Composition and Refinement Mapping based Construction of Distributed Applications

Arnulf Mester, Heiko Krumm
Universität Dortmund, Fachbereich Informatik, D-44221 Dortmund, Germany
Phone +49 231 755-4662, Fax -4730, {mester|krumm}@ls4.informatik.uni-dortmund.de

Abstract

Major steps of the design of distributed applications correspond to the integration of predefined patterns. To support such design steps, a concept for refinement by pattern composition is introduced which applies formal process composition and provides functions for the tool assisted construction and modification of specifications. The approach is based on L. Lamport's Temporal Logic of Actions TLA and the related theory of refinement mappings and system composition.

1 Introduction

The increasing complexity of distributed applications can be faced by enhanced design support, by implementation-near tools and building blocks as well as by more abstract design guides, building blocks and methods. On an implementation near level, syntax and static semantics based tools - as they are integrated into current development environments (e.g., [18]) - support the construction of design documents and implementation modules. Libraries provide for reusable solutions of specific application fields and implementation platforms (e.g., [7, 15, 19]). On a more abstract informal level, the design process can be guided by design patterns [9]. Formal description techniques (e.g., Lotos [12], TLA [14]) model the dynamic semantics and support the specification and verification of system behaviour. Correctness preserving transformation methods (e.g., [3, 4, 5, 8, 13]) view the design as a series of stepwise refinements which can be guided by transformation rules.

Like correctness preserving transformation methods, our approach supports the stepwise refinement and verification of formal description technique based specifications. A new type of transformation 'pattern integration' provides for the efficient reuse of specification modules. It corresponds to syntactical transformations which can be performed by a tool. So, even very specific design patterns can be utilized efficiently (However, unlike CPTs, some transformations have to be accompanied by explicit verification steps).

The approach concentrates on the design of control structures of distributed applications. They can be viewed as a composition of patterns which provide abstract distributed algorithms for the distribution of functionality, user models of standardized communication services, and detailed interface mechanisms to application program interfaces of application platforms.

The approach is based on L. Lamport's Temporal Logic of Actions (TLA [14]) and refers to the concepts of refinement mappings [2] and composition by logical conjunction [1]. We apply a compositional TLA style which supports the specification of process systems and corresponding decompositional verifications [10].

The paper introduces pattern integration and an additional operation for process splitting in principle and with respect to tool functions. (A similar process splitting operation is proposed in [6] for algebraic process definitions.) The concepts are exemplified by views of an example (presented in more detail in [17]). The example is 'academic'. However, it is designed to sketch practical aspects of the application of pattern integration in the complete design process of computer network applications. So, the paper concentrates on the application, the formal background is outlined only.

2 TLA and Refinement Mappings

TLA is a linear time temporal logic modelling systems as state transition systems [14]. The state space is defined by a set of variables. The next state relation is structured into so-called actions. A closed system can be described by a TLA formula in canonical form.

```
variables id, od, s ;
 Init     ≜ s=idle ;
 Start(p) ≜ s=idle ∧ id'=p  ∧ s'=pre ;
 Step     ≜ s=pre  ∧ id'=id ∧ od'=f(id) ∧ s'=post ;
 F        ≜    Init                                       ! initial states
            ∧ □ (  Step ∨ ∃ p: Start(p)                   ! next state: actions
                 ∨ (id'=id ∧ od'=od ∧ s'=s) )             ! stuttering steps
            ∧ ∀ p: WF_id,od,s(Start(p)) ∧ WF_id,od,s(Step);  ! liveness
```

The canonical formula F above describes a system with three variables *id*, *od*, and *s*. Each system execution starts with a state fulfilling the initial predicate *Init*. A pair of successive states $<\sigma, \sigma'>$ always fulfills one of the actions *Start* or *Step* or is a stuttering step (i.e., $\sigma = \sigma'$). In the definitions of the actions, the primed variables refer to the successor state. The fairness assumptions in the last line of F express the liveness of the system: both actions are assumed to be scheduled weakly fairly. An action is called weak fair in a behaviour, if it is infinitely often executed unless infinitely often disabled.

Besides of the weak fairness operator WF, TLA provides for a strong fairness operator SF (not used in this paper). The always operator is denoted by $\Box$. $\mathsf{Enabled}(\alpha)$ is true in a state σ, if there can exist a value combination σ', such that $<\sigma, \sigma'>$ fulfills the action α.

Verification is performed by proofs of implications. Assume the formulas F and G to describe two systems, F intended to be a refinement / implementation of G. The two systems shall correspond to each other with respect to so called visible variables which occur in both formulas. Additionally F and G may contain internal variables x_F and x_G. F is a correct refinement of G iff the formula $\exists x_F : F \Rightarrow \exists x_G : G$ can be proved. Proofs are supported by the TLA inference rules. Refinement proofs can profit from refinement mappings [2]. The existence of a refinement mapping f_R from F to G is sufficient for the proof. Later we utilize an appropriate specification construction, where a refinement mapping gets obvious. A refinement mapping f_R is a function from the state components of the fine-system to the state components of the coarser system. f_R has to map states of F to states of G mapping visible variables by identity. Initial states of F have to be mapped to initial states of G. The images of state pairs fulfilling the next state relation of F have to fulfill the next state relation of G. Finally each image of a state sequence of F has to fulfil the fairness assumptions of G.

3 Processes in a compositional TLA style

A compositional TLA style – supporting both safety and liveness properties – is introduced in [10] and applied here in an enhanced version. It facilitates the description of process systems. The style provides for process abstractions[1]. As in Lotos [12], a process in principle is an open subsystem which can perform joint actions with its environment, and data parameters of actions support the communication of values. Moreover, a process specification can be interpreted for itself. It reflects a closed system consisting of the process and an environment which is universal in the sense that it does not constrain the process.

```
variables   e_Start, e_Step,                          ! environment readiness variables
            id, od, s ;                               ! private state variables
 Init     ≜ s=idle ;
 Start(p) ≜ s=idle ∧ id'=p  ∧ s'=pre ;
 Step     ≜ s=pre  ∧ id'=id ∧ od'=f(id) ∧ s'=post ;
 cStart(p)≜ Start(p) ∧ p ∈ e_Start ;                  ! conditional Start action
 cStep    ≜ Step ∧ e_Step ≠ ∅;                        ! conditional Step action
 Appl1    ≜    Init                                   ! initial states
         ∧ □ (  Step ∨ ∃ p: Start(p)                  ! next state: actions
              ∨ (id'=id ∧ od'=od ∧ s'=s) )            ! stuttering steps
         ∧ ∀ p: WF_id,od,s(cStart(p))
         ∧ WF_id,od,s(cStep) ;                        ! liveness
```

[1] TLA and it's accompanying notations are intended as broad-spectrum specification tools. The compositional TLA style aroused from the experiences of our 'Tools for TLA based specifications' project [16], where the need of developers for more guidance in specification has been identified.

The canonical TLA formula *Appl1* above describes a process. The formula $Appl1 \wedge \Box(e_{Start} = V_p \wedge e_{Step} \neq \emptyset)$ describes a corresponding separated process and is equivalent to the formula F of Sect. 2. V_p denotes the set of values of the data type of the parameter p of the action *Start*.

The liveness properties are described by fairness assumptions on conditional actions which are conjunctions of actions and environment conditions.

The environment readiness variables (introduced for each fair process action) are assumed to be set by the environment of the process. By writing these variables, the environment can define, which possible system action execution it tolerates – either combined as joint action or with a stuttering step of the environment. For our example above, e.g. if $p \in e_{Start}$, the environment can tolerate the action *Start(p)* in the next step, i.e. either stutters or makes a joint action with it.

By style convention, the actions only affect private variables. Additional conventions concern those actions which are supplied with a fairness assumption. Fair actions of a process have to be disjoint to all other actions of the process. Also, they have to be disjoint to stuttering steps.

```
process Appl1 ;
  variables id, od : array [0..5] of integer ; s : (idle, pre, post) ;
  Init ≜ s=idle ;
  actions
    Start(p : array [0..5] of integer) ≜ s=idle ∧ id'=p ∧ s'=pre ;
    Step ≜ s=pre ∧ id'=id
      ∧ od'=[2*id[0],2*id[1],..,2*id[5] ] ∧ s'=post ;
  WF : Start, Step ;
end ;
```

Fig. 1 *Process* Appl1

For the description of processes, we use a PASCAL like notation in the sequel. The notation only specifies the non-canonical parts. So, Fig. 1 represents the formula *Appl1* from above. Because it represents tool input syntax, the variables are now typed and the unknown symbol *f* is replaced by scalar multiplication. We will refer to this form as flat system formula or flat process definition synonymously. 'Flat' characterizes a definition with all specification operations (cf. Sect. 5) performed. The formal semantics of this notation is given by the correspondence to TLA formulas outlined above.

4 Process patterns

Any productive use of formal design must rely on building blocks. Our building blocks for specification are processes and process patterns. They are able to define various abstractions in mixed bottum-up, top-down and 'middle-out' design.

The process *Appl1* of Fig. 1 corresponds to a more general pattern. It is a sequential and terminating process which is defined over three control states and computes result data by the application of a function to input data.

```
process Prepost (tdata : datatype ; f : function) ;
  variables d : tdata ; s : (idle, pre, post) ;
  Init ≜ s=idle ;
  actions
    Start(p : tdata) ≜ s=idle ∧ d'=p ∧ s'=pre ;
    Step ≜ s=pre ∧ d'=f(d) ∧ s'=post ;
  WF : Start, Step ;
end ;
process Auxvar (tdata : datatype) ;
  variables v : tdata ;
  Init ≜ true ;
  actions  Write(p : tdata) ≜ v'=p ;
end ;
```

Fig. 2 *Process patterns* Prepost *and* Auxvar

The pattern is called *Prepost* and shown in Fig. 2. *tdata* and *f* denote generic parameters of the pattern. The second pattern *Auxvar* can be used to introduce auxiliary variables, i.e. variables used for verification purposes only and not to appear in implementations.

It's obvious, that a library of process patterns modelling common abstractions[2] in distributed applications design significantly increases the productivity of the designer. Appropriate concepts and tool

[2] More representative process patterns appear in Sect. 6.

support for using process patterns and processes as building blocks in design are described in the next section.

5 Operations on specifications

During design, design decisions are normally introduced stepwise into the artifact. In formal design, the formal specification of each maturity 'level' captures the sum of design decisions done and introduced already. We are capturing design decisions by recording the sequence of specification operations and their process/pattern parameters. Specification operations are formally relating their resulting process definitions with their input process definitions.

Features, formal background and motivation for the operations are presented. The syntax for the operations is exemplified in the next section.

5.1 Process composition

The compose operation takes as input a number of processes and a coupling description and delivers a flat process definition for the resulting system.

Formally, a system S composed of processes $P_1, P_2, .., P_m$ is described by a compositional system formula $S \triangleq P_1 \wedge P_2 \wedge .. \wedge P_m \wedge CC$. The different P_j stand for the process formulas. CC is an invariant and describes the coupling of the system. CC defines the system actions which are conjunctions of process actions and process stuttering steps. To each system action, each process contributes by exactly one action respectively stuttering step.

With respect to fair process actions, CC defines the environment readiness variables of each process to be functions of the 'real' state variables of the other processes. Let α be a fair action of a process P_i which occurs in a system action β together with the actions γ_j of other processes. The readiness variable e_α of process P_i has always to be equal to the intersection of the system readiness variable e_β and the sets $\{p : \mathsf{Enabled}(\gamma_j(p))\}$.

The occurence of fair process actions in system actions is constrained by additional style conventions. Each fair action of a process has to occur exactly once in the set of system actions. If the fair process action $\alpha(p_1, .., p_n)$ occurs in a system action $\beta(q_1, .., q_m)$, the actual parameters $p_1, .., p_n$ of α have to be identical to the formal parameters $q_1, .., q_m$ of β. If α is supplied with the strongest fairness assumption of the process actions of β, the fairness assumption of β is analogous to the fairness assumption of α.

Due to the conventions, the compositional system formula can be transformed syntactically into an equivalent flat formula (as defined in Sect. 3) which meets the syntax of a process description.

5.2 Pattern integration

Input to the pattern integration operation are processes P_i which are instances of patterns. We intend R to be a combination of the P_i for which implications $R \Rightarrow P_i$ hold (dependent on the role of P_i, for some P_i, $R \Rightarrow \exists e_\alpha, \ldots : P_i$ may be sufficient). The benefit of this is very important: The resulting process is formally related by refinement mappings with all its integrated processes, i.e. their behaviour and theorems.

The style conventions of composition (cf. last Subsect.) are weakened. There may be fair process actions which occur in more than one system action. The actual parametrization of fair process actions not necessarily has to be identical to the formal parameters of system actions.

We split the task of pattern integration into a syntactical operation and accompanying proof. The syntactical operation is prepared mechanically and followed by creative modification. The preparing operation results in an intermediate system IR which is constructed by firstly computing the flat form of the composition and then applying variable substitutions which replace component variables by system variables. The substitutions can serve as proposals of refinement mappings for proofs of $R \Rightarrow P_i$.

5.3 Composed patterns

Composition and pattern integration can support the application design. Additionally it can facilitate the definition of patterns if a pattern regards different aspects in combination. So, a pattern may be a composition of resource oriented processes which reflect a predefined distribution of functionality. By means of a composition of constraint oriented processes, a pattern can reflect a logical decomposition of system properties.

5.4 Process splitting

Design by pattern integration would result in one large flat specification of a distributed application. Nevertheless, distributed applications consist of several components which are located at different sites. Therefore, we propose the operation of process splitting. Like pattern integration, it is based on composition and performed interactively.

Process splitting starts from a flat specification F and transforms it into a compositional specification C which is composed from a set of processes P_i and which is a refinement of F. We assume that the actions of F (as well as the *Init* predicate) are defined as conjunctions of predicates.

The process splitting is performed in a sequence of steps. First, the designer proposes a set of process names $P_1, \ldots, P_n$. Then he has to define the distribution of variables. For each variable of F he can define a non-empty set of those processes, in which a copy of the variable will occur. Thereafter a tool can check, if the process splitting can be prepared mechanically. It checks the occurrence of variables in predicates, and decides if all variables of a predicate are assigned to the same set of processes. If this is the case, the predicate is associated with this set of processes. If all predicates can be associated with process sets, a proposal IC for C can be computed mechanically. Finally IC can be modified interactively to form C.

IC is a composition of processes $P_1, \ldots, P_n$. IC has the notation of a composition which is equivalent to F. Additionally, the specifications of the processes P_i are parts of IC. The variables of P_i are copies of all the variables of F which are assigned to P_i. The *Init* condition and the actions are conjunctions of copies of predicates. For each action β of F, a process P_i will contain a corresponding action α. α is defined by the conjunction of these predicates of β the process set of which contains P_i (if β does not contain such predicates, α equals to $true$).

An arbitrary splitting – not only motivated by physical variable distribution – is possible, e.g. for preparing simplification and verification tool employment (cf. Sect. 7).

6 A distributed ISO/OSI-conformant producer/consumer example

We sketch parts of the stepwise design of a distributed application, modelling a producer and a consumer agent targeted at a realistic use of an ISO/OSI-conformant transfer service and appropriate procedure style application program interface. The full example is available as [17]. The first three subsections exemplify the specification operations described above, using the syntax of our manipulation tool. The last two subsections illustrate the benefits of the manipulation tool when using larger process patterns and specifications as building blocks for specifications.

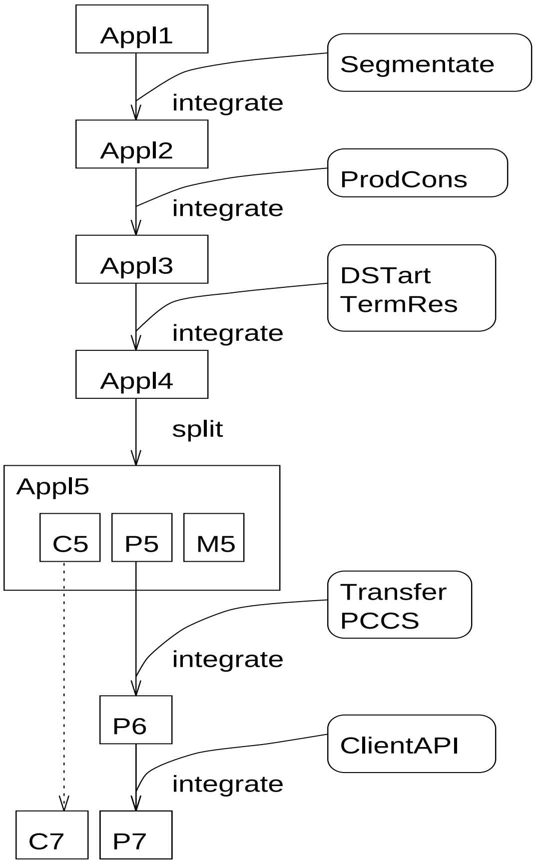

Fig. 3 *Overview of the example design*

Fig. 3 gives an outline of the example design. An abstract process *Appl1* performs a terminating two-step operation. The final result is a system of two remote processes *P7* and *C7*. *P7* produces segments of data, *C7* consumes and processes segments. *P7* and *C7* communicate via a 'standard' communication service. The application programming interface to the service is provided by a special application platform. In order to refine *Appl1* into this producer consumer system, the design references the predefined patterns *Segmentate*, *ProdCons*, *DStartTermRes*, *TransferPCCS*, and *ClientAPI*.

6.1 Process composition

```
type  aoi = array [0..5] of integer ;
process CAppl1 ;
  processes  P : Prepost(aoi, [x0,..,x5]→[2*x0,..,2*x5]);
             A : Auxvar(aoi) ;
  actions
    Start(p : aoi) ≜ P.Start(p) ∧ A.Write(p) ;
    Step ≜ P.Step ∧ A.Stutter ;
end ;
process FAppl1 ;
  variables P.d : aoi ; P.s : (idle, pre, post) ; A.v : aoi ;
  Init ≜ P.s=idle ∧ true ;
  actions
    Start(p : aoi) ≜ P.s=idle ∧ P.d′=p ∧ P.s′=pre ∧ A.v′=p ;
    Step ≜    P.s=pre ∧ P.d′=[2*P.d[0],..,2*P.d[5] ]
            ∧ P.s′=post ∧ A.v′=A.v ;
  WF : Start, Step ;
end ;
```

Fig. 4 *System composed from* Prepost *and* Auxvar *instances*

Fig. 4 shows the processes *CAppl1* and *FAppl1*. *CAppl1* is composed from instances P and A of the patterns *Prepost* and *Auxvar* (cf. Fig. 2). Furthermore it is an example for the syntax of a compositional system. The coupling constraint CC follows from the actions part of *CAppl1*. It defines the system actions to be joint actions of *P.Start* and *A.Write*, or to be *P.Step* actions accompanied by stuttering steps of A. Implicitly, it defines the environment readiness variables of the component P to be functions: $\Box(P.e_{Start} = \{p \in aoi : \mathsf{Enabled}(A.Write(p))\} \cap e_{Start} \wedge P.e_{Step} = e_{Step})$. *CAppl1* is equivalent to *FAppl1* which represents the flat system formula (which is – under appropriate variable renaming – equivalent to *Appl1* of Fig. 1).

6.2 Pattern integration

```
process IAppl1 ;
  variables id, od : aoi ; s : (idle, pre, post) ;
  integrate
       A : Auxvar(aoi) substitute A.v by id ;
       P : Prepost(aoi,[x0,..,x5]→[2*x0,..,2*x5])
           substitute P.s by s ; P.d by if s=post then od else id ;
  actions
    Start(p : aoi) ≜ A.Write(p) ∧ P.Start(p) ;
    Step ≜ A.Stutter ∧ P.Step ;
 end ;

process IAppl1 ;
  variables id, od : aoi ; s : (idle, pre, post) ;
  Init ≜ true ∧ s=idle ;
  actions
    Start(p : aoi) ≜
     id′=p ∧ s=idle ∧ (if s=post then od else id)′=p ∧ s′=pre ;
    Step ≜
     id′=id ∧ s=pre ∧ s′=post
      ∧ (if s=post then od else id)′=
          [2*(if s=post then od else id)[0],..,
           2*(if s=post then od else id)[5] ] ;
  WF : Start, Step ;
 end ;
```

Fig. 5 *Construction of* Appl1 *by pattern integration*

For exemplification, we alternatively construct *Appl1* of Fig. 1 by pattern integration. Fig. 5 shows definition and result of the mechanical operation. The variable *id* of *IAppl1* is shared between an *Auxvar* and a *Prepost* process. By simplifications *IAppl1* can be modified interactively and *Appl1* of Fig. 1 can be derived.

6.3 Design by pattern integration

Pattern integration steps can substantially facilitate the construction of the detailed specification of a distributed application. We assume that a suitable library of patterns exists. The designer starts from an abstract specification of the application and successively develops more detailed specifications by the integration of pattern instances.

```
 process Segmentate (n : cardinal ; t1, t2 : datatype ; fs : function) ;
  variables  id : array[0..n-1] of t1 ; od : array[0..n-1] of t2 ;
             sc : 0..n ;  s : (idle, ready, term) ; stc : cardinal ;
  Init ≜ s=idle ∧ stc>0 ;
  actions
    Start(p : array[0..n-1] of t1) ≜
      s=idle ∧ id′=p ∧ sc′=0 ∧ s′=ready ∧ stc′=stc ;
    Reset ≜
      s=ready ∧ stc>0 ∧ id′=id ∧ sc′=0 ∧ s′=ready ∧ stc′<stc;
    DoSeg ≜
      s=ready ∧ sc<n ∧ id′=id ∧ od′=od besides od[sc]=fs(id[sc])
      ∧ sc′=sc+1 ∧ s′=ready ∧ stc′=stc ;
    Terminate ≜ s=ready ∧ sc=n ∧ id′=id ∧ od′=od
      ∧ sc′=sc ∧ s′=term ∧ stc′=stc ;
  WF : Start, DoSeg, Terminate ;
 end ;
```

Fig. 6 *Pattern* Segmentate

Now, we outline the refinement of *Appl1* of Fig. 1 into the distributed application, which consists of two application processes, a producer of segments and a remote consumer. The first design step is the integration of the pattern *Segmentate* (Fig. 6) in order to introduce the segmentwise processing of data. *Segmentate* successively applies the function *fs* to the elements of array *id*, the results stored in *od*. To prepare reactions on exceptions of the distributed system, the process may be reset and may start the processing again. Since the number of resets is limited (by *stc*), the process will eventually terminate due to the fairness assumptions.

```
process IAppl2 ;
  variables  id, od, aod : array[0..5] of integer ; sc : 0..6 ;
             s : (idle, ready, term) ; stc : cardinal ;
  integrate
    A : Appl1 substitute
         A.id by id, A.od by if s=term then od else aod,
```

```
        A.s by  if s=idle then idle elsif s=ready then pre else post;
    S : Segmentate(6,aoi,aoi,x→2*x) substitute
        S.id by id, S.od by od, S.sc by sc, S.s by s, S.stc by stc;
    actions
        Start(p : aoi) ≜ A.Start(p) ∧  S.Start(p) ;
        Reset ≜          A.Stutter  ∧  S.Reset ;
        DoSeg ≜          A.Stutter  ∧  S.DoSeg ;
        Terminate ≜      A.Step     ∧  S.Terminate ;
end ;
```

Fig. 7 *Integration of* Segmentate *into* Appl1

The integration of *Segmentate* into *Appl1* is shown in Fig. 7. We want *IAppl2* to be a refinement of a composition of an instance of *Segmentate* with an instance of *Appl1*. Note, that the third array variable *aod* is an auxiliary variable as it is only written. It does not influence the execution of the system and may be omitted in an implementation.

```
process IAppl2 ;
  variables
    id, od, aod : array[0.. 5] of integer ;
    sc : 0.. 6 ; s : (idle, ready, term) ; stc : cardinal ;
  Init ≜
    (if s=idle then idle elsif s=ready then pre else post)=idle
     ∧  s=idle ∧  stc>0 ;
  actions
    Start (p : aoi) ≜
         (if s=idle then idle elsif s=ready then pre else post)=idle
      ∧  id′=p
      ∧  (if s=idle then idle elsif s=ready then pre else post)′=pre
      ∧  s=idle ∧  sc′=0 ∧  s′=ready ∧  stc′=stc ;
    Reset ≜
         id′=id
      ∧  (if s=term then od else aod)′=(if s=term then od else aod)
      ∧  (if s=idle then idle elsif s=ready then pre else post)′ =
          (if s=idle then idle elsif s=ready then pre else post)
      ∧  s=ready ∧  stc>0 ∧  sc′=0 ∧  s′=ready ∧  stc′<stc ;
    DoSeg ≜
         id′=id
      ∧  (if s=term then od else aod)′=(if s=term then od else aod)
      ∧  (if s=idle then idle elsif s=ready then pre else post)′ =
           (if s=idle then idle elsif s=ready then pre else post)
      ∧  s=ready ∧  sc<6
      ∧  od′=od besides od[sc]=(x→2*x)(id[sc])
      ∧  sc′=sc+1  ∧  s′=ready ∧  stc′=stc ;
    Terminate ≜
         (if s=idle then idle elsif s=ready then pre else post)=pre
      ∧  id′=id
      ∧  (if s=term then od else aod)′ =
         [2*id[0], 2*id[1],.., 2*id[5] ]
      ∧  (if s=idle then idle elsif s=ready then pre else post)′=post
      ∧  s=ready ∧  sc=6 ∧  od′=od ∧  sc′=sc ∧  s′=term
      ∧  stc′=stc ;
  WF : Start, DoSeg, Terminate ;
end ;
```

Fig. 8 *Integration result* IAppl2

```
process Appl2 ;
  variables
       id, od : array[0.. 5] of integer ;
       sc : 0.. 6 ; s : (idle, ready, term) ; stc : cardinal ;
  Init ≜ s=idle ∧  stc>0 ;
  actions
    Start (p : aoi) ≜
         id′=p ∧  s=idle ∧  sc′=0 ∧  s′=ready ∧  stc′=stc ;
    Reset ≜
         id′=id ∧  s=ready ∧  stc>0  ∧  sc′=0
      ∧  s′=ready ∧  stc′<stc ;
    DoSeg ≜
         id′=id ∧  s=ready ∧  sc<6
      ∧  od′=(od besides od[sc]=2*id[sc])
      ∧  sc′=sc+1  ∧  s′=ready ∧  stc′=stc ;
    Terminate ≜
         id′=id  ∧  s=ready  ∧  sc=6 ∧  od′=od  ∧  sc′=sc
          ∧  s′=term ∧  stc′=stc ;
  WF : Start, DoSeg, Terminate ;
end ;
```

Fig. 9 *Modified integration result* Appl2

Fig. 8 shows *IAppl2*, the result of the mechanical integration. It can be modified interactively by simplifications to form *Appl2* of Fig. 9. Moreover we removed the auxiliary variable *aod*. With respect to the action *Terminate*, we see that the primed value of *od* is constrained twice in Fig. 7 (equations $od' = od$ and $(if\ s = term\ then\ od\ else\ aod)' = [2*id[0], 2*id[1], .., 2*id[5]\,])$. One can prove, that both equations are not in contradiction in this system. The equations are equivalent, the proof is an essential part of the safety proof of *Appl2* $\Rightarrow$*Appl1*. The proof can be facilitated if the pattern *Segmentate* is accompanied in the library by a theorem $\Box(s = term \Rightarrow od = [fs(id[0]), .., fs(id[n-1])])$. A second theorem of the pattern *Segmentate* can state that *Terminate* eventually occurs and facilitates the liveness proof.

```
process ProdCons (tdata : datatype);
  variables psq, bsq, csq : queue of tdata;
         s: (idle, ready, term) ; stc: cardinal;
  Init ≜ s=idle ∧ stc>0 ;
  actions
    Start(p: queue of tdata) ≜
      s=idle ∧ psq'=p ∧ bsq'=empty ∧ csq'=empty
      ∧ s'=ready ∧ stc'=stc ;
    Reset(p: queue of tdata) ≜
      s=ready ∧ stc>0 ∧ psq'=p ∧ bsq'=empty ∧ csq'=empty
      ∧ s'=ready ∧ stc'<stc ;
    Produce ≜
      s=ready ∧ psq≠empty ∧ psq'=tail(psq) ∧ stc'=stc
      ∧ bsq'=insert(bsq,front(psq)) ∧ csq'=csq ∧ s'=ready ;
    Consume ≜
      s=ready ∧ bsq≠empty ∧ psq'=psq ∧ bsq'=tail(bsq)
      ∧ csq'=insert(csq,front(bsq)) ∧ s'=ready ∧ stc'=stc ;
    Terminate ≜
      s=ready ∧ psq=empty ∧ bsq=empty ∧ s'=term ;
  WF : Start, Produce, Consume, Terminate ;
end ;
```

Fig. 10 *Pattern* ProdCons

At next, we integrate the pattern *ProdCons* (Fig. 10) into *Appl2*. It models a producer/consumer interaction. Initially the producer gets a queue of data *psq*, which is the source for producing data items and forwarding them into a buffer *bsq*. Data items are removed from *bsq* and stored in the queue *csq* by the consumer.

```
process Appl3 ;
  variables
    buffer : queue of integer ; s : (idle, ready, term) ;
    stc : cardinal ; id, od : aoi ; ic, oc : 0.. 6 ;
  Init ≜ s=idle ∧ stc>0 ;
  actions
    Start (p : aoi) ≜ id'=p ∧ s=idle ∧ s'=ready ∧ stc'=stc
      ∧ ic'=0 ∧ buffer'=empty ∧ oc' =0 ;
    Reset ≜ id'=id ∧ s=ready ∧ stc>0 ∧ s'=ready ∧ stc'<stc
      ∧ ic'=0 ∧ buffer'=empty ∧ oc'=0 ;
    Produce (p : integer) ≜
         id'=id ∧ od'=od ∧ oc'=oc ∧ s'=s ∧ stc'=stc ∧ s=ready
      ∧ ic<6 ∧ ic'=ic+1 ∧ p=id[ic] ∧ buffer'=insert(buffer, p);
    Consume (p : integer) ≜
         id'=id ∧ s=ready ∧ oc<6 ∧ p=front(buffer)
      ∧ od'=(od besides od[oc]=2*p) ∧ oc'=oc+1
      ∧ s'=ready ∧ stc'=stc ∧ buffer ≠ empty
      ∧ ic'=ic ∧ buffer'=tail(buffer) ;
    Terminate ≜
         id'=id ∧ s=ready ∧ oc=6 ∧ od'=od ∧ oc'=oc ∧ s'=term
      ∧ stc'=stc ∧ ic=6 ∧ buffer=empty ;
  WF : Start, Produce, Consume, Terminate ;
end ;
```

Fig. 11 *Modified integration result* Appl3

The result of the integration of *ProdCons* has been simplified resulting in *Appl3* of Fig. 11. Additionally, parameters were introduced into the actions *Produce* and *Consume* (this will be explained on page 11).

6.4 Composed patterns

We exemplify composed patterns by considering the pattern *DStartTermRes*. The example application *Appl3* (Fig. 11) yet contains non-distributed actions *Start, Terminate*, and *Reset*. *DStartTermRes* provides for the distribution of these actions. It is composed from three resource-processes, the producer (modelled by *ProdState*), the consumer (modelled by *ConsState*), and the medium. The medium is modelled by a set of constraint processes.

The distributed start of an application is performed by a sequence of two actions, either *StartP1* followed by *StartC2* or *StartC1* followed by *StartP2* will happen. Similarly termination and reset are performed by distributed action sequences.

```
process  ProdState ( pdtyp, ptyp : datatype ; p0 : ptyp ) ;
   variables
     pd : pdtyp ;                          ! problem data structure
     pp : ptyp ;                           ! problem state component
     ps : (idle, ready, compl, term) ; ! control state
   Init ≜  ps = idle ;
   actions
     StartP ( nd : pdtyp ) ≜
              ps = idle ∧ ps' = ready ∧ pp' = p0 ∧ pd' = nd ;
     Produce ( np : ptyp ) ≜
              ps = ready ∧ ps' in {ready,compl} ∧ pp' = np ∧ pd' = pd ;
     TermP1 ≜ ps = compl ∧ ps' = ps ∧ pp' = pp ∧ pd' = pd ;
     TermP2 ≜ ps = compl ∧ ps' = term ∧ pp' = pp ∧ pd' = pd ;
     ResetP ≜ ps in {ready,compl} ∧ ps' = ready ∧ pp' = p0 ∧ pd' = pd ;
 end ;
```

Fig. 12 *Process* ProdState *of composed pattern* DStartTermRes

Fig. 12 shows the process *ProdState* of *DStartTermRes*. It models a 'producer' with respect to its state components and actions. The action *StartP* of *ProdState* contributes to *StartP1* as well as to *StartP2*. The actions *StartC1* and *StartC2* of *DStartTermRes* are joined by stuttering steps of *ProdState*.

Since the application *Appl3* not yet has a marked process structure, we used the flat form of *DStartTermRes* for the ongoing refinement of the example application while the compositionality of the specification style would support also that subpatterns of a pattern are integrated into processes of an application. The result of the integration of *DStartTermRes* into *Appl3* has been modified by simplification resulting in *Appl4* (Fig. 13).

```
process Appl4 ;
   variables
     id : aoi ; ic : 0 .. 6 ; ps : (idle, ready, term) ; od : aoi ;
     oc : 0 .. 6 ; cs : (idle, ready) ;
     buffer : queue of integer ; sts : set of (statp, statC) ;
     .....
   Init ≜
     stc > 0 ∧ ps = idle ∧ cs = idle ∧ sts = {} ∧ rs = {} ∧ ms = idle ;
   actions
     StartP1 (p : aoi) ≜ buffer' = buffer ∧ stc' = stc ∧ od' = od
        ∧ ps = idle ∧ ps' = ready ∧ cs' = cs ∧ ic' = 0 ∧ id' = p
        ∧ oc' = oc ∧ sts = {} ∧ sts' = {statP} ∧ rs' = rs ∧ ms' = ms ;
     StartP2 (p : aoi) ≜ stc' = stc ∧ buffer' = empty ∧ ps = idle
        ∧ ps' = ready ∧ ic' = 0 ∧ id' = p ∧ oc' = oc ∧ cs' = cs
        ∧ buffer' = empty ∧ sts = {statC} ∧ sts' = {statP, statC}
        ∧ rs = {} ∧ rs' = rs ∧ ms = idle ∧ ms' = ready;
     Produce (p : integer) ≜
        od' = od ∧ stc' = stc ∧ ic < 6 ∧ p = id[ic]
        ∧ buffer' = insert(buffer,p) ∧ ps = ready ∧ ps' = ready
        ∧ ic' = ic+1 ∧ id' = id ∧ oc' = oc ∧ cs' = cs ∧ sts' = sts
        ∧ rs ≠ {resatP} ∧ rs' = rs ∧ ms = ready ∧ ms' = ms ;
     .....
   WF : StartP1, StartP2, StartC1, StartC2, Produce, Consume,
        ResetP2, ResetC2, TermP1, TermC1, TermC2, TermP2 ;
 end ;
```

Fig. 13 *System* Appl4

6.5 Process splitting

```
split Appl4 ;
   processes  P has ps, id, ic ;
              C has cs, od, oc ;
              M has stc, buffer, rs, ms, sts ;
 end ;

process P ;
   variables  id : aoi ; ic : 0 .. 6 ; ps : (idle, ready, term) ;
   Init ≜ ps = idle ;
   actions
```

```
      StartP1 ( p : aoi) ≜ 1
        ps=idle ∧ ps'=ready ∧ ic'=0 ∧ id'=p ;
      StartP2 ( p : aoi) ≜
        ps=idle ∧ ps'=ready ∧ ic'=0 ∧ id'=p ;
      StartC1 ( p : aoi) ≜
        id'=id ∧ ic'=ic ∧ ps'=ps ;
      StartC2 ( p : aoi) ≜
        id'=id ∧ ic'=ic ∧ ps'=ps ;
      Produce ( p : integer) ≜ ic<6 ∧ p=id[ic]
        ∧ ps=ready ∧ ps'=ready ∧ ic'=ic+1 ∧ id'=id ;
        .....
    WF : StartP1, StartP2, StartC1, StartC2, Produce, ..... ;
  end ;
  process C ;  .....  process M ;  .....
  process SAppl5 ;
    processes P : P ; C : C ; M : M ;
    actions
      StartP1(p : aoi) ≜ P.StartP1(p) ∧ C.StartP1(p) ∧ M.StartP1(p);
      StartP2(p : aoi) ≜ P.StartP2(p) ∧ C.StartP2(p) ∧ M.StartP2(p);
      Produce(p : integer) ≜ P.Produce(p) ∧ C.Produce(p) ∧ M.Produce(p);
      .....
  end ;
```

Fig. 14 SAppl5 - *Splitting of* Appl4 *and result*

To exemplify process splitting, we split *Appl4* (cf. Fig. 13) into three components: a producer process *P*, a consumer process *C*, and a medium process *M*. Fig. 14 shows the corresponding split statement and result. The integration of *DStartTermRes* (cf. page 9) introduced the structuring of the system into the three processes. It also introduces a corresponding allocation of variables. The variables *id*, *ic*, and *ps* represent the state of *P*, the variables *od*, *oc*, and *cs* the state of *C*. All other variables represent the state of the medium *M*.

Note, that we introduced parameters *p* for the actions *Produce*, and *Consume* in *Appl3* (cf. Fig. 11). The parameter introduction supported the process splitting. Originally, the action *Consume* of *IAppl3* contained the predicate $od' = od$ besides $od[oc] = 2 * front(buffer)$ which references private variables of *C* as well as private variables of *M*. By means of the parameter *p* the predicate has been splitted equivalently into the two predicates $od' = od\ besides\ od[oc] = 2{*}p$ and $p = front(buffer)$ which only reference private variables of one process.

With respect to the distribution of actions, the splitting seems to fail. Each process contributes to each system action by a component action while the process system intended shall provide for actions which are local at one site (i.e., if *P* contributes to a system action, *C* has to perform a stuttering step). We look more carefully to *SAppl5*. In fact, the actions *StartC1*, *StartC2*, *Consume*, *ResetC1*, *ResetC2*, *TermC1*, and *TermC2* of *P* are equivalent to stuttering steps of *P*. They can be replaced by *P.Stutter* and removed in the definition of *P*.

The last problem concerns the two-phase execution of start and reset. Globally there is a sequence of two actions, one action at producer's site and one at consumer's site. Locally, an application process is not able to know, if it performs its local action at first or at second while the definition of *P* makes a distinction between these two cases (e.g., two actions *StartP1* and *StartP2* are defined). We recognize that the two actions are pairwise equivalent. Each pair can be merged to one action (e.g., *StartP1* and *StartP2* can be replaced by one action *StartP*).

6.6 Communication service integration

The next two subsections illustrate the benefit of tool support when manipulating larger process patterns. They also serve as an example for a productive pattern library utilization.

At next, the application shall use a special communication service. The service *Transfer* does not directly comply with standards. However, it refers to the ISO/OSI-concepts of session service connections and of unconfirmed respectively confirmed service elements. Service elements are represented in terms of two respectively four service interface events "request→indication" respectively "request→indication, confirmation←response" (cf. [11]).

A pattern *TransferPCCS* is used which describes a system consisting of a service provider *Med*, a data packet producing client *Cli*, and a consuming server *Ser*. The integration of *TransferPCCS* into the simplified result of process splitting (cf. Fig. 14) is performed by separately but consistently integrating

Med to *M*, *Ser* to *C*, and *Cli* to *P*. Since we assume an existing implementation of *Transfer*, the design can concentrate on the two application processes. The resulting producer process *P6*, is shown in Fig. 15. It is a producer process of our application as well as it is a user process of *TransferPCCS*.

```
process P6 ;
  variables
    s : (idle, nocon, incon, conn, inrel, term) ; id : aoi ; ic : 0..6;
  Init ≙ s=idle ;
  actions
    Start (p : aoi) ≙ id'=p ∧ s=idle ∧ s'=nocon ∧ ic'=0 ;
    ConReq ≙ id'=id ∧ ic'=ic ∧ s=nocon ∧ s'=incon ;
    ConCon ≙ id'=id ∧ ic'=ic ∧ s=incon ∧ s'=conn ;
    DatReq (p : usd) ≙
      id'=id ∧ ic<6 ∧ p=id[ic] ∧ ic'=ic+1 ∧ s=conn ∧ s'=conn;
    RelReq ≙ id'=id ∧ ic'=ic ∧ s=conn ∧ ic=6 ∧ s'=inrel ;
    RelCon ≙ id'=id ∧ ic'=ic ∧ s=inrel ∧ s'=term ;
    AboReq ≙
      id'=id ∧ ic'=0 ∧ s in {nocon,incon,conn,inrel} ∧ s'=nocon;
    AboInd ≙
      id'=id ∧ ic'=0 ∧ s in {nocon,incon,conn,inrel} ∧ s'=nocon;
  WF : Start, ConReq, ConCon, DatReq, RelReq, RelCon, AboInd ;
end ;
```

Fig. 15 *Service using producer* P6

We integrated a process pattern modelling the abstract communication pattern (abstract service) of an implementation building block to be integrated in the next step. As the abstract service and the implementation building block are formally related by a refinement, we are assuring that the correct abstract behaviour neccessary is modelled as result of this step.

6.7 Application program interface integration

At last, the producer process of our example shall be refined into a process which applies a special application program interface (API) to an implementation of the service *Transfer*. The application program interface used has a procedure interface, the exchange of service events is represented by invocations and returns of interface procedures. Events which are passed from the user process to the service implementation correspond to procedure calls, events which are passed in the opposite direction correspond to returns of interface procedures, e.g., a call of the procedure *Connect* corresponds to a *ConnectRequest* event, a return of *Connect* corresponds to *ConnectConfirmation* or if the connection is aborted during establishment to *AbortIndication*. The procedure calls have to be parametrized by a data structure *cb* which is used internally by the service implementation. Before any events can be exchanged, the user process has to install *cb* by the invocation of the procedure *Initiate*.

The pattern *ClientAPI* is a refinement of the subpattern *Cli* of *TransferPCCS*. It describes the possible temporal orderings and parametrizations of API procedure calls and has been integrated to *P6* (Fig. 15) resulting in *P7* (Fig. 16). The new variables stem from *ClientAPI*: *pins* and *pouts* represent the actual parameter values of procedure invocations, *cproc* and *prsta* document which procedure is called respectively has returned.

P6 integrates the required properties of a producer of our example application as well as it assures the correct integration of the local API. Since the API handling is described in detail, it is a suitable starting point for the implementation.

```
process P7 (d : aoi ; apicbtype, adrtype : datatype ;
        ownadr, peeradr : adrtype ; length : function) ;
  variables
    id : aoi ; ic : 0 .. 6 ;
    s : (idle,wini,ready,wconn,conn,wsend,wrel,term,wtest,wabo) ;
    cb : apicbtype ; pins, pouts : array cardinal of any ;
    cproc : literal ; prsta : (called, returned) ;
  Init ≙ s=idle ∧ cproc=none ∧ prsta=returned ;
  actions
    InitiateCall ≙ s=idle ∧ s'=wini ∧ cb'=cb
      ∧ cproc'=INITIATE ∧ prsta'=called
      ∧ pins[0]'=cb ∧ pins[2]'=ownadr ∧ id'=d ∧ ic'=0 ;
    InitiateReturn ≙ s=wini ∧ s'=ready ∧ cproc'=cproc
      ∧ prsta'=returned ∧ cb'=pouts[0] ∧ id'=id ∧ ic'=ic ;
    ConnectCall ≙ s=ready ∧ s'=wconn ∧ cb'=cb
```

```
      ∧ cproc'=CONNECT ∧ prsta'=called
      ∧ pins[0]'=cb ∧ pins[1]'=peeradr ∧ id'=id ∧ ic'=ic ;
    ConnectReturnOk ≙
      s=wconn ∧ s'=conn ∧ cproc'=cproc ∧ prsta'=returned
      ∧ cb'=pouts[0] ∧ pouts[1]=connected ∧ id'=id ∧ ic'=ic ;
    ConnectReturnAbo ≙
      s=wconn ∧ s'=ready ∧ cproc'=cproc ∧ prsta'=returned
      ∧ cb'=pouts[0] ∧ pouts[1]=aborted ∧ id'=id ∧ ic'=0 ;
      .....
end ;
```

Fig. 16 *Integration result* P7

Due to integration of a process pattern modelling the correct driving of a special platform, we benefited to be sure about the correct usage of this API in our resulting application.

7 Conclusion

We proposed operations for pattern integration and process splitting which support the computer assisted construction and writing of specifications of distributed applications as well as the understanding and verification of design steps. As the experience of the development of practical computer network applications shows, the dominating design task is not the design of 'new' algorithms, but the design of consistent combinations of known algorithms. Moreover, the introduction of the detailed interface mechanisms to application platforms is important. Both types of design steps correspond well to the integration of process patterns. Furthermore tool support has been realized which mechanically produces major parts of specification texts.

Until now, we studied the application of the approach by means of few 'academic' and two more complex examples. With respect to assistance by tools, we resorted to a set of general TLA tools [16] (editor, browser, interpreter/animator, refinement mapping searcher, refinement mapping checker, model checker, frontend for a general automated predicate logic theorem prover). Additionally, we used prototypical tools which perform the text manipulation operations (combine, integrate, compose, split and refine).

Present work concentrates on the one hand on the extension of the libraries and the examples. While large productive distributed applications are in the scope of our work (e.g., applications of computer integrated manufacturing, administration, and commerce), we firstly collect experiences by means of smaller applications. So, agents for the Internet applications FTP and SMTP are under development.

On the other hand, we want to enhance the tool support. During the construction of a specification, information can be recorded which will be utilized to prepare the application of our TLA verification tools, esp. the refinement mapping checker. Additionally, a tool is under development which performs simplifications of finite state processes automatically in order to enhance the support of pattern integration. The tool utilizes process composition and process splitting. It firstly splits a system into a finite state process and a 'remainder' process. The finite state process is analysed and simplified by reachability graph and term rewriting methods. At last, a composition joins the processes.

Acknowledgements. Thanks to the anonymous referees for their helpful comments.

References

[1] Martin Abadi, Leslie Lamport. Conjoining Specifications. Research Report 118, Digital Equipment Corporation, Systems Research Center, December 1993.

[2] Martin Abadi, Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[3] Friedrich Ludwig Bauer, Bernhard Möller, Helmut Partsch, Peter Pepper. Formal Program Construction by Transformations — Computer Aided, Intuition-Guided Programming. *IEEE Transactions on Software Engineering*, 15(2):165–180, February 1989.

[4] T. Bolognesi (Ed.). Catalogue of LOTOS Correctness Preserving Transformations. Final report. Deliverable Lo/WP1/T1.2/N0045/V03. ESPRIT Project 2304 (Lotosphere). January, 1992.

[5] T. Bolognesi, et al. Correctness Preserving Transformations for the early phases of software development. to appear in: LOTOSphere – software development using LOTOS, Kluwer Academic.

[6] Ed Brinksma, Rom Langerak, Peter Broekroelofs. Functionality Decomposition by Compositional Correctness Preserving Transformation. in: Costas Courcoubetis (Ed.), Proc. of the 5th Int. Conf. *Computer Aided Verification*, CAV '93, Elounda, Greece. 371–384, Springer, 1993.

[7] Citibank Distributed Processing Technology. Objtran Programmer's Guide. Technical Document, December, 1993.

[8] Claus Dendorfer, R. Weber. From Service Specification to Protocol Entity Implementation. in: R.J. Linn,Jr. and M.Ü. Uyar (Eds.) *Protocol Specification, Testing, and Verification XII.* 163–178, Elsevier, 1992.

[9] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: Abstraction and Reuse of Object-Oriented Design. in: O.M. Nierstrasz (Ed.) *7th European conference on Object oriented programming*, Kaiserslautern, Germany Springer, 1993.

[10] Peter Herrmann, Heiko Krumm. Compositional Specification and Verification of High-Speed Transfer Protocols. to appear in: S.T. Vuong and S.T. Chanson (Eds.) *Protocol Specification, Testing, and Verification XIV.* Chapman & Hall, 1994.

[11] ISO. International Standard: Basic Reference Model for Open Systems Interconnection. ISO: IS7498.

[12] ISO. *LOTOS: Language for the temporal ordering specification of observational behaviour.* International Standard ISO/IS 8807, 1987.

[13] R. Kurki-Suonio, H.-M. Järvinen. Action system approach to the specification and design of distributed systems. *ACM Transactions on Programming Languages and Systems*, 4(10):510–54, October 1988.

[14] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[15] William Leddy, Arjun Khanna. DCE++: A C++ API for DCE. Technical Document 030-00209, Hal Computer Systems, Inc. March, 1993.

[16] Arnulf Mester, Peter Herrmann. Tools for TLA-based Specifications. Universität Dortmund, Fachbereich Informatik, Lehrstuhl IV. Technical Report RvS-TLA-94/35, 1994.

[17] A. Mester, H. Krumm. Composition and Refinement Mapping Based Construction of Distributed Applications. Universität Dortmund, Fachbereich Informatik. Research Report/Forschungsbericht No. 548, 1994. `WWW: http://ls4-www.informatik.uni-dortmund.de/RVS/Exports/FB_548.ps.Z`, `FTP: ftp.informatik.uni-dortmund.de` in `/pub/ls04-info/research-reports`

[18] Max Mühlhäuser, W. Gerteis, L. Heuser. DOCASE: A Methodic Approach to Distributed Programming. *Communications of the ACM*, 36(9):127–138, Sept. 1993.

[19] Douglas C. Schmidt. The ADAPTIVE Communication Environment. Technical Report, Dept. ICS, University of California, Irvine, 1994. also: ASX: an Object-Oriented Framework for Developing Distributed Applications. in: *USENIX C++ Technical Conference* April, 1994.