On Automatic and Interactive Design of Communicating Systems*

Jürgen Bohn   Stephan Rössig

FB Informatik, C.v.O. Universität Oldenburg
Postfach 2503, 26111 Oldenburg, Germany†

Abstract

This paper presents a transformational approach to the design of distributed systems where environment and concurrently running components communicate via synchronous message passing along directed channels. System specifications that combine trace-based with state-based reasoning are gradually modified by application of transfromation rules until occam-like programs are achieved finally. We consider interactive and automatic aspects of such a design process and illustrate our approach by sketching the development of a shared register implementation.

1 Introduction

The design of provable correct software requires formal methods whose usage should be assisted by suitable tools. Following a transformational approach the design needs interactive user help when important design decisions have to be made. Nevertheless simple parts should be automated as far as possible. Ideally the user only guides the design process by indicating the design ideas which are then carried out automatically. Typically sequential implementations are more appropriate for automation while parallelization needs interaction to determine the intended program architecture.

Our approach deals with the transformational development of communicating systems in the mixed term language MIX which encompasses specification and programming notation. A formal refinement notion guarantees that starting from a specification of a desired system only correct implementations can be reached. As part of the ESPRIT Basic Research Action ProCoS a refinement calculus for communicating systems was developed in order to provide a constructive and mathematically sound way for bridging the gap between specifications and programs [Old91, Rös94]. We consider communicating systems as an approach to distributed computing that integrates the state transformation aspect of iterative programs in the sense of UNITY [CM88] and action systems [Bac90] with the CSP paradigm of synchronous message passing along communication channels. When designing such systems several different aspects like concurrency, communication, nondeterminism, deadlock, termination, divergence and assignment to variables have to be considered. A state-trace-readiness semantics in a specification-oriented fashion provides the necessary power to express such properties and concepts. Additionally it induces immediately a refinement relation which is used to define correctness of system transformations.

*This research was partially supported by the CEC with the ESPRIT Basic Research Project No. 7071 ProCoS II and by the German Ministry of Research and Technologie (BMFT) as part of the project KORSO (Korrekte Software) under grant No. 01 IS 203 N.

†{bohn,roessig}@informatik.uni-oldenburg.de

The rest of this paper is structured as follows. Section 2 introduces our specification language SL and explains how SL constructs can be applied in order to specify a regular register with concurrent access. Section 3 considers basic aspects of a transformational approach to system design. Section 4 sketches major steps within the development process of a parallel architecture of sequential components implementing the regular register. Section 5 treats the derivations of sequential implementations by systematic exploitation of specifications. Section 6 deals with the automation of such systematic proceeding in order to decrease the degree of user interaction within the whole design process. A final section concludes this paper with a short discussion of the achieved results.

2 Specification Language SL

The specification language SL develops further the ProCoS specification language SL_0 [JROR90] that was designed to describe continuously running embedded systems communicating with their environment via synchronous message passing along directed channels. A communication along a channel takes place if both, system and environment, are ready for communication on that channel. A system is in a deadlock whenever it does not become ready for communication on at least one channel.

An SL specification provides several parts to describe such communicating systems in a constraint-oriented style. Syntactically a specification is a list of so-called basic items enclosed by `spec` – `end` brackets. The following sketches the basic ideas of these constructs using the general specification pattern given in figure 1. Afterwards a few more details

`spec`

$dir_c\ c\ [\texttt{of}\ ty_c]$ $mode_x\ x\ [\texttt{of}\ ty_x]$	Δ
$ta =\ \texttt{trace}\ \alpha_{ta}\ \texttt{in}\ re_{ta}$	TA
$ca =\ \texttt{com}\ na_{ca}\ \texttt{write}\ \overline{w}_{ca}\ \texttt{read}\ \overline{r}_{ca}$ $\texttt{when}\ wh_{ca}\ \texttt{then}\ th_{ca}$	CA
$\texttt{var}\ v\ \texttt{of}\ ty_v$	lV
$\texttt{chan}\ c\ [\texttt{of}\ ty_c]$	lC
$ini =\ \texttt{initial}\ p_{ini}$	SR^{ini}
$sta =\ \texttt{stable}\ p_{sta}\ \texttt{for}\ \overline{c}_{sta}$	SR^{sta}
$alw =\ \texttt{always}\ p_{alw}$	SR^{alw}
$est =\ \texttt{establish}\ p_{est}\ \text{by}\ \overline{c}_{est}$	SR^{est}

`end`

Figure 1: Specification format.

are discussed in the context of an example specification (cf. figure 3).

The *interface* Δ stresses a static view of the intended system by listing all entities which may be used for interaction with the environment. It consists of optionally typed declarations of external channels with associated direction indication (`input` or `output`) and of global variables with assoicated access mode (`write` or `read`-only).

Essentially the description of the intended dynamic behaviour is split into two parts in SL. The *trace part TA* specifies in which order communications may take place on the various channels. A trace assertion $ta \in TA$ describes a sequencing constraint for the channels of alphabet α_{ta} by giving a regular expression[1] re_{ta} over these channels. Several ordering aspects can be specified in a modular fashion by stating different trace assertions. Technically the so-called *trace language* $\mathcal{L}[\![\Delta, TA]\!]$ of the specification is that regular language over all channels which obeys all sequencing constraints simultaneously. The trace part prevents any communication trace of which the channel order does not belong to $\mathcal{L}[\![\Delta, TA]\!]$.

The *state part* CA relates single communications with the current system state. A communication assertion $ca \in CA$ consists of a channel name na_{ca}, two disjoint lists $\overline{w}_{ca}$ and $\overline{r}_{ca}$ of write and read-only variables, respectively, and two predicates. The when-predicate wh_{ca} over free variables $\overline{w}_{ca}, \overline{r}_{ca}$ disables channel na_{ca} for communication whenever wh_{ca} does not hold in the current state. The value of a communication refered to by $@na_{ca}$ as well as its effect on the system state are specified by the then-predicate th_{ca} over free variables $\overline{w}_{ca}, \overline{r}_{ca}, \overline{w}'_{ca}, @na_{ca}$. In the style of TLA [Lam94] and Z [Spi89] the unprimed variables refer to the values in the before state while the primed ones to those in the after state in which read-only variables $\overline{r}_{ca}$ must not change their values. Giving empty lists as well as predicate `true` is optional. Several communcation assertions for the same channel must be obeyed all together.

The use of a more operational formalization approach to the behaviour specification is supported by declarations of local variables lV and local channels lC. The various state restrictions SR provide a good basis for the integrated reasoning with state-based arguments as invariance and stable properties and with control flow arguments as initial state and establish properties. Technically these latter constraints could be replaced by certain more or less complex combinations of other basic items of which intuitive understanding is then often lost. The same holds for the always possible replacement of the trace part by additional local variables and communications assertions.

In [LG89] a good overview can be found about the various kinds of shared registers treated in the literature on distributed algorithms. According to the classification in [Lam86] we use as running example in this paper a regular register with a single reader and a single writer. In general a register stores values of a type V and the most recently written value shall be returned to the reader if its access does not overlap with a write. In the case of overlapping phases the regular behaviour guarantees that a read phase will return a value that was hold before or after one of write accesses. Figure 2 presents our

Figure 2: Register as communicating system

view of single-writer, single-reader register as communicating system. The writer initiates a writing phase by sending the new value along the input channel W. This phase ends when a corresponding acknowledgment signal is output on channel A. Conversely, the reader initiates a reading phase by sending a signal along the input channel R. This phase ends when a value is returned along the output channel T.

Figure 3 shows a complete SL specification of a regular register which is explained here shortly.[2] Here the interface consisting of the declarations of channels W, A, R, T together

[1] of an extended format additionally using `pref` as prefix closure operator

[2] Similar SL specifications of various registers are presented in [OR93, Rös94] together with a very detailed motivation of the single components.

$$
\begin{array}{rl}
\text{RegisterSpec} = \texttt{spec} & \texttt{input}\ W\ \texttt{of}\ V \\
& \texttt{output}\ A\ \texttt{of signal} \\
& \texttt{input}\ R\ \texttt{of signal} \\
& \texttt{output}\ T\ \texttt{of}\ V \\
ta_1: & \texttt{trace}\ W, A\ \texttt{in pref}(W.A)^* \\
ta_2: & \texttt{trace}\ R, T\ \texttt{in pref}(R.T)^* \\
& \texttt{var}\ new, old\ \texttt{of}\ V \\
& \texttt{var}\ C\ \texttt{of}\ V\texttt{-set} \\
ca_W: & \texttt{com}\ W\ \texttt{write}\ new, C\ \texttt{then}\ new' = @W \wedge C' = C \cup \{@W\} \\
ca_A: & \texttt{com}\ A\ \texttt{write}\ old\ \texttt{read}\ new\ \texttt{then}\ old' = new \\
ca_R: & \texttt{com}\ R\ \texttt{write}\ C\ \texttt{read}\ new, old\ \texttt{then}\ C' = \{new, old\} \\
ca_T: & \texttt{com}\ T\ \texttt{read}\ C\ \texttt{then}\ @T \in C \\
\texttt{end} &
\end{array}
$$

Figure 3: Specification of a regular register.

with the trace part consisting of trace assertions ta_1, ta_2 formalize the value independent aspects. Communications along channels of type `signal` are used for synchronization purposes only but do not pass any message value. The trace assertions guarantee that initiating and ending communications of write as well as read phases always occur in alternating order starting with channels W and R, respectively.

To specify the values that may be returned we use local variables to store certain pieces of information. Variables *old* and *new* shall hold the before and the after value when a write access is active and otherwise that unique value which is stored in the register. Therefore a communication on W updates *new* with the newly received value what is formalized by conjunct $new' = @W$ in the then-predicate of ca_W. Analogously $old' = new$ expresses that *old* gets the value of *new* whenever an A signal ends up a write phase. The idea of the set-valued variable C is to collect all possible return values for a read access. Thus the value $@T$ to be passed by an ending T communication can be easily chosen from C. The initialization of variable C is performed by the initiating R signal of a read phase. Note that the values of *new* and *old* are equal outside of write phases and hence then-predicate $C' = \{new, old\}$ of ca_R leads to the only return value. Any write phase starting during a read phase overlaps this and thus every newly written value becomes a possible return value. Therefore each W communication enriches C by its communication value $@W$.

3 Transformational Implementation Design

To implement communicating systems we use an `occam`-like programming language PL [INM88]. Programs are terms constructed from the 0-ary operators `STOP`, `SKIP`, multiple assignments, input and output on channels, the unary operators `WHILE`, `var` and `chan` for describing loops and declaration of local variables and channels, and the operators `SEQ`, `IF`, `ALT` and `PAR` for sequential, conditional, alternative and parallel composition of lists of n arguments. Figure 4 shows a PL program which implements the register specification of figure 3. Analogously to specifications a program declares its interface to the environment explicitly. The `system` – `end` brackets emphasize that programs represent implementations of communicating systems.

Semantically a communicating system is viewed as pair $\Delta : P$ where the interface Δ declares the communication channels and global variables. The predicate P characterizes the dynamic behaviour of the system as the set of possible observations in a state-trace-readiness model. This model integrates a purely event-based readiness approach [OH86] and a standard input/output semantics into a specification-oriented semantics of which

```
system input W of V
       output A of signal
       input R of signal
       output T of V
       chan u, d of V
       chan r of signal
       PAR[ var new of V
              WHILE true do SEQ[ W?new, u!new, A! ] od,
            var x of V
              WHILE true do ALT[ u?x-->SKIP, r?-->d!x ] od,
            var y of V
              WHILE true do SEQ[ R?, r!, d?y, T!y ] od          ]
   end
```

Figure 4: Register Implementation.

details are presented in [Old91, Rös94]. A major reason for this semantics construction is the immediate presence of a refinement notion for communicating systems. A system $\Delta_1\!:\!P_1$ *refines* a system $\Delta_2\!:\!P_2$ if both ones have the same interface and if behaviour P_1 implies behaviour P_2:

$$\Delta_1\!:\!P_1 \Rrightarrow \Delta_2\!:\!P_2 \quad \text{iff} \quad \Delta_1 = \Delta_2 \text{ and } \models P_1 \;\Rightarrow\; P_2.$$

This definition encompasses a correctness notion *Prog* $\Rrightarrow$ *Spec* since specifications and programs are special representations of communicating systems.

Figure 5 shows a design sequence of a transformational implementation approach. Starting from an SL specification *Spec*, a PL implementation *Prog* is derived in a top-

$$\begin{array}{ccc}
\mathit{Spec} \equiv & S_1 & \\
 & \Lleftarrow\!\!\Uparrow & \\
 & \vdots & \\
 & \Lleftarrow\!\!\Uparrow & \\
 & S_n & \equiv \mathit{Prog}
\end{array}$$

Figure 5: Implementation design sequence.

down fashion by iterated application of transformation rules such that the specification notation is gradually replaced by programming language constructs. The intermediate system expressions S_i are so-called *mixed terms* of the language MIX. This language encompasses specifications and programs as disjoint subsets and extends the application of every programming operator to arbitrary mixed terms. Moreover, there exist additional MIX specific operators in order to express intermediate stages of a system design much more conveniently. E.g. the treatment of the semantically complex PL operator `PAR` can be reduced within MIX to a combination of the simpler operators `SYN` and `HIDE` dealing separately with the aspects of multiple synchronization and of divergence raised by infinite internal communication.

Typically a transition step from mixed term S_i to S_{i+1} is performed by replacing some specification expression S in S_i by a mixed term T where the refinement $T \Rrightarrow S$ is guaranteed by a transformation rule. Then the overall implementation correctness follows from the transitivity of $\Rrightarrow$ and the monotonicity of all operators.

In easy cases a transformation step will replace a specification by a basic PL statement as e.g. an input or output communication or an assignment. Figure 13 below shows appropriate equivalences of specification and programming constructs. But more often more complex specifications have to be decomposed into mixed terms applying some composition operator to several simpler arguments. As typical example supporting this later kind of refinements, figure 6 shows a transformation rule which introduces the synchronization

$$\texttt{spec}\ \Delta\ TA\ CA\ lV\ \texttt{end}$$

$$\Uparrow\!\!\!|||$$

$$\begin{array}{l}\texttt{SYN[}\ \texttt{spec}\ \Delta_1\ TA_1\ CA_1\ lV_1\ \texttt{end}, \ldots, \\ \qquad \texttt{spec}\ \Delta_n\ TA_n\ CA_n\ lV_n\ \texttt{end} \qquad]\end{array}$$

provided $\Delta = \bigcup_{||}{}_{i=1}^{n}\Delta_i,\ TA = \bigcup_{i=1}^{n} TA_i,\ CA = \bigcup_{i=1}^{n} CA_i,\ lV = \biguplus_{i=1}^{n} lV_i$ and ...

Figure 6: Transformation rule SYN decomposition.

operator `SYN`. Generally a side condition "provided ..." restricts the applicability of the transformation rule and describes how the new mixed term is derived by syntactic modifications from the given one. In the example it is expressed that essentially the basic items of the given specification have to be shared out between the new argument specifications $\texttt{spec}\ \Delta_i\ TA_i\ CA_i\ lV_i\ \texttt{end}$ obeying some static semantic constraints.

For practical implementation designs a user needs guidance how to realize intuitive implementation ideas by application of such transformation rules. Here so-called design *strategies* provide recipes how to combine several rules in order to derive implementations in certain situations systematically or even mechanically. Data refinement, parallelization concepts or the development of specific sequential implementations are implementation concepts that can be supported by such strategies. As example we will later consider the automated synthesis of sequential programs based on the syntax directed transformation strategy SDT.

Tool support

An interesting consequence of basing all semantic reasoning on a uniform predicate language is that this reasoning comes close to what can be mechanically supported higher order logic theorem provers. In the German national research project KORSO one of the goals was to provide tool support for formal methods in software design. As part of this work a computer assisted validation of our semantical model was performed within the theorem prover Lambda [BR95]. To this end first the model was implemented in the higher order logic of Lambda [FM91, FFHM93] and various basic propositions about the model have been verified in the Lambda framework interactively. On the one hand this validation gives great confidence in soundness of the model as well as of its formalization in Lambda. On the other hand a basic transformation environment for communicating systems emerges from the verification of transformation rules since Lambda provides mechanisms for the representation of syntactic objects and supports their modification by rule applications. Particularly a transformational design processes is assisted by saving the design history, backtracking mechanisms, generation of proof obligations and a rule browser. Furthermore the tactics concept provides a possibility to perform algorithmic rule applications and automatic condition checking.

4 Parallel Register Architecture

Frequently specifications require that sometimes a system should be ready for communication on several channels. As in occam, the restriction to so-called input guards as arguments of the alternative operator ALT forces parallel implementations in such cases where an output channel must be together ready with at least one other channel.

In the regular register such a situation is present e.g. when a first communication took place. Initially the regular register must be ready for input channels W and R. Independently on the channel along which a communication is performed in the next situation common readiness is required for an input and an output channel. Hence an occam-like implementation of this register has to use concurrently running subcomponents which interact via internal communication. Obviously we shall choose one write manager component *WM* dealing with write access and a read manager *RM* serving the reader. Both these components require access to the value stored in the register. But PL does not provide shared variables and therefore a third component *SV* will play this role. Figure 7 indicates how these components are connected via local channels u, r, d of which usage is

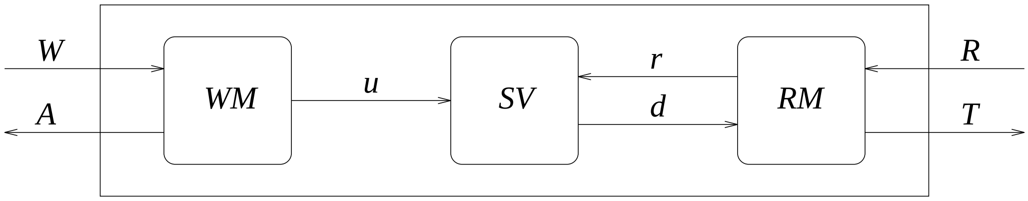

Figure 7: Intended process architecture.

as follows. After having received a new value along W the *WM* component updates the current register value by sending the new value along channel u to *SV* before the external acknowledgment on A is offered. *RM* serves a read request along R by sending an internal request along r to *SV*. The shared variable process immediately answers by delivering its actual value along channel d to *RM* which then transmits this value along T to the reader.[3][4] The specifications *WMspec*, *SVspec* and *RMspec* presented in figure 8 express this intuitive description of *WM*, *SV* and *RM* formally. They are designed by systematic transformation from the original specification shown in figure 3. Due to space limitations we list the major transformation steps towards the parallel decomposition in the following only.[5] Essentially these steps are motivated by the intended architecture which reflects the overall design ideas.

1. The local channels u, r, d are declared and their global communication behaviour is restricted according to the indicated communication order by modification of the trace assertions ta_1, ta_2 to

$$\texttt{trace}\ W, A, u\ \texttt{in}\ \texttt{pref}(W.u.A)^*$$
$$\texttt{trace}\ R, T, r, d\ \texttt{in}\ \texttt{pref}(R.r.d.T)^*$$
$$\texttt{trace}\ u, r, d\ \texttt{in}\ \texttt{pref}(u + r.d)^*\ .$$

2. To store the register value in *SV* and to hold the return value in *RM*, respectively, the state space is extended by variable declaration `var x,y of` V.

[3] The different treatment of write and read accesses to the shared variable process are necessary in order to allow a sequential implementation of *SV* because otherwise the problem of output channels in non singleton readysets would only be delayed.

[4] Note that this register implementation refines the regular specification properly because of a more deterministically chosen return value in the case of overlapping write and read accesses. Essentially this implementation realizes the stronger behaviour of an atomic register.

[5] In [OR93, Rös94] detailed explanations are given on the execution of such steps.

$WMspec$ =	$SVspec$ =	$RMspec$ =

```
WMspec =
spec
  input W of V
  output A of signal
  output u of V
  trace W, A, u
        in pref(W.u.A)*
  var new of V
  com W write new
        then new' = @W
  com A
  com u read new
        then @u = new
end
```

```
SVspec =
spec
  input u of V
  input r of signal
  output d of V
  trace u, r, d
        in pref(u + r.d)*
  var x of V
  com u write x
        then x' = @u
  com d read x
        then @d = x
end
```

```
RMspec =
spec
  input R of signal
  output T of V
  output r of signal
  input d of V
  trace R, T, r, d
        in pref(R.r.d.T)*
  var y of V
  com T read y
        then @T = y
  com d write y
        then y' = @d
end
```

Figure 8: Component specifications.

3. The original variables old and C are removed. To this end they are made auxiliary variables by introduction of appropriate state restrictions and strengthening of communication effects.
4. The local channel declarations are moved in front of the specification and thus they become global ones for the body. Since the trace part prevents infinite communication on local channels u, r, d only, their hiding from the specification does not introduce divergence.
5. The communication assertions of channels u and d are split in order to enable the intended distribution of local variables new, x, y onto the components WM, SV, RM.
6. Now the synchronous decomposition rule shown in figure 6 is applied and we end up with the mixed term

   ```
   chan u, d of V
   chan r of signal
   HIDE u, r, d in SYN[ WMspec, SVspec, RMspec ]
   ```

   with the component specifications of figure 8. Finally the operators HIDE and SYN are replaced by PAR because exactly all channels linking two argument systems of SYN are hidden.

The steps 2. and 3. perform a data refinement on the internal state space thereby proceeding quite systematically. A partial automation of this strategy would be very useful and seems to be possible. Generally executing the above steps and especially those performing the parallel decomposition requires a high degree of user interaction because the underlying rules allow various instantiations of their parameters leading to quite different refinements.

In contrast implementations of the three component specifications $WMspec$, $SVspec$ and $RMspec$ can be achieved by automatic synthesis of sequential programs. The conceptual basis of this automation and its implementation within LAMBDA are dealt with in the rest of this paper.

5 Designing Sequential Implementations

A notion of termination is essential when dealing with sequential implementations. In this section we present a suitable extension of SL to enable the description of termination. This new notion provides the basis for a transformational design of sequential implementations.

In order to refine a specification into a sequential composition of several specifications of reduced complexity, the circumstances have to be expressed, under which the control flow passes from one system to the next one. Therefore so-called *T-specifications* are introduced in SL. These are syntactically distinguished by `system` – `end` brackets instead of `spec` – `end` bracketed so-called S-specifications. Dependent on the trace part T-specifications may terminate in certain situations where the corresponding S-specifications would reach a deadlock. For a detailed comparison of S- and T-specifications see [Rös94]. A consequence of this differentiation is that an empty T-specification `system end` immediately terminates what is equivalent to the `SKIP` statement at the programming level. In contrast the empty S-specification `spec end` denotes an immediate deadlock which is represented in PL by `STOP`.

The following presents two transformation rules which relate S- and T-specifications. The first one in figure 9 allows in particular to switch from an S- to a T-specification which at most differs in the trace part. The trace language $\mathcal{L}[\![\Delta, TA]\!]$ of the S-specification must be equal to $pc\mathcal{L}[\![\Delta, TA_1]\!]$ which denotes the prefix closure of the trace language of the T-specification. When the refined system reaches the STOP it starves in a deadlock.

$$\texttt{spec}\ \ \Delta\ TA\ CA\ \ \texttt{end}$$

$$\Uparrow$$

$$\texttt{SEQ[system}\ \ \Delta\ TA_1\ CA\ \ \texttt{end, STOP]}$$

provided $\mathcal{L}[\![\Delta, TA]\!] = pc\mathcal{L}[\![\Delta, TA_1]\!]$.

Figure 9: Linking S- and T-specifications

Figure 10 shows a more general rule for the sequential decomposition of S-specifications. The first condition of this rule links the trace languages of the different specifications. The other condition "$\mathcal{L}[\![\Delta, TA_1]\!]$ is prefix free" guarantees a unique transition of the control flow from the first to the second argument in the mixed term.

$$\texttt{spec}\ \ \Delta\ TA\ CA\ \ \texttt{end}$$

$$\Uparrow$$

$$\texttt{SEQ[system}\ \Delta\ TA_1\ CA\ \ \texttt{end, spec}\ \ \Delta\ TA_2\ CA\ \ \texttt{end]}$$

provided	$\mathcal{L}[\![\Delta, TA]\!] = pc\mathcal{L}[\![\Delta, TA_1]\!] \cup \mathcal{L}[\![\Delta, TA_1]\!].\mathcal{L}[\![\Delta, TA_2]\!]$
and	$\mathcal{L}[\![\Delta, TA_1]\!]$ is prefix free.

Figure 10: Transformation rule sequential decomposition.

In the following we concentrate on the implemention of T-specifications. The introduction of while-loops within the implementation design process simplifies T-specifications of which trace languages are iterations of prefix-free base languages. The body of an achieved while-loop is built up from the given specification by reducing the trace language to this base language as shown in the conditions of the while rule in figure 11. The termination condition $(\bigvee_{c \in first(\Delta, TA_1)} wh_c)$ is constructed from the when-predicates of those channels which are initially enabled by the trace language.

The decomposition of S-specifications into while-loops can be performed by an preparatory application of the rule in figure 9 and afterwards introducing a while-loop for the T-specification part. In case of a never terminating loop as first argument the sequential composition with `STOP` as second argument can be simplified using the rewriting rule:

$$\texttt{SEQ[WHILE true do}\ P\ \texttt{od}, Q\ \texttt{]}\ \ \rightarrow\ \ \texttt{WHILE true do}\ P\ \texttt{od}\ .$$

$$\texttt{system}\ \Delta\ \mathit{TA}\ \mathit{CA}\ \texttt{end}$$

$$\Updownarrow$$

$$\texttt{WHILE}\ \bigvee_{c \in \mathit{first}(\Delta, \mathit{TA}_1)} wh_c\ \texttt{do system}\ \Delta\ \mathit{TA}_1\ \mathit{CA}\ \texttt{end od}$$

provided $\mathcal{L}[\![\Delta, \mathit{TA}]\!] = \mathcal{L}[\![\Delta, \mathit{TA}_1]\!]^*$ and $\mathcal{L}[\![\Delta, \mathit{TA}_1]\!]$ is prefix free.

Figure 11: Transformation rule loop decomposition.

Another way of decomposing a specification into several ones with simpler trace languages are disjunctive decompositions thereby introducing an ALT or IF operator. Figure 12 shows a transformation rule for alternative decomposition which splits a T-specification into k subspecifications, where k is the number of that interface channels that occur as first element in at least one word of the trace language. Immediate termination is im-

$$\texttt{system}\ \Delta\ \mathit{TA}\ \mathit{CA}\ \texttt{end}$$

$$\Updownarrow$$

$$\begin{aligned}\texttt{ALT[}\ &\texttt{system}\ \Delta\ \mathit{TA}\ \texttt{trace}\ \overline{c}\ \texttt{in}\ d_1.(c_1 + \cdots + c_n)^*\ \ \mathit{CA}\ \texttt{end},\\ &\ldots,\\ &\texttt{system}\ \Delta\ \mathit{TA}\ \texttt{trace}\ \overline{c}\ \texttt{in}\ d_k.(c_1 + \cdots + c_n)^*\ \ \mathit{CA}\ \texttt{end}\ \ ]\end{aligned}$$

provided $\varepsilon \notin \mathcal{L}[\![\Delta, \mathit{TA}]\!]$ and $\mathit{first}(\Delta, \mathit{TA}) = \{d_1, \ldots, d_k\} \neq \emptyset$ and $\overline{c} = \mathit{Chans}(\Delta)$.

Figure 12: Transformation rule alternative decomposition.

possible due to the first rule condition. Each subspecification contains an additional trace assertion that marks one channel to precede each communication trace of that subsystem.

Using these decomposition rules and similiar ones a specification can be systematically refined into a mixed term where the trace languages of all occuring specifications are very simple. Here the languages consists of the empty word or of a single channel name. If furthermore the state part is also of a simple pattern then such specifications can be directly replaced by PL statements. Figure 13 shows that certain T-specifications are

$$c?v\ \equiv\ \begin{array}{l}\texttt{system input}\ c\ \ \texttt{write}\ v\\ \qquad \texttt{trace}\ c\ \texttt{in}\ c\\ \qquad \texttt{com}\ c\ \texttt{write}\ v\ \texttt{then}\ v' = @c\\ \texttt{end}\end{array} \qquad c?\ \equiv\ \begin{array}{l}\texttt{system input}\ c\ \texttt{of signal}\\ \qquad \texttt{trace}\ c\ \texttt{in}\ c\\ \texttt{end}\end{array}$$

$$c!e\ \equiv\ \begin{array}{l}\texttt{system output}\ c\ \ \texttt{read}\ \mathit{free}\ (e)\\ \qquad \texttt{trace}\ c\ \texttt{in}\ c\\ \qquad \texttt{com}\ c\ \texttt{read}\ \mathit{free}(e)\ \texttt{then}\ @c = e\\ \texttt{end}\end{array} \qquad c!\ \equiv\ \begin{array}{l}\texttt{system output}\ c\ \texttt{of signal}\\ \qquad \texttt{trace}\ c\ \texttt{in}\ c\\ \texttt{end}\end{array}$$

Figure 13: Meaning of input and output communication statements in PL.

equivalent to input and output communications in PL. Other simple specifications can be transformed into these patterns and are therefore automatically implementable, as described in the next chapter.

Tool support for application of single rules

A transformational design step based on one rule application can be supported by a tool with the generation of the modified system and the check of the side conditions.

A single application of one transformation rule in a theorem prover like LAMBDA on the one hand modifies the current MIX term and on the other hand generates proof obligations from the rule conditions. To reduce the necessary interaction with the tool the proof programming language of tactics can be used. Tactics are based on possibly guided single rule applications and equational rewriting which are combined by tactical composition constructs like sequences, if-then-else statements and repetitions to proof searching algorithms.

Since most application conditions of our transformation rules are decidable their verification can be automated. For example all conditions concerning regular expressions are decidable. Many other conditions are provable by simple set operations. The tool only needs user guidance when a transformation rule modifies a MIX term in a way that cannot be generated from the context. For example the user should describe the desired subspecifications when applying the parallel decomposition of figure 10.

6 Automatic Program Synthesis

A transformational software design requires even with tool assistance user support to realize creative design decisions. Nevertheless, if the designer has made some decision a tool should perform all necessary transformation steps and check their correct execution. Thus we have started to implement design strategies thereby exploiting the LAMBDA implementations of the transformation rules which arose from a formal validation of our approach [BR95].

There are two ways how to integrate strategies inside LAMBDA. The first one is provided by tactics. Strategies can be realized by sequential combinations of tactics for single transformation rules. This method allows a flexible combination of previously defined tactics. But reasoning about the strategies is impossible in LAMBDA itself because tactics are expressed in a meta language. E.g. termination of tactic applications cannot be proven in LAMBDA.

The second way overcomes this disadvantage. Here strategies are formalized within LAMBDA as functions which implement algorithms that describe the design ideas. This integrated treatment allows us to prove properties of strategies as termination and applicability in certain situations in LAMBDA. While the correctness of tactical strategies follows immediately from the correctness of their underlying rules the correctness of strategy functions has to be proved itself, although these proofs are also reducible to easier rules or simple statements. The correctness of a function *strat* realizing a certain strategy is easily

$$\begin{array}{c} S \\ \Uparrow \\ strat(S) \end{array}$$

provided ...

Figure 14: Strategy as function.

expressed as transformation rule (cf. figure 14) where "..." characterize all side conditions of the strategy. The automated application of such a strategy in LAMBDA is then reduced to a call of a simple tactic which applies the corresponding rule and afterwards expands the definition of *strat*.

A tactical combination of several rules requires the explicit condition check for each rule application. Often in the context of a strategy similar conditions have to be checked for the various rules applications. Such overlapping checks can be avoided in the case of functional strategy implementation. Here all these checks are collected in the single strategy condition thereby removing redundant checks.

SCS: IMPLEMENTING SPECIFICATIONS OF SINGLE COMMUNICATIONS

In a last step of any transformation process simple specifications of communications and their effects to the systems state have to be implemented. Therefore the equivalences of input and output communications in figure 13 are extended to specifications with less restricted communication assertions. Figure 15 shows the implementation of a so-called SCS (*Single Communication System*) for an input channel. The new variable v_c is introduced to

`system input` c `of` ty `write` w `read` r
`trace` c `in` c
`com` c `write` w `read` r `when` wh_c `then` th_c
`end`

$$\Updownarrow$$

`IF[` $wh_c \rightarrow$ `var` v_c `of` ty `SEQ[` $c\,?\,v_c, impl(th_c[v_c/@c])$ `] ]`

provided v_c is a fresh variable

Figure 15: SCS transformation for input channel.

pass the received value from the input to the effect computation. An analogous rule with the sequence `SEQ[` $impl(th_c[v_c'/@c]), c\,!\,v_c$ `]` holds in the case of an output channel. Here the communication value has to be computed before it can be offered to the environment.

The mixed term derived from an SCS rule applications is transformed further by replacing $impl(th_c[v_c/@c])$ and $impl(th_c[v_c'/@c])$, respectively. For a transition predicate p we use $impl(p)$ to denote any program that computes this state transition and afterwards terminates. Not every transition predicate is implementable, e.g. $false$. Thus the design process should yield then-predicates which can be treated by rules of the following kind :

$impl(x' = e)$

$$\Updownarrow$$

$x := e$

$impl(p \wedge q)$

$$\Updownarrow$$

`SEQ[` $impl(p),\ impl(q[Writes(p)/Writes(p)'])$ `]`

provided $Writes(p) \cap Reads(q) = \emptyset$

Figure 16: Implementing transition predicates.

Applying SCS and $impl()$ rules recursively yields a little basic strategy which implements specifications of which the trace part cannot be further decomposed. Automating this SCS strategy as tactic would first apply the SCS rules and then repeatedly $impl()$-rules. A formalization as function in LAMBDA recursively walks through the structure of a mixed term and replaces SCS suitable systems by PL implementations as follows :

```
SCS( SEQ[ P,Q ] )          = SEQ[ SCS( P ),SCS( Q ) ]
SCS( ALT[ P,Q ] )          = ALT[ SCS( P ),SCS( Q ) ]
SCS( WHILE b do P od )     = WHILE b do SCS( P ) od
...
SCS( system ouput c of ty_c ... trace c in c  com c ... end )
                           = IF[ wh_c → var v_c of ty_c SEQ[ impl(th_c[v_c'/@c]), c ! v_c ] ]
```

$$\texttt{SCS(system input } c \texttt{ of } ty_c \ldots \texttt{ trace } c \texttt{ in } c \ \texttt{ com } c \ldots \texttt{ end)}$$
$$= \texttt{IF[} wh_c \rightarrow \texttt{var } v_c \texttt{ of } ty_c \texttt{ SEQ[} c?v_c, impl(th_c[v_c/@c]) \texttt{]]}$$

All other mixed terms remain unchanged by SCS. The corresponding strategy rule is presented in figure 17.

$$S$$
$$\Uparrow$$
$$SCS(S)$$

provided no local variable v_c occurs free in th_c

Figure 17: SCS strategy rule

In the following SDT strategy we will use this SCS implementation as basic strategy.

SDT: GENERATING SEQUENTIAL IMPLEMENTATIONS

For restricted classes of specifications it is possible to generate a program structure from the trace part automatically. The idea of the *Syntax Directed Transformation* strategy (SDT) is to drive the transformation process by the structure of the regular expression of the only trace assertion of a specification. A tactical automation of this strategy would recursively apply the decomposition rules presented in chapter 5. This tactic would perform many similar checks of application conditions which are avoided by the following functional implementation.

The function PCS formalizes in LAMBDA the inductive construction of a *Program Control Structure* from the operators of one regular expression and calls the SCS strategy to generate communication statements for channel names in the regular expression.

$$\begin{aligned}
\texttt{PCS(} \Delta, re1 + re2, CA \texttt{)} &= \texttt{ALT[SEQ[PCS(} \Delta, re1, CA \texttt{),PCS(} \Delta, re2, CA \texttt{)]]} \\
\texttt{PCS(} \Delta, re1.re2, CA \texttt{)} &= \texttt{SEQ[PCS(} \Delta, re1, CA \texttt{),PCS(} \Delta, re2, CA \texttt{)]} \\
\texttt{PCS(} \Delta, re*, CA \texttt{)} &= \texttt{WHILE} \ldots \texttt{do} \ldots \texttt{od} \\
\texttt{PCS(} \Delta, c, CA \texttt{)} &= \texttt{SCS(system } \Delta|_c, \texttt{trace } c \texttt{ in } c \texttt{ , } CA|_c \texttt{ end)}
\end{aligned}$$

The interface Δ and communication assertions CA are used for calls of the SCS strategy where $\Delta|_c$ denotes the restriction of Δ and $CA|_c$ gives the communication assertion of channel c. Figure 18 shows the corresponding PCS rule which generates sequential programs for certain T-specifications.

$$\texttt{system } \Delta \texttt{ trace } \overline{c} \texttt{ in } re \quad CA \texttt{ end}$$
$$\Uparrow$$
$$\texttt{system } \Delta \texttt{ PCS(} \Delta, re, CA \texttt{) end}$$

provided re is SDT suitable and $impl(th_c)$ is defined for all $c \in \overline{c} = Chans(\Delta)$.

Figure 18: PCS implementation of system specifications.

Basically PCS uses the rules presented in chapter 5 and the SCS function. The conditions of the PCS rule guarantee that all application conditions corresponding to the intermediate transformation steps are satisfied. SDT suitable regular expressions contain no nested iterations (stars). Further more alternative regular expressions are restricted to input channels as first letters.

Now the *SDT strategy* is defined as follows: An S-specification is transformed by the rule in figure 9 into a T-specification with a following `STOP`. Then PCS and SCS are applied to this T-specification. Based on algebraic laws, the so far generated program is finally simplified by rewriting rules like those in figure 19.

`SEQ[ WHILE do` $true$ `od`P, Q `]`	$\rightarrow$	`WHILE` $true$ `do` P `od`
`ALT[ ALT[` ... `] ]`	$\rightarrow$	`ALT[` ... `]`
`IF[` $true, P$ `]`	$\rightarrow$	P
`SEQ[` $c?v, x:=v$ `]`	$\rightarrow$	`SEQ[` $c?x, v:=x$ `]`
`SEQ[` $v:=e, c!v$ `]`	$\rightarrow$	`SEQ[` $v:=e, c?e$ `]`
`var` v `of` ty P	$\rightarrow$	P, if v is an auxiliary variable in P

Figure 19: Rewriting Rules for the SDT strategy

The SDT strategy can be applied to each of the component specifications *WMspec*, *RMspec* and *SVspec* (see figure 8) of the register example. The combined application of PCS, SCS, *impl*() and simplifying rewriting rules yield the implementations which are shown as the three arguments of the PAR operator in figure 4.

The three specifications *WMspec*, *SVspec* and *RMspec*satisfy the application conditions of the SDT strategy. Its application yields the following implementations of *WM*, *SV* and *RM*:

$$\begin{array}{lcl} WM & = & \texttt{var}\ new\ \texttt{of}\ V \\ & & \quad \texttt{WHILE true do SEQ[}\ W?new, w!new, A!\ \texttt{] od} \\ SV & = & \texttt{var}\ x\ \texttt{of}\ V \\ & & \quad \texttt{WHILE true do ALT[}\ w?x \rightarrow \texttt{SKIP}, r? \rightarrow t!x\ \texttt{] od} \\ RM & = & \texttt{var}\ y\ \texttt{of}\ V \\ & & \quad \texttt{WHILE true do SEQ[}\ R?, r!, t?y, T!y\ \texttt{] od} \end{array}$$

Figure 20: Implementations of *WMspec*, *RMspec* and *SVspec*.

7 Discussion

We reported on a mixed term language MIX for the transformational design of communicating systems. Using the example of a register specification we demonstrated how to realize certain implementation ideas in a transformational design approach.

In the theorem prover LAMBDA the mixed terms and transformation rules have been formalized in order to validate the whole approach and prove the rules mechanically. At a first stage this embedding provides a simple tool for interactive execution of transformation steps.

In a transformational setting strategies systematically combine several rules in order to direct large transformation steps. To decrease the degree of user interaction in a design process the execution of such strategies has been automated in LAMBDA. Aspects of different realizations are discussed on the examples SCS and PCS. These strategies are used to generate implementations for the sequential components of the previously parallel decomposed register specification. A formal treatment of strategies inside LAMBDA allows to prove properties like correctness, termination and applicability to certain mixed terms.

Ideas for further strategies reveals in the context of parallel implementations concerning the systematic treatment of shared variables and methods of data refinement. Building up these strategies together with their integration in a design tool yields improved support of important design tasks.

References

[Bac90] R.J.R. Back. Refinement calculus, Part II: Parallel and Reactive Programs. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems - Models, Formalisms, Correctness*, LNCS 430, pages 67–93. Springer-Verlag, 1990.

[BR95] J. Bohn and S. Rössig. Towards a design assistant for communicating systems. Technical Report ProCoS Doc. Id. OLD JB 2/1, Univ. Oldenburg – FB10, 1995.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design - A Foundation.* Addison-Wesley, 1988.

[FFHM93] M. Francis, S. Finn, R.B. Hughes, and E. Mayger. *LAMBDA Version 4.3, Documentation Set.* Abstract Hardware Limited, London, 1993.

[FM91] M. Fourman and E. Mayger. Integration of formal methods with system design. In *Proc. VLSI'91, Edingburgh*, 1991.

[INM88] INMOS Ltd. *occam 2 Reference Manual.* Prentice Hall, 1988.

[JROR90] K.M. Jensen, H. Rischel, E.-R. Olderog, and S. Rössig. Syntax and informal semantics of the ProCoS specification language level 0. Technical Report ESPRIT Basic Research Action ProCoS, Doc. Id. ID/DTH KMJ 4/2, Technical University of Denmark, Lyngby, Dept. Comput. Sci., 1990.

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

[Lam94] L. Lamport. The temporal logic of actions. *TOPLAS*, 16(3):872–923, 1994.

[LG89] N.A. Lynch and K.J. Goldman. Distributed algorithms. Technical Report MIT/LCS/RSS 5 6.852 Fall 1988, MIT, 1989.

[OH86] E.-R. Olderog and C.A.R. Hoare. Specification-oriented semantics for communicating processes. *Acta Inform.*, 23:9–66, 1986.

[Old91] E.-R. Olderog. Towards a Design Calculus for Communicating Programs. In J.C.M. Baeten and J.F. Groote, editors, *Proc. CONCUR '91*, LNCS 527, pages 61–77. Springer-Verlag, 1991. invited paper.

[OR93] E.-R. Olderog and S. Rössig. A case study in transformational design of concurrent systems. In M.-C. Gaudel and J.-P. Jouannaud, editors, *TAPSOFT'93: Theory and Practice of Software Development*, LNCS 668, pages 90–104. Springer-Verlag, 1993.

[Rös94] S. Rössig. *A Transformational Approach to the Design of Communicating Systems.* PhD thesis, Univ. Oldenburg, 1994. Tech. report 4-94, Fachbereich Informatik, Univ. Oldenburg.

[Spi89] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, London, 1989.