example below we saw that specifications are not only built from an interface D, trace assertions TA and communication assertions CA. A full specification has the form

spec D TA CA LV LC SR end

where LV is a set of typed and optionally initialized local variables, LC is a set of local bidirectional typed channels and SR is a list of state restrictions. A state restriction is one of

Initial p In the system initially the predicate p holds.

Stable p If the predicate p is true in the current environment, i.e. before a communication happens, then it also holds after the communication.

Always p This is a combination of Initial and Stable, i.e. p is an invariant of the system.

The formal semantics of such specifications is reduced back to simpler basic cases. The definition of the local channels and variables can be shifted to the outside of the specification, if the interface is extended in an appropriate way. For example we define

$$\text{spec } D \;\; TA \;\; CA \;\; \textit{var m of Bool} \;\; LC \;\; SR \text{ end}$$

to be semantically the same as

$$\text{var } m \text{ of } \textit{Bool} \;\; \text{spec } D \cup \{\textit{write m of Bool}\} \;\; TA \;\; CA \;\; \emptyset \;\; LC \;\; SR \text{ end}$$

and

$$\text{spec } D \;\; TA \;\; CA \;\; LV \;\; \textit{chan ch of Signal} \;\; SR \text{ end}$$

to be defined semantically as the same as

$$\text{chan } ch \text{ of } \textit{Signal} \;\; \text{spec } D \cup \{\textit{input ch of Signal}\} \;\; TA \;\; CA \;\; LV \;\; \emptyset \;\; SR \text{ end}$$

The state restrictions are easily defined with new trace and communication assertions. For the Initial property we introduce a new local channel and a new trace assertion that ensures that every possible trace performs at first this new local channel and a new communication assertion for this channel with the stable predicate as effect predicate. The Stable property is defined only by additional communication assertions with the effect $p(initial) \rightarrow p(final)$, i.e. if p holds in the initial environment then p holds after the communication.

6 Semantics of local channels and variables

The semantics of basic specifications is defined above, so it remains to define the semantics of local channels and variables. We do this not only for specifications but for arbitrary systems.

```
fun VarP ((v,ty,init):LOCAL_VAR,sys:system):behPred =
       fn (st,tr,Rdy,initial,final,tyV,tyCh) =>
        exists l_initial,l_final.
          (forall w.  (VarS (delVarS(v,fst sys))) w
                 ->> (l_initial w == initial w ∧ l_final w == final w))
        ∧ (init == None ∨ init == Some (l_initial v))
        ∧ (snd sys)(st,tr,Rdy,l_initial,l_final,subst tyV v ty,tyCh)
```

`VarP` applied to one typed initialized variable and a system `sys` yields a behaviour predicate, i.e. a truth function in the variables *st,tr,Rdy,initial,final,tyV,tyCh*. Since we deal with the semantics, `sys` is a pair of an interface and a behaviour predicate. The meaning of the definition is easy: we have to exchange the global environments `initial` and `final` by local ones, which coincide with the global ones in all interface variables except the new local variable `v`, whose values should be hidden to the outside. `v` is either not initialized (`init == None`) or the initial value is assigned to the local initial environment `l_initial`. At last it is required that the behaviour predicate of the system holds for these new initial and final environments and the variable type function `tyV` with `ty` substituted for the type of `v`.

In the case of local channels we have to hide all local occurrences from the outside. This concerns no variables but the trace and ready set.

```
fun ChanP ((ch,ty),sys:system):behPred =
       fn (st,tr,Rdy,initial,final,tyV,tyCh) =>
        exists t.  tr == (t ↓ delS(ch,ChanS(fst sys)))
                ∧ NOT(Rdy ch)
                ∧ (snd sys)(st,t,Rdy,initial,final,tyV,subst tyCh ch ty)
```

The typed channel `ch` may be used in the system `sys` but it is hidden to the outside. Therefore the observable trace `tr` must be the local trace `t` projected to all interface channels of `sys` except `ch`. (`sys` is an (interface, behaviour predicate)-pair, `fst` yields the interface, snd the predicate.) The local channel must not occur in the viewable ready set `Rdy` and the behaviour predicate of the system `sys` must hold for the local trace `t` and channel type function `tyCh` with `ty` substituted for `ch`.

7 The satisfaction relation: combining syntax and semantics

Above we have shown how we embed the semantic model in the higher order logic. For the specification and transformation of communicating systems a syntactic system language and transformation rules on this syntax are defined.

The relation between syntax and semantics is given by the function `Sem`, which maps syntactical systems to their (interface, behaviour predicate) - pair.

```
Sem sys = (Interface sys, Beh sys);
```

The definition of `Interface` is straightforward:

```
Interface (SPEC (D, TA, CA))    = D
Interface (CHAN ((ch,ty), sys)) = delChanS(ch, Interface sys)
Interface (SYN [])              = set []
Interface (SYN (sys::tl))       = Interface sys ∪ Interface (SYN tl)
                                  ...
```

`Beh` is simply the assignment of the previously defined behaviour predicates to the syntactic systems, for example

```
Beh (SPEC (D, TA, CA))    = S_Spec (D, TA, CA)
Beh (C_SPEC (D, TA, CA, LC))
                          = Beh (CHANL ((IntfOfLC LC), SPEC (D, TA, CA)))
Beh (VC_SPEC (D, TA, CA, LV, LC))
                          = Beh (VARL (LV, (C_SPEC (D, TA, CA, LC))))
Beh (CHAN ((ch,ty),sys)) = ChanP((ch,ty),(Interface sys,Beh sys))
Beh (CHANL (chs,sys))    = ChanSetP(chs,(Interface sys,Beh sys))
```

Note that we need different system constructors for specifications with different arguments. As described above the behaviour of specifications with more components is reduced to simpler specifications.

Having the semantics it is possible to define the implementation or satisfaction relation `sat` on syntactic systems:

```
P sat Q = (Interface P == Interface Q) ∧ (Beh P implies (Beh Q))
                                       ∧ (Proper Q ->> (Proper P))
P Eq Q  = (P sat Q) ∧ (Q sat P)
        = (Interface P == Interface Q) ∧ (Beh P == (Beh Q))
                                       ∧ (Proper Q == (Proper P))
```

8 Transformation rules

All transformation rules belong to the relation `sat`. The formalized semantics allows us to prove this mechanically. In general our formalized transformation rules have the following shape:

```
   premises
----------------
⊢ A_SPEC (D1, TA1, CA1, LV1, LC1, SR1)
  sat
  A_SPEC (D, TA, CA, LV, LC, SR)
```

If the premises hold, then the first specification (with the components appended by '1') satisfies the second one. We distinguish two classes of premises. The first class specifies application conditions for the rule, such as "An introduced variable does not occur in the interface". The second class of premises describes, how the new components `D1, TA1,...` are constructed from the old ones (`D, TA,...`).

As examples for transformation rules we present the introduction of local channels and variables, the introduction of an invariant and the parallel decomposition rule. These rules are used later for the parallel decomposition of the register example.

Introduction of one local channel with trace modification:

```
⊢ (a,re) IN TA ∧ newChanTaProp(a,c,re,re1)
⊢ TA1 == (insS(c,a),re1) :: delL((a,re),TA)
⊢ LC1 == (c,ty) ∪ LC
------------------------------------------
⊢ A_SPEC(D, TA1, CA, LV, LC1, SR)
  sat
  A_SPEC(D, TA, CA, LV, LC, SR)
```

Here we choose one trace assertion `(a,re)` with alphabet `a` and regular expression `re` and replace it in `TA` by `(insS(c,a),re1)`, the alphabet extended by the channel name `c` and a new regular expression `re1`. The predicate `newChanTaProp` links the new and old regular languages. The following rule introduces one new local variable `x` with type `ty` and no initial value (`None`). `x` is not used in the old communication assertions, and all channels in the set `CS` get the right to write values to `x`:

```
⊢ NOT((caVarS CA) x)
⊢ CA1 == introWriteVar(CS,x,CA) ∧ LV1 == insS((x,ty,None),LV)
-----------------------------------------------------------
⊢ A_SPEC(D, TA, CA1, LV1, LC, SR)
  sat
  A_SPEC(D, TA, CA, LV, LC, SR)
```

Introducing a new invariant (Always p) with free variables vs:

```
⊢ (freeS p ⊆ vs) ∧ (vs ⊆ VarS D ∪ LVarS LV)
-----------------------------------------------
⊢ A_SPEC(D, TA, CA, LV, LC, (Always(vs, p))::SR)
  sat
  A_SPEC(D, TA, CA, LV, LC, SR)
```

Parallel decomposition of specifications with all components:

```
⊢ D = D1 ∪ D2
⊢ TA = TA1 ∪ TA2           ⊢ CA = CA1 ∪ CA2
⊢ LV = LV1 ∪ LV2           ⊢ LV1 ∩ LV2 = set []
⊢ LC = LC1 ∪ LC2           ⊢ SR = SR1 ∪ SR2
---------------------------------------------
⊢ SYN [ A_SPEC (D1, TA1, CA1, LV1, LC1, SR1)
        A_SPEC (D2, TA2, CA2, LV2, LC2, SR2) ]
  sat A_SPEC (D, TA, CA, LV, LC, SR)
```

9 Application of transformation rules

There are several ways to generate programs. One possibility is to start with the term "P sat spec", where spec is a concrete specification and P is a variable representing the so far unknown implementation. If an implementation is already known one can instantiate `P` with it and prove that the relation `sat` is satisfied.

In our approach stepwise transformation shall be used. During the synthesis in each step first the transitivity rule of `sat` is applied once to introduce an intermediate system `Q`, followed by an instantiation of `Q` with an appropriate system, so that afterwards a transformation rule is applicable and the validation of its application conditions, i. e. the reduction of its premises, are possible. In the easiest case after `N` repetitions of this procedure there is the following situation:

$$\frac{\vdash \texttt{P sat spec}}{\vdash \texttt{P sat spec}} \Rightarrow \frac{\begin{array}{l}\vdash \texttt{P sat Q}\\ \vdash \texttt{Q sat spec}\end{array}}{\vdash \texttt{P sat spec}} \Rightarrow \frac{\begin{array}{l}\vdash \texttt{P sat system1}\\ \vdash \texttt{system1 sat spec}\end{array}}{\vdash \texttt{P sat spec}}$$

$$\Rightarrow \frac{\vdash \texttt{P sat system1}}{\vdash \texttt{P sat spec}} \Rightarrow \cdots \Rightarrow \frac{\vdash \texttt{P sat systemN}}{\vdash \texttt{P sat spec}}$$

If `systemN` is the desired system, then substitution of `systemN` for `P` and application of the reflexivity of `sat` yields the desired theorem

$$\Rightarrow \frac{}{\vdash \texttt{systemN sat spec}}$$

10 Mechanizing transformations

In principle each transformation step requires the same procedure: Apply the transitivity law of `sat`, then a transformation rule and then perform proofs of application conditions for the transformation rule. Also the proofs of application conditions are equal in many cases for a single transformation rule in different situations. Since most rules depend only on simple syntactic conditions it is possible to write tactics which prove the conditions automatically. Tactics can be viewed as algorithms with sequences, conditionals and loops for rule applications. For example applying the transitivity rule `transSatR` of `sat` is done by

```
at (unifyTac transSatR)
```

Here `at` is a ML command, which applies a tactic to the actual goal and `unifyTac` yields a tactic, which unifies the rule argument with the actual goal. The following is a function, which takes a (transformation) rule `rl`, applies the transitivity rule (this generates two new premises) and then applies `rl` to the first of the new premises. The second of the new premises remains unchanged, `idT` does nothing but is always successful.

```
fn rl => (unifyTac transSatR) thenLT [ (unifyTac rl), idT ]
```

This combines the first two steps in the picture above. Note that the free variable `Q`, which is introduced by `transSatR`, is instantiated automatically. Backtracking allows to choose other unifications, if there are more than one. Using "*cut*" commands disables backtracking at certain points. For example

```
fn rl => (unifyTac transSatR) cutThenLT [ (unifyTac rl), idT ]
```

takes only the first possibility, that the system finds to apply `transSatR` and fails, if then the rule `rl` is not applicable, even if there would be other unifications between transSatR and the actual goal. This remain to PROLOG programs with the PROLOG "cut" command.

The aim is now to develop powerful tactics, that reduce all application conditions (premises) of certain transformation rules. These tactics tend to be large and unreadable. Their application time lies between a few seconds and several minutes, depending on the concrete given goal. An interesting point is, how far we could automate the application of transformation rules. Indeed for many rules one application coincides with one tactic application. Here the tactics depend on parameters like variable names, channel names, the position of a communication or trace assertion and similar. For example the introduction of a new local variable (see rule above) requires as parameters the name of the new variable and the names of the channels, whose communication assertions shall get the right to write the new variable. The function `stateSpaceExtTac` takes the parameters in string format and yields a tactic, that applies the transformation rule for the introduction of a new local variable and then proves the premises of this rule.

```
stateSpaceExtTac :  string -> string -> tactic
```

The following example demonstrates the effect of an application of the tactic for a nearly empty specification. If the goal is

```
⊢ x sat (A_SPEC (set [(input ch, Nat)], []     (* D, empty TA *),
                [(ch,set [],set [],wh,th)],   (* one comm.ass *)
                set [], set [], []))          (* empty LV, LC, SR *)
```

the application of

```
at (stateSpaceExtTac "xv" "set [ch]");
```

leads to

```
⊢ x sat A_SPEC (set [(input ch, Nat)], [],
               [(ch,set [],set [xv],wh,th)],      (* one write variable *)
               set [(xv,ty,None)], set [], [])    (* one local variable *)
```

Here the new variable `xv` is introduced in the set of local variables and in the set of write variables of the communication assertion of channel `ch`. The new variable `ty` is free in the sense, that it may be instantiated later in the design process. Of course it would also be possible to add a type argument to the tactic function.

For the introduction of local channels we use the following tactic generating function:

```
locChanWithTraceModTac :  int -> string -> string -> tactic
```

The first argument specifies the position of the trace assertion, which shall be modified. The second argument is the name of the new local channel and the last takes the regular expression, which replaces the one in the mentioned trace assertion. Applying this tactic leaves a premise about the regular languages. We have embedded a complete finite system of equations about equivalence of regular languages, but it is not possible to determine this equivalence with automatic rewriting. Regular expressions have no unique normal form. To overcome this problem we use an automatic procedure on the basis of the tool AMORE[JPTW90]. A result of this extern procedure is used as axiom in the system logic.

The automation is also limited in another aspect. For the implementation of enable and effect predicates we use functions, whose arguments are states (e.g. environments initial and final). In transformation rules sometimes the free variables of predicates are mentioned. The set of free variables of a predicate-function is encoded in a semantical sense as set of all variables, whose values the predicate depends on.

```
fun substP env x d = fn y => if (x == y) = TRUE then d else (env y);
fun indep (p:('a -> 'b)PRED, v:'a) =
                              forall env, d.  p env == (p(substP env v d));
fun freeS p = fn v => NOT ( indep(p, v) );
```

Unfortunately these definitions are not constructive in the sense, that one can directly compute the free variables. `freeS` is not compositional, for example it holds:

```
freeS (fn ini => ini(v) = true ∨ ini(v) = false) == (set [])
```

because the predicate is always true. So it does not depend on the value of `v`. On the other hand we have

```
(  freeS (fn ini => ini(v) = true)
 ∪ freeS (fn ini => ini(v) = false) ) == (set [v])
```

because each subpredicate depends on `v`. So proving the application condition, that the free variables of a predicate are a subset of a certain set means to guess the free variables and then to prove that the predicate indeed depends on exactly these variables.

Similar problems arise, when the application conditions of transformation rules deal with the meaning of predicates. For example strengthening the effect in a communication assertion is done by replacing the effect predicate P by a predicate Q under the condition Q `implies` P. `implies` is defined as usual as

```
p implies q = forall x.  p(x) ->> q(x).
```

This leads for effect predicates to

```
forall m, ini, fin.  p(m,ini,fin) ->> q(m,ini,fin)  ,
```

the implication must hold for all communication values m and the initial and final values of the variables. Proving such properties is trivial in most transformations, but cannot be automated for every situation, because the predicates are not decidable.

The following example demonstrates another problem that may arise from the formalization of predicates as functions, and how we can avoid it. We had to delete an (Always p) invariant from a list of invariants. This means to walk through the list and compare every invariant in the list with p in a semantical way.

A transformation rule where this occurs is *splitInvariantR*:

```
⊢ SR1 == (Initial p) :: (Stable p) :: (delL (Always p,SR))
⊢ (Always p) IN SR
----------------------------------------------------------
⊢ A_SPEC (D,TA,CA,LV,LC,SR1) sat A_SPEC (D,TA,CA,LV,LC,SR)
```

Proving the first premise requires to compare `p` with all invariants in `SR` during the computation of `delL(_)`. One can avoid this problem by a different formulation of the rule:

```
⊢ SR1 == (Initial p) :: (Stable p) :: (delLn i SR)
⊢ (Always p) == (el i SR)
----------------------------------------------------------
⊢ A_SPEC (D,TA,CA,LV,LC,SR1) sat A_SPEC (D,TA,CA,LV,LC,SR)
```

This rule has the same effect as the first formulation, but the comparison of predicates is avoided. Another advantage of the second version is, that a function `splitInvTac` in the first version has as parameter a predicate p which must be given as string in the formalized syntax

`splitInvTac :  string -> tactic`

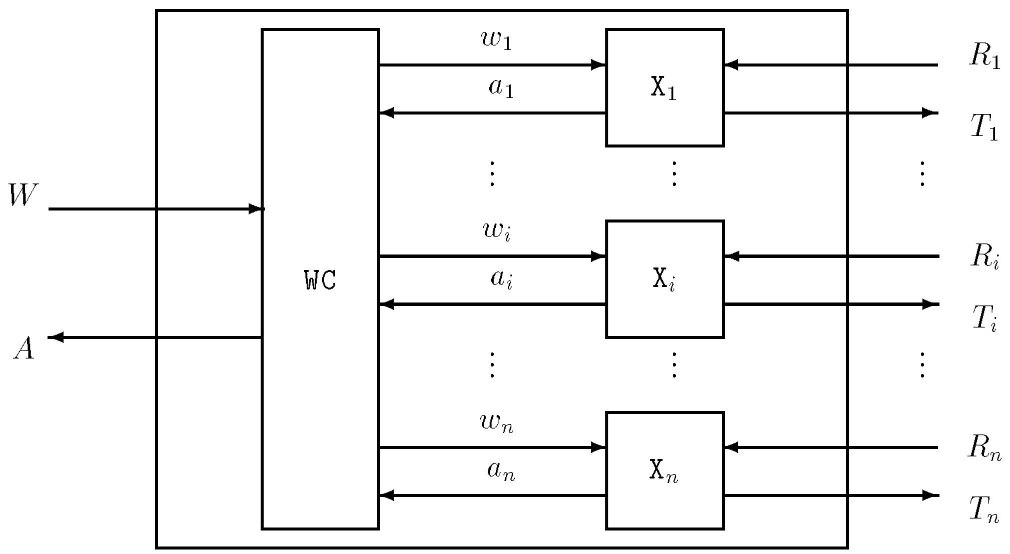
whereas the second version only requires the position of the invariant, which shall be splitted

`splitInvTac :  int -> tactic`

During the interactive work, i.e. the stepwise transformational system development, this is easier to use and faster to perform.

11 Parallel decomposition of the register specification

The following picture shows the aim of the parallel decomposition of the register example. The X_i's are copies of a single-reader register and WC specifies the write management component.



Although we have defined the specification exactly as presented at the beginning, we have only used a specification with three readers for the formal transformation. In principle the transformation with the number n of readers as free variable is possible, but this requires other rules. For example the introduction of one local channel has to be generalized to the introduction of a set of channels. This requires more application conditions and more complicated tactics to prove them in concrete situations. Since we do not present the raw (not pretty printed) specifications, we remain in using n as parameter.

The transformation is performed in four steps:
1. The local channels w_i and a_i are introduced with the rule mentioned above. Thereby copies of the trace assertion for the writer are modified, so that internal communications on the new channels can only occur at certain points in communication sequences. As result we get the following new parts in the original specification:

$$\begin{array}{ll} i \in 1..n: & \texttt{trace } \{W, A, w_i, a_i\} \texttt{ in } \mathit{pref}(W.w_i.a_i.A)^* \\ & \texttt{chan } w_1, \ldots, w_n \texttt{ of } V \\ & \texttt{chan } a_1, \ldots, a_n \texttt{ of } \mathit{Signal} \end{array}$$

2. For the development of the n copies X_i of the single-reader registers we have to extend the systems state space. Each X_i needs a variable xv_i for the register content, one variable xm_i indicating the mode, i.e. read and write phases, and one variable xc_i which shows, whether X_i has to return the correct value.

For this purpose the state space extension rule described above is used. In every single step one variable is introduced and the right to write to this new variable is given to appropriate channels. As result we get the following new part for local variables and communication assertions:

$$\begin{array}{l} \texttt{var } xv_1, \ldots, xv_n \texttt{ of } V \\ \texttt{var } xm_1, \ldots, xm_n \texttt{ of } \mathit{Bool} \\ \texttt{var } xc_1, \ldots, xc_n \texttt{ of } \mathit{Bool} \end{array}$$

$$\begin{array}{ll} i \in 1..n: & \texttt{com } w_i \texttt{ write } \{xv_i, xm_i, xc_i\} \\ i \in 1..n: & \texttt{com } a_i \texttt{ write } \{xm_i\} \\ i \in 1..n: & \texttt{com } R_i \texttt{ write } \{c_i, xc_i\} \texttt{ read } \{m\} \texttt{ then } c_i' = \neg m \end{array}$$

There are no restrictions for communications on the new local channels and no effect is specified, i.e. everything is allowed. The external channels R_i also get the right to write the variables xc_i.

Now we strengthen the effects of communications in a way, such that we later can reach exactly the specification of a single-reader register. We additionally define $@w_i = v$ as effect of a communication on w_i, because we want that each X_i receives the value, which the whole system receives from its environment.

$$\begin{array}{ll} i \in 1..n: & \texttt{com } w_i \texttt{ write } \{xv_i, xm_i, xc_i\} \texttt{ read } \{v\} \\ & \qquad \texttt{then } xm_i' \wedge xv_i' = @w_i \wedge \neg xc_i' \wedge @w_i = v \\ i \in 1..n: & \texttt{com } a_i \texttt{ write } \{xm_i\} \texttt{ then } \neg xm_i' \\ i \in 1..n: & \texttt{com } R_i \texttt{ write } \{c_i, xc_i\} \texttt{ read } \{m, xm_i\} \\ & \qquad \texttt{then } c_i' = \neg m \wedge xc_i = \neg xm_i \\ i \in 1..n: & \texttt{com } T_i \texttt{ read } \{c_i, v, xc_i, xv_i\} \\ & \qquad \texttt{then } (c_i' \Rightarrow \ (@T_i = v)) \wedge (xc_i' \Rightarrow \ (@T_i = xv)) \end{array}$$

3. Before the parallel decomposition is possible, the declarations of the local variables m and c_i have to be removed. This requires a lot of single transformation steps. Here we will only give the idea.

It is a correct transformation step to reduce the possible states of the system by the introduction of new Always invariants by the rule given above. For example we introduce for $i = 1, \ldots, n$

$$\texttt{Always } c_i \Rightarrow \ (xc_i \wedge xv_i = v).$$

This invariant relates the old correctness variables c_i to the new local variables xc_i and the register content v to the local copies xv_i. Then we strengthen the effect of communication assertions according to the invariant, here the effects of channels T_i. Initializing the c_i with false makes the invariants redundant, because then they hold initially and are stable for all channels. The invariants are removed. Now the c_i's are removable, since the effect predicates do not use the initial values of the c_i any more.

Important is here only the idea: New invariants relate old and new variables, the invariants are used to strengthen the effect predicates, the invariants are made redundant and are removed, and finally the old variables are no longer used and removed.

4. In the last step we decompose the specification into synchronized simpler ones as shown in the picture. Therefore we use the transformation rule for synchronous decomposition generalized for specifications with more components than presented above. This rule is now applicable. As result we get the following specifications (for $i \in 1..n$:):

$X_i =$

`spec input` w_i `of` V
`output` a_i `of` *Signal*
`input` R_i `of` *Signal*
`output` T_i `of` V

`var` xv_i `of` V
`var` xm_i, xc_i `of` *Bool*

`trace` $\{w_i, a_i\}$ `in` *pref*$(w_i.a_i)*$
`trace` $\{R_i, T_i\}$ `in` *pref*$(R_i.T_i)*$

`com` w_i `write` $\{xv_i, xm_i, xc_i\}$
`then` $xm_i' \wedge xv_i' = @w_i \wedge \neg xc_i'$

`com` a_i `write` $\{xm_i\}$
`then` $\neg xm_i'$

`com` R_i `write` $\{xc_i\}$ `read` $\{xm_i\}$
`then` $xc_i' = \neg xm_i$

`com` T_i `read` $\{xc_i, xv_i\}$
`then` $xc_i' \Rightarrow \ (@T_i = xv_i)$

`end`

$WC =$

`spec input` W `of` V
`output` A `of` *Signal*
`output` $w_1, \ldots, w_n$ `of` V
`input` $a_1, \ldots, a_n$ `of` *Signal*

`var` v `of` V

`trace` $\{A, W, w_i, a_i\}$
`in` *pref*$(w.w_i.a_i.A)*$

`com` W `write` $\{v\}$
`then` $v' = @W$

`com` w_i `read` $\{v\}$
`then` $@w_i = v$

`end`

12 Conclusion

Our work shows that the embedding of our transformational approach in the higher order logic of LAMBDA is possible. It allows formal reasoning about the semantics of reactive systems and the transformation on a syntactic level. Dealing with syntactic systems and their semantics in one single tool provides a high degree of correctness.

It seems to be possible to embed the transformation model in other higher order logic proof-systems like HOL [Gor88] or PVS [OSR93], but the author is not really familiar with them.

Up to now we have studied only small examples. The tactics, which prove the application conditions of transformation rules automatically, are developed parallel to the register example. The specification of the register used here is called *safe register* in the literature. It may be refined by so-called *regular* registers, which are more restrictive on the returned value in the case of overlapping read and write phases. We have formally proved the correctness of this refinement by stepwise transformation. Based on the current experiments it seems to be possible to combine single transformation steps to larger transformation strategies.

We have also implemented an extension of the model presented here for the transformation into OCCAM-like programs. Therefore a syntactic language of so-called *mixed terms* [Old91],[ORSS92] is defined, which contains all specification constructs, implementation parts like input and output communication and assignment from OCCAM, and some additional operators, for example for general recursion and hiding. This language and its semantics are embedded in LAMBDA. Up to now some algebraic laws about program statements are proved. The parallelly decomposed register presented above is transformed into a program by stepwise transformation, but this last step is up to now not performed formally in LAMBDA. Transformation into programs is one of our next goals.

References

[Bjø89] D. Bjørner, A ProCoS project description, ESPRIT BRA 3104, Bulletin of the EATCS 39, 60-73, 1989.

[FFHM92] M.Francis, S.Finn, R.B.Hughes, E.Mayger. LAMBDA Version 4.2, Documentation Set. Abstract Hardware Limited, 1992.

[Gor88] Michael J.C.Gordon. *HOL: A proof generating system for higher order logic.* In Graham Birtwistle & P.A.Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, 73 – 128. Kluwer Academic Publishers.

[JPTW90] V. Jansen, A. Potthoff, W. Thomas, U. Wermuth. *A Short Guide to the AMORE System.* Aachener Informatik-Berichte 90-02, RWTH Aachen, 1990.

[Lam86] L. Lamport. *On interprocess communication Part II.* Distributed Computing, 1:86–101, 1986.

[MF91] M. Fourman, E. Mayger. *Integration of formal methods with system design*, in Proc. VLSI'91, Edingburgh.

[Old91] E.-R.Olderog. *Towards a design calculus for communicating programs.* CONCUR 1991, LNCS 527, 61-77.

[OR93] E.-R. Olderog and S. Rössig. *A Case study in transformational design of concurrent systems.* ProCoS Technical Report, Universität Oldenburg, January 1993, appeared in: M.-C. Gaudel, J.-P. Jouannaud, Eds., Proc. TAPSOFT '93, LNCS 668 (Springer-Verlag, 1993) 90–104.

[ORSS92] E.-R. Olderog,S. Rössig, J.Sander, M. Schenke. *ProCoS at Oldenburg: The Interface between Specification Language and occam-like Programming Language.* Technischer Bericht 3/92, Univ. Oldenburg 1992.

[OSR93] S. Owre, N. Shankar, J.M. Rushby. *User Guide for the PVS Specification and Verification System, Language, and Proof Checker (Beta Release).* Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993. Three volumes.