Addressing performance requirements in the FDT-based design of distributed systems

Jeroen Schot proposes a distributed systems design method which incorporates performance constraints

The development of distributed systems is generally regarded as a complex and costly task, and for this reason formal description techniques such as LOTOS and ESTELLE (both standardized by the ISO) are increasingly used in this process. Our experience is that LOTOS can be exploited at many stages on the design trajectory, from requirements specification to implementation, but that the language elements do not allow direct formalization of performance requirements. To avoid duplication of effort by using two formalisms with distinct approaches, we propose a design method that incorporates performance constraints in an heuristic but effective manner.

Keywords: performance requirements, FDT, distributed systems design

This paper discusses a methodology for communications systems design which aims to incorporate performance requirements in the design process using a formal language like LOTOS[1]. We use LOTOS to describe those system properties that are considered to determine most of the complexity of the design and implementation process. This complexity stems from the fact that communications systems are distributed and concurrent, i.e. many events are taking place in parallel, some of them are related (in time) while others are independent. It is commonly recognized that these systems are hard to comprehend, and that their design is a vast task.

Standardization bodies (ISO, CCITT) cope with the complexity problem by using architectural models, while more and more they favour the use of FDTs for the unambiguous description of protocols and services that are identified in the models. This approach can be adopted by the communications industry, and the formal specifications provided by standardization groups can serve as a starting point for the development process of communications equipment. A first step in this approach is defining the top-level description of the system under design, which is to be derived by a process of requirements capturing. In this top-level description, LOTOS can be used to define the 'functional' behaviour of the system in terms of *events* (i.e. interactions between the system and its environment), their *ordering*, and related *parameters* whose values are exchanged in the events. This formal specification bears some significant advantages for the designer:

- it can act as an interface between the manufacturer and the customer, since it exactly describes the 'functional' properties of the required system;
- these properties can be assessed using appropriate tools that allow verification of the syntactic and (static) semantic properties of the specification, and symbolic execution to assess its behaviour[2];
- it can be used as a starting point for further design, i.e.implementations should be correct with respect to this description, and a conformance test suit can be derived from it.

Tele-Informatics Group, University of Twente, PO Box 217, 7500 AE Enschede, The Netherlands

0140-3664/92/004235-08 © 1992 Butterworth-Heinemann Ltd

Apart from the requirements that are defined by the LOTOS specification, there may be other requirements like performance constraints that are relevant for the system under design. These non-LOTOS requirements should be included in the top-level description using other formalisms or natural language, and their fulfilment should be achieved in successive design steps. In fact, the latter consideration agrees with the commonly accepted idea that a design process is not performed in a single step, but should progress through a series of steps where system requirements are incrementally incorporated in the design, i.e. in each consecutive step particular design concerns are addressed so that the implementation is gradually shaped. As a consequence, the description of the system at level $N + 1$ is of a lower abstraction level (i.e. more implementation-oriented) than the description of the system at level N. This design process is depicted in Figure 1.

It would be beneficial if a single FDT could be applied for a large part of the design trajectory, hence procuring the predicate *broad-spectrum language*. However, we can identify two *weak links* in Figure 1, *viz.* in the first step where we enter the LOTOS domain, and the point where we leave LOTOS and enter the hard- and software domains. For these steps we also present here some methodological support with respect to considering performance requirements. What remains are transformations between LOTOS specifications that should respect performance concerns, which is elaborated on below. Prior to this we discuss the operational aspects of the design process in more detail.

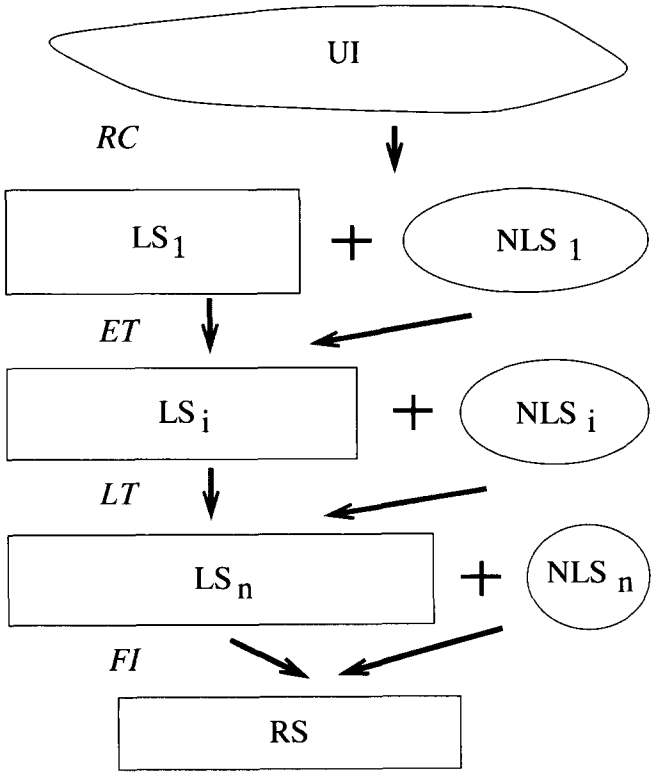

Figure 1. Generalized design trajectory. UI: user ideas; LS: LOTOS specification; NLS: non-LOTOS specification; RS: real system; RC: requirements capturing; ET: early transformations; LT: later transformations; FI: final implementation

The ideas presented are illustrated by examples. One of these, originated in the ESPRIT project PANGLOSS[3,4], is a switching system for telephony traffic (*viz.* a PABX). Such a system is indeed a realistic example for at least two reasons: first, the design of a modern switching system that supports a large number of connections and incorporates advanced features (supplementary services) has proven to be a costly and manpower-consuming task; second, it is a part of a distributed system (the telephony network), and displays concurrency, for example the support of many connections in parallel (and to a certain extent independently)[5,6].

GENERIC DESIGN METHODOLOGY

The generic design methodology presented here partly evolves from the LOTOS-based design methodologies developed and applied in the ESPRIT PANGLOSS[3] and LOTOSPHERE[7] projects. We briefly discuss some characteristics of this methodology in this section.

Cyclic approach

Our systematic design method should limit the cost of the design process by managing its complexity. One result of this is that the design trajectory as sketched in Figure 1 should not be entered with all the functionality that may be required for the final implementation. A subset out of the total functionality in which so called *key functions* are preserved should be determined. The design trajectory will then be traversed several times (in so called *cycles*), and in each cycle extra functionality is included until the fully-fledged implementation is obtained. This is called the *cyclic approach*, and is a major characteristic of our methodology[3,7]. In this approach it is essential that the selected key functions are key (or basic) in the sense that they are mainly responsible for the structure of the final implementation. In the switching system example, *basic interconnection* (i.e the capability to handle calls) has been selected as a key function. Additional functions such as call barring, call redirecting, call forwarding, and conference calling can be based upon this key function.

Stepwise refinement

In the refinement steps following requirements capturing, the top-level specification is stepwise transformed into specifications that are more implementation-oriented. In other terms, the top-level specification is an abstraction of all possible implementations of the system. However, only a small subset of this set contains implementations that also fulfil the non-LOTOS requirements. This is illustrated in Figure 2.

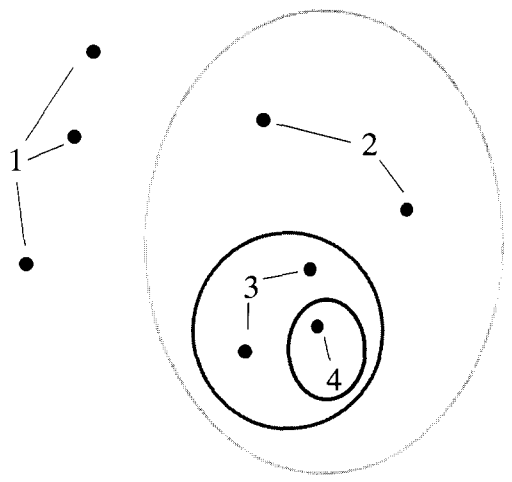

Figure 2. Stepwise refinement. 1: elements from the universe of implementations; 2: elements from the implementations defined by step **1**; *3: elements from the implementations defined by step* **i**; *4: element from the implementations defined by step* **n**

In each transformation step we take design decisions that will be reflected in the real system, and they should be based on one or more non-formalized requirements. In which order we consider these requirements is determined by their relative importance. For example, when high-performance is a severe demand, we should address it early in the design trajectory. Note that from Figure 2, it appears that the order in which the reduction of the set of correct implementations is performed plays no role. In practice, however, arbitrary sequences may lead to unnecessary backtracking[8], i.e. at level j it may appear that the next non-formal requirements to be considered cannot be fulfilled in this transformation step, and the design process has to be resumed from level i, where $i < j$.

Bottom-up knowledge

In the above we sketched the design process as a strictly top-down activity. This more or less applies to the early transformation steps, in which design decisions are mainly taken on the basis of generic structuring principles such as 'divide and conquer', and 'separation of concerns'. Later on, we have to take more implementation aspects into account, in particular when going from LOTOS to hard- and software elements. This step can, for example, be performed by generating code from LOTOS specifications (the compiler approach); also 'predefined implementation elements', which are LOTOS descriptions of generic hard- and software constructs, can be used for the implementation of LOTOS specifications[9]. In fact, these elements are direct representatives of implementation knowledge, thus go bottom-up. Since the performance of a system can only be determined after it has been built with hard- and software, we will show that in this case implementation knowledge already comes into play at the beginning of the design trajectory.

Correctness preserving transformations

One advantage of FDTs like LOTOS is that their formal semantics allow correctness verification between specifications. For the design method this means that in the stepwise refinement process, where we define several specifications of the system under design at different levels, we require a specification at level $i + 1$ to be (implementation) equivalent to a level i specification. For a behaviour description of realistic size, however, this verification process is beyond feasibility. Hence we prefer to transform a specification according to procedures that have been shown to preserve correctness[10,11], and that may be supported by software tools. In the cases of predefined implementation elements and the compiler approach, this condition is already fulfilled. For requirements capturing we can only validate its 'correctness' by simulation. For heuristic transformation steps and transformations that are not yet supported by formal procedures simulation is the only method.

PERFORMANCE ISSUES IN ARCHITECTURE DEFINITION

According to Figure 1, and what has previosuly been discussed, the first step should result in a formal specification of the system under design in LOTOS, annotated with requirements that cannot be formalized in LOTOS. The latter part could be defined using other formalisms (e.g. to express quantitative requirements) and/or using a natural language (e.g. to express the remaining requirements).

The LOTOS part contains a specification of the functional behaviour of the system at the highest level of abstraction. It does not tell *how* system functions are to be performed, but *what* functions should be performed. Its 'correctness' can be validated using suitable simulation tools[2]. In dialogue with the customer, adjustment of the specification can be performed, if necessary. An external description of a system refraining from implementation structure is also termed a *Black Box* description.

For the LOTOS part of the top-level description it is essential to preserve the concurrency or parallelism that is intrinsically present in the user requirements. When the parallelism is not preserved, it cannot be exploited in the implementation of the system, since this information will be lost forever. This is also in conflict with high-performance constraints, because parallelism is a means to achieve high performance at the implementation level. In the switching system example, this rule applies as follows: the switching system should support a large number of connections, say N, simultaneously and in

principle independently. We can now describe this system in terms of a single connection, without compromising its correctness. The number N is determined by the amount of available resources within the system, and defines the only relation between connections. This is illustrated in the following LOTOS specification, where the process of N connections is refined into the definition of a single connection (using the ||| operator, which expresses independent parallelism), and a resource limitation constraint (using the || operator, i.e. logical **and**). In this specification we obeyed the *constraint-oriented* specification style (CO style), which is one of several styles that are identified in the literature[12]. For a top-level specification, this style is favoured, since it allows one to describe the behaviour of the system as a conjunction of constraints which are logically separated, based on architectural criteria such as orthogonality, modularity, generality, openendedness, and parsimony. Structuring according to the CO style is particularly enabled by the LOTOS parallel composition operator:

```
process PABX [g] : noexit :=
   choice max_number : NAT [] i;
   (*determined by the implementation*)
   (Infinite_Connections [g] || Resources [g]
   (max_number)) where
   process Infinite_Connections [g] : noexit :=
      One_Connection [g] ||| Infinite_Connections [g]
   endproc (*Infinite_Connections*)
   process One_Connection [g] : noexit :=
   (*starting point for further definition*)
   enproc (*One_Connection*)
   process Resources [g] (remain : NAT) : noexit :=
   g ? adr : Addr ? sp : SP [IsCallSetup(sp) implies
   (remain gt 0)];
      ([IsCallSetup(sp)] -> Resources [g] (remain - 1)
      [] [IsCallRelease(sp)] -> Resources [g]
      (remain + 1)
      [] [not(IsCallSetup(sp) or IsCallRelease(sp))] ->
      Resources [g] (remain))
   endproc (*Resources*)
endproc (*PABX*)
```

The switching system specification presented above follows the architectural principle of *orthogonality*, which is defined as *do not relate what is independent*. This principle can thus be used to avoid the *state space explosion* of a concurrent system to blow up our specification. For example, suppose a connection is characterized by **i** states, then the global state of the system would be one out of $\mathbf{i}^N$ states. The CO style allows this separation of concerns. It also supports the cyclic approach: when new functionality is to be included in a subsequent cycle, we are not forced to completely rewrite our specification, but we can do with the inclusion of process definitions that add new constraints on the behaviour of the system. This in contrast with, for instance, Petri-nets, where there is a fair chance that one has to break open the complete transition network to include new states. From the viewpoint of cost and reusability (in general, quality of the design process), this is an important property. In fact, we do not wish to describe the states explicitly in the specification, and this can be achieved by using the CO style. We experienced that the removal of concurrency (i.e. sequencing) is an irreversible process, and therefore should be avoided if no justification can be found.

We have now obtained a formal specification of the functional behaviour of the system under design which preserves the parallelism that can be found in the user requirements. What remains are requirements that cannot be formulated directly in LOTOS. Examples of such requirements are:

- 'A call should be setup through the switch *within* **t** milliseconds' — this is a real-time constraint for which our method is not applicable. The real-time constraints and the functional behaviour described in the LOTOS specification are directly related, thus cannot be handled separately.
- 'The *mean* time for call setup through the switch should be **t** milliseconds' — note that this requirement is not a real-time constraint, but a stochastic constraint referring to probability distributions with mean values, variances, and the like. These requirements are addressed in our method.
- 'The probability of a call refusal due to system failures should be less than 1%' — this is a *realiability* requirement, and if it is expected that it will be difficult to meet this constraint in the implementation, we have to consider it in the design process in the way we deal with performance issues in this paper. This is not worked further here.
- 'The system should be unobservant' — this requirement can be fulfilled in the last transformation step where we leave LOTOS and enter the hard- and software domain, by dazzle-painting its enclosure with army colours.

For the first kind of requirements we have to use another FDT that can express behaviour in relation to real-time. For this purpose, Petri-nets with value passing may be used. However, they are not suited for application at different levels of abstraction, hence cannot be classified a broad spectrum language, and do not support the constraint-oriented specification style. For the second and third kind of requirements we present method support in the forthcoming sections. The fourth kind of requirements will be maintained during the transformation trajectory and fulfilled in the realization phase.

PERFORMANCE ISSUES IN THE EARLY TRANSFORMATIONS

Several types of transformation steps have been defined and explored in the literature. Associated with each transformation type is a (formal) *equivalence* or *implementation relation* which defines the correctness of the transformed specification with respect to its original. A particular type of transformation step, which is predominant in the first phase of the design process, is *process-* or *functionality-decomposition*. In this transformation a single process definition is decomposed into

several processes that mutually communicate through *internal gates*. This process structure will be reflected in the final implementation, hence we are taking design decisions. Possible relations for this transformation step include *weak bisimulation equivalence* and *testing equivalence*[13-15]. Some formalized procedural support can be found[10,11].

Functionality decomposition

Process decomposition should be guided by criteria that are derived from the non-formalized requirements. In our case, high-performance constraints have a high priority and therefore should direct the decomposition step, so that the class of implementations that are still permitted by the resulting specification contains all high-performance systems.

One criterion that could control the distribution of functions over distinct processes is the *implementability* of a certain function class. In our method we present three types of functions that can be considered as generalized implementation elements (composed with von Neumann):

- *transformation functions* (TF) which are able to perform operations on data, but are equipped with a limited number of communication ports, and which can be implemented by processors, transputers, etc.
- *communication functions* (CF) which are able to transport data among a large number of communication ports, but do not transform the data they transport, and which can be implemented by local networks, computer busses, switching circuits, etc.
- *storage functions* (SF) which are able to store large amounts of data for a certain period, but do not transform the data, and which can be implemented by memory components, disks, cartridges, etc.

The approach we follow here is to decompose those processes for which a mapping to implementation elements cannot be found easily into processes that display the characteristics defined above. We then obtain a structured specification in terms of TFs, CFs and SFs that is expected to be more implementable., Before we illustrate such a transformation with our example, we note that those types of functions that emerge as potential performance bottlenecks in the final implementation should especially be considered. For the switching system, this means that the decomposition should be in terms of transformation functions (used for call handling) and communication functions (used for transfer of user data, e.g. voice). In other applications, for example a distributed database or electronic mail system, storage functions could be more appropriate. Furthermore, each of these functions can again be implemented by a set of TFs, CFs and SFs.

In Figure 3, process decomposition is applied to the switching system example. The single process that represents all functions of the switching system is refined into a structured process consisting of two subprocesses.

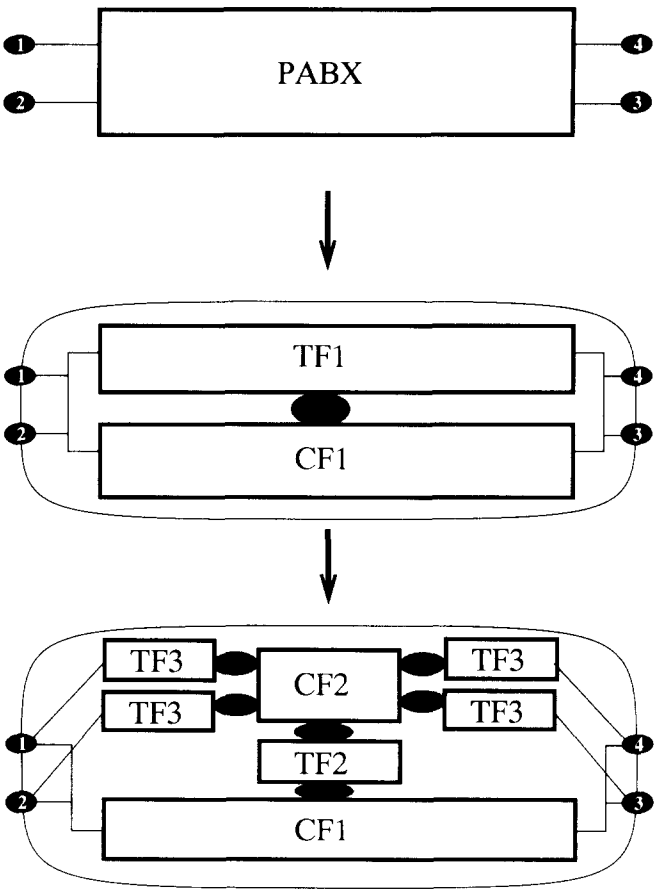

Figure 3. Decomposition of the PABX. TF: transformation function; CF: communication function; ● : internal gate; ●: external gate (in this example four)

One process (the TF) is concerned with call handling, and one process (the CF) with the data phase of the calls. They communicate through an internal gate (the TF 'controls' the CF). In a next step the call handling function is further refined into a number of transformation functions. The peripheral TFs are concerned with those functions that are local to a single connection, thereby reducing the complexity of the centralized TF, which performs all remaining call handling functions. However, in all cases we need an extra CF, since the TF is defined as having only a limited number of communication ports.

Structure assessment

When successively performing process decomposition transformations, it is likely that in each of these transformation steps multiple alternative decomposition structures can be considered, all appearing to fulfil the non-formalized requirements that were input to the transformation. However, some may be easier to implement, or turn out to be more economical solutions. One would already like to be able to judge their feasibility in the early stages of design, without going through the costly process of implementing all potential structures.

Below we demonstrate how such an assessment can be made.

As stated earlier, a design at level N will consist of a formal specification in LOTOS and non-LOTOS requirements, e.g. related to performance. An example of a performance requirement is a statement about the mean time that should elapse between an event a and a related event b. For the design at level $N + 1$, this delay demand is still valid, but we can derive additional timing requirements that apply to the composite processes of the refined specification. As an example, consider Figure 4.

On the single process at level N we imposed the constraint that out should follow in after an average interval of **t** milliseconds. This is a basic performance requirement, and from queueing theory we can immediately deduce that in case buffering is included in the process, and the in events arrive according to a Poisson distribution with a mean interarrival time of λ, the mean number of packets in transit is $\mathbf{t} * \lambda$ (1); in case packets are also processed according to a Poisson distribution with a mean processing time of μ, this mean number of buffered packets is also equal to $\rho/(1 - \rho)$ (2), where $\rho = \lambda/\mu$, and a combination of (1) and (2) yields $\mathbf{t} = 1/(\mu - \lambda)$. In the structured specification, this constraint is still applicable, but can be fulfilled by separate constraints put upon the composite processes. As a result, we obtain that $\mathbf{t} = t_1 + t_2$, but also $\lambda = \lambda_1 = \lambda_2$, and for each process again the number of packets buffered equals $t_i * \lambda$, and in case of Poisson processing $t_i = 1/(\mu_i - \lambda)$. In our method we may now decide to either select a particular value for t_1 and t_2 (e.g. based on implementation knowledge), or defer this decision to a later stage.

In more realistic examples, the structure will be far more complicated, and the derivation of the performance requirements per process may not follow directly from the structure. Also, we may have several alternatives that we want to compare. Fortunately, we have specified the behaviour of the processes before and after decomposition using a formal language, so the relation between all events is unambiguously defined. Now a queueing network can be built that corresponds to the process structure obtained after decomposition. For a queueing network that is to be evaluated analytically, the building-bricks (see Figure 5) should meet the following conditions[16]: the arrival and service times are Poisson distributed, the queueing discipline is *first come first served* (FCFS) or, in the case of a single server, *last come first served* (LCFS) and *processor sharing* (PS) are also allowed; a component has one or more inputs, and one or more outputs with a probability for each output (and $\Sigma P_i = 1$); the number of servers may be infinite (a single server for each packet, i.e. no queueing). In case these conditions are not fulfilled, the composed queueing network may be simulated using the appropriate tools[17].

The approach presented here enables us to select the most appropriate structure among several alternatives by evaluating the implementability of the performance constrains for each of the composite processes, which are derived by mapping the process structure on a queueing network. Thus we can take an implementation decision without having to go through the complete design trajectory and make an *a posteriori* assessment.

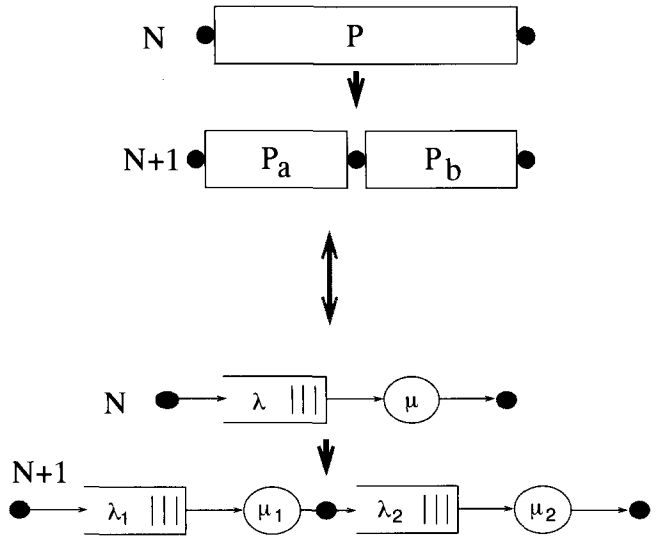

*Figure 4. Example mapping of process structures on queueing networks. P: in; out (*mean* **t** *mS*); P_a: in; mid (*mean t_1 mS*); P_b: mid; out (*mean t_2 mS*)*

PERFORMANCE ISSUES IN LATER TRANSFORMATIONS

Transformations that are usually applied in later steps of the design trajectory are, amongst others, *interaction point decomposition* and *event refinement/integration*. They are used to achieve the mapping of abstract interfaces on real interfaces, in other words, the final implementation of the abstract interfaces defined in a given specification. Another transformation we will briefly discuss here is *process integration*.

Interaction point decomposition

In Figure 3 we see that the interactions between the call-handling functions, the data transfer process, and the systems environment take place at one (of four) common external gates. When the transformation of *interaction point decomposition* is applied here, we would obtain separate gates for the exchange of call-handling information and user data. This is again a step towards implementation, and allows us, for example, to model out-of-

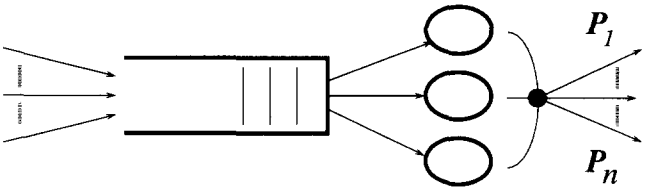

Figure 5. Building brick for queueing networks

band signalling, as is common in telephony networks. Note that at the highest abstraction level these two information streams are not separated, since they are related to the same call, i.e. user data can only be exchanged after a call setup has been granted.

In the final transformation steps the interaction points, their related events and parameters should find corresponding elements in the implementation. In general, we will select appropriate call structures, such as interrupts, procedure calls and system calls, but also semaphores and hardware interfaces, to implement the interactions, and the efficiency of their implementation will affect the final performance of the system. Also, related mechanisms like buffering at a local interface play a role in this respect. We will not discuss this further here.

Event refinement and integration

An example of event refinement is illustrated in the left part of Figure 6, where a single event E(p) (*p* are its parameters) is decomposed into a request, in which E(p) is offered, and a confirmation or rejection of this event-offer. This is common practice when going to an implementation level, since synchronous communication (with value checking, passing, or generation) is scarcely supported by current hardware and software. Similar motivations for event refinement apply to the implementation of multi-party events by means of two-party events.

The right part of Figure 6 indicates how a connect-request (cr) primitive is refined into an r(cr) and a c(cr), and similarly for the connect confirm (cc) and data_request (dr) primitives that normally follow a connect-request. What we note here is that for the complete connection-setup and issuance of the first data primitive, six refined events are used at implementation level. It may now be the case that some events can be integrated, in the sense that they collapse into a single event. This is called *event integration* or *event grouping*, and can contribute to achieving a better performance in the final system.

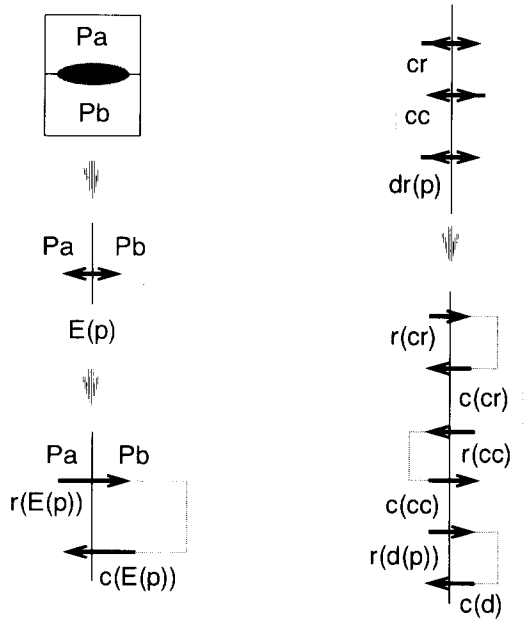

Figure 6. Example of event refinement. E(p): event with p being established c.q. exchanged; r(x): request for x; c(x): confirmation of x; cr: connect request; cc: connect confirm; d: data request

Process integration

After performing a number of process decomposition steps in succession, we may obtain a specification with several functional layers, interacting through a number of internal gates. This is depicted in Figure 7, where in a layered protocol architecture (with interaction points for the execution of service primitives) each protocol entity is decomposed into an upper and lower part. These two parts interact through an internal gate where the events correspond to PDUs. At some point on the design trajectory we may now decide to integrate an upper part of layer *N* with a lower part of layer *N* + 1, thus eliminating the actual implementation of the *service access point* of layer *N*. In this way we have defined how the PDUs at layer *N* + 1 are related to the PDUs at layer *N*, and *vice versa*. This step is termed *process integration*, and may lead to more efficient implementations.

Process integration is enabled by the definition of the *hiding* operator in LOTOS. However, the single process obtained after joining two processes by hiding the gate through which they communicate may now exhibit non-deterministic behaviour. We will not elaborate on this here, but confine ourselves to two remarks: under certain conditions, the integration of two deterministic processes will yield a single deterministic process; when transforming towards the final implementation, we may select a deterministic behaviour out of the non-deterministic behaviour of the specification and still obtain a *testing equivalent* system[15].

CONCLUSION

In this paper we have presented our view on how performance requirements can be considered in the

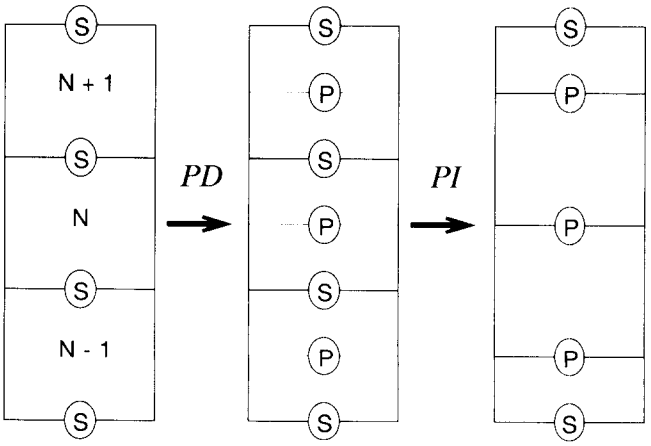

Figure 7. Process decomposition (PD) and integration (PI)

design of distributed and concurrent systems using LOTOS. Furthermore, the design trajectory to be traversed in this approach has been discussed in more detail in terms of specific transformation steps that are relevant to the fulfillment of non-LOTOS requirements. In summary, performance requirements can be addressed in the following ways:

1. In the requirements capturing process we should represent the parallelism present in user requirements in the first LOTOS specification. This is supported by the constraint-oriented specification style.
2. During the early transformation steps process decomposition is performed that has to be guided by implementation knowledge. The definition of abstract implementation elements like TFs, CFs and SFs supports this method.
3. Selection among different process structures can be performed by assessment of their corresponding queueing networks. The latter can be carried out analytically, or by simulation.
4. During the later transformation steps, interaction point decomposition and event refinement and integration are applied for the implementation of abstract interfaces defined in the specification. Here the selection of efficient interface mechanisms will affect the performance obtained in the final implementation. Finally, process integration can also be used to limit the number of real interfaces in the real system.

The usefulness of this approach has been demonstrated in a design example, partly carried out in an industrial environment in the context of two research projects.

ACKNOWLEDGEMENTS

This work has been partly supported by the CEC under the contracts of two research projects the author was involved with: project # 890, PANGLOSS[3], and project # 2304, LOTOSPHERE[7]. In the PANGLOSS project, a method for the application of LOTOS in the design of parallel systems has been developed, and applied to the design of a gateway for the interconnection of heterogeneous networks. The method turned out to be useful in a shared industrial-academic environment[4]. In the ongoing LOTOSPHERE project a further refinement of the PANGLOSS method is carried out, and applied to the development of communication systems in industry. It shows that the practical employment of LOTOS pays off, and expectations about its potential are confirmed. Last but not least, I would like to thank Marten van Sinderen and Giuseppe Scollo for their comments and suggestions on an early version of this paper.

REFERENCES

1 **Brinksma, H (ed.)** *ISO - Information Processing Systems - Open Systems Interconnection: LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour,* ISO International Standard 8807, ISO, Geneva, Switzerland (1988)
2 **van Eijk, P H J** *Software Tools for the Specification Language LOTOS,* PhD Thesis, Tele-Informatics Group, University of Twente, Enschede, Netherlands (1988)
3 **Bogaards, K, Pires, L, Pras, A and Schot, J** 'The Pangloss Method', *Proc. ESPRIT Conf.,* Elsevier, Brussels (1988)
4 **Schot, J and Pires, L (eds)** *PANGLOSS Architecture Task Final Deliverable — Results of the ESPRIT/ PANGLOSS Project,* University of Twente, Enschede, Netherlands (1989)
5 **Schot, J** 'Systematic high level design of a switching system for stream oriented traffic', *Proc. 7th Int. Conf. on Softw. Eng. for Telecommun. Switching Syst.,* Bournemouth, UK (1989)
6 **Schot, J** 'Systematic design of telecommunication systems using formal description techniques', *Proc. ESPRIT Conf.,* Elsevier, Amsterdam (1989)
7 **van de Lagemaat, J and Vissers, C A** 'Formal description techniques for distributed computing systems, the challenges for the 1990s', *Proc. IEEE Workshop on Future Trends of Distributed Comput. Syst.,* IEEE Press, USA (1990)
8 **Bogaards, K** 'LOTOS supported system development', *Proc. 1st Int. Conf. on Formal Description Techniques (FORTE),* North-Holland, Amsterdam (1988)
9 **Conti, G** *Methodologie d'Implementation des Protocoles de Communication,* PhD Thesis (No 842), Ecole Polytechnique Fédérale de Lausanne, Switzerland (1990)
10 **Langerak, R** 'Decomposition of functionality: A correctness preserving LOTOS transformation', *Proc. 10th IFIP Conf. on Protocol Specifications, Testing & Verification,* North-Holland, Amsterdam (1990)
11 **Parrow, J** 'Submodule construction as equation solving in CCS', *Theor. Comput. Sci.,* Vol 68 (1989) pp 175–202
12 **Vissers, C A, Scollo, G, van Sinderen, M and Brinksma, H** 'On the use of specification styles in the design of distributed systems', *Proc. Advanced Seminar on Foundations of Innovative Software Development (TAPSOFT),* Barcelona, Spain (1989)
13 **Abramsky, S** 'Observational equivalence as a testing equivalence', *Theor. Comput. Sci.,* Vol 53 (1987) pp 225–241
14 **Brinksma, H, Scollo, G and Steenbergen, C** 'LOTOS specifications, their implementations and their tests', *Proc. Conf. on Protocol Specification, Testing & Verification VI,* North-Holland, Amsterdam (1987) pp 349–360
15 **De Nicola, R and Hennessy, M C B** 'Testing equivalences for processes', *Theory of Computer Science 34,* Springer-Verlag, Berlin (1984) pp 83–133
16 **Baskett, F, Chandy, K M, Muntz, R R and Palacios, F G** 'Open, closed and mixed networks of queues with different classes of customers', *J. ACM,* Vol 22 No 2 (1975) pp 248–260
17 *QNAP2 User and Reference Manuals,* Simulog S.A., Paris, France (1989)