Generic Layout of Communication Subsystems - *A Case Study*

R. Gotzhein, B. Geppert, C. Peper, F. Rößler

SFB 501 Report 14/96

Generic Layout of Communication Subsystems
- A Case Study

R. Gotzhein, B. Geppert, C. Peper, F. Rößler
{gotzhein, geppert, peper, roessler} @ informatik.uni-kl.de

Report 14/96

Sonderforschungsbereich 501

Computer Networks Group
Computer Science Department
University of Kaiserslautern
Postfach 3049
67653 Kaiserslautern
Germany

Generic Layout of Communication Subsystems
- A Case Study§

R. Gotzhein, B. Geppert, C. Peper, F. Rößler

Computer Science Department, University of Kaiserslautern
Postfach 3049, 67653 Kaiserslautern, Germany
{gotzhein, geppert, peper, roessler} @ informatik.uni-kl.de

Abstract

The purpose of this exposé is to explain the generic design of a customized communication subsystem. The exposé addresses both functional and non-functional aspects. Starting point is a real-time requirement from the application area building automation*. We show how this application requirement and some background information about the application area lead to a system architecture, a communication service, a protocol architecture and to the selection, adaptation, and composition of protocol functionalities. The reader will probably be surprised how much effort is necessary in order to implement the innocuous, innocent, inconspicuous looking application requirement. Formal description techniques (FDTs) will be used in all design phases.*

1. Introduction

A central goal of the Sonderforschungsbereich (SFB) 501 is to devise methods and techniques for the generic development of large software systems [SFB94]. Today, large systems typically are concurrent and distributed. Therefore, communication systems are usually an integral part of large systems, they form the basis for applications and operating systems. Due to the large variety of applications and technologies, the requirements on communication systems are manifolded. It is therefore expected that they will not be satisfiable by a small number of general-purpose protocol stacks. Rather, customization of communication systems will play a major role.

To overcome the additional development effort resulting from customization, and in accordance with the goals of the SFB, we propose to use the concept of genericity during all development phases. In the following, we will illustrate some of our ideas for the generic layout and the customization of communication subsystems using a single innocuous-looking application requirement as starting point.

---

§ This work has been supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the Sonderforschungsbereich (SFB) 501 "Development of Large Systems with Generic Methods"

To start with, a few central notions will be briefly addressed. In Figure 1, two different computer-communications architectures are shown. Both are layered architectures. The first one comprises seven layers, as it is, for instance, the case for the OSI Basic Reference Model [ISO84] or IBMs Systems Network Architecture. Also, the Department of Defense (DoD) architecture [CeCa83] now used in the Internet falls into this category. The relatively large number of layers and the more or less fixed set of general-purpose protocols has turned out to be inappropriate for many applications. Here, the second architecture, consisting of *basic technology*, *communication subsystem*, and *application layer*, together with special-purpose, customized protocols, is more suitable.

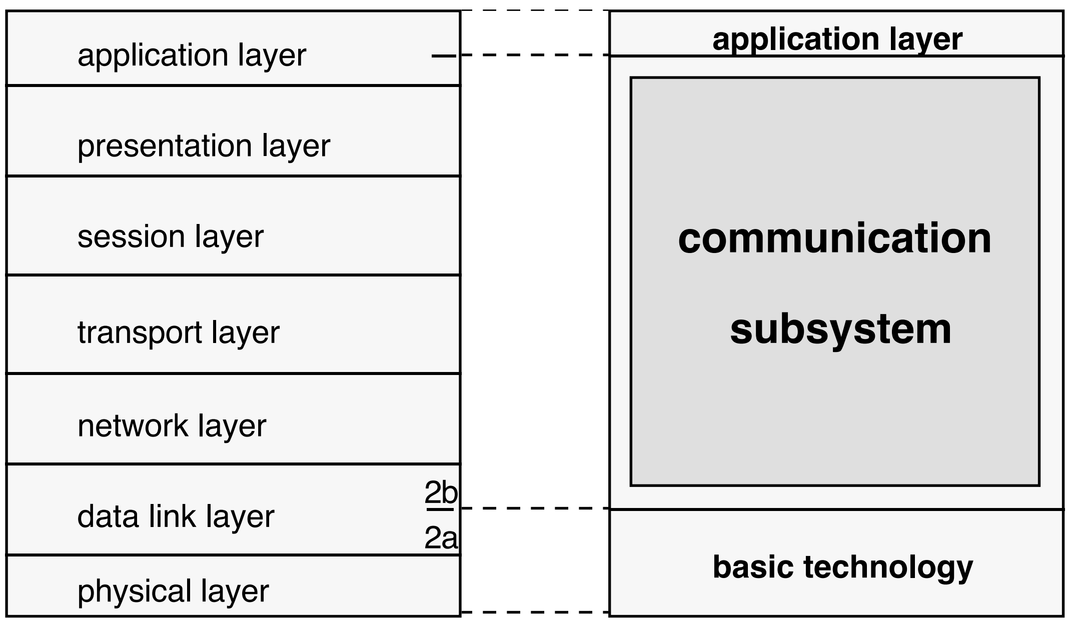

Figure 1: Computer-communications architectures

Basic technology is referring to components of a communication system that are typically manufactured in hardware (including microcode), such as Ethernet, ATM, and Token Ring. Basic technology may be application-neutral - in a sense that its characteristics does in principle allow all kinds of applications to be based on this technology -, or application-specific. The functionality covered by the basic technology is not uniquely determined, it may or may not, for instance, comprise routing mechanisms.

The adaptation between the basic technology and the application is established by a *communication subsystem*, which together with the basic technology forms the entire communication system. It is this communication subsystem and its generic layout on which the case study presented in the following will focus.

Genericity is another central notion. We may distinguish between genericity of products and of the development process. Genericity of products is supported by concepts such as compositionality, adaptation, parameterization, and reusability[1]. Genericity of the

[1] These concepts are not orthogonal.

development process is additionally supported by concepts such as synthesis and generation. We will refer to these concepts in the following in order to point out potential of using genericity for developing customized communication subsystems.

The presentation of the case study is structured as follows. In Section 2, the application requirement is presented. Early design decisions then lead to a refined system architecture (Section 3) and a communication service (Section 4). This communication service is proven adequate w.r.t. the application requirement (Section 5). Next, a communication protocol providing this service is configured (Section 6). Finally, implementation issues will be briefly addressed (Section 7). We conclude with some remarks.

2. Application requirement

Starting point is a real-time requirement[2] from the application area *building automation*, which has been worked out by the customer embodied by project D1 in cooperation with the contractor embodied by Team 1 and Team 2. The requirement is the following:

> *"Each time hazardousCondition holds continuously for at least T time units, the upper sash must be closed within this time span and must then remain closed as long as hazardousCondition holds."*

Formally[3]: $\square\,(\mathit{hazardousCondition} \Rightarrow_{\leq T} \mathit{upperSashClosed})$ (ar$_1$)

This allows to leave the windows open if *hazardousCondition* holds for less than T time units; however, the windows may be closed at any time, if *hazardousCondition* holds.

We have used a real-time temporal logic with customized operators (in the example, we have introduced a new operator called "bounded response and continuity operator" [KrGoPe96]). This has allowed us to be concise and to formally capture certain non-functional aspects.

hazardousCondition is an auxiliary predicate with the following important property:

$$\square\,([\mathit{hazardousCondition}] \rightarrow \square_{\leq T3}\ \mathit{hazardousCondition}) \qquad (\mathrm{aa}_1)$$

Informally, once *hazardousCondition* becomes true, it remains true for at least *T3* time units before becoming false again.

The auxiliary predicate *hazardousCondition* has been defined in terms of a primitive predicate *hazardousSituation* in [PeGoKr96, GoKrPe96]. Examples of a hazardous situation (not to be confused with hazardous condition!) are: heavy winds, heavy rainfall, danger of burglary (e.g. during darkness). While hazardous situations can occur at all moments in time, and can exist for an arbitrary time span, the predicate *hazardousCondition* should not change its value too frequently. Otherwise, this may lead to a frequent opening and closing of windows. By choosing *T3* suitably, frequent window actions can be avoided.

---

2 For a more comprehensive treatment, see [PeGoKr96].

3 The logic used in this section is called tTL (tailored real-time temporal logic) and introduced in [KrGoPe96]. The customized operator in the formula can be defined syntactically: $p \Rightarrow_{\leq T} q =_{\mathrm{Df}} \Diamond_{\leq T}\,(p\ \mathcal{W} \neg q)$.

While working on the application requirements in the area of building automation, we have found that most of them obey a small number of syntactic *patterns*, like the requirement above or the definition of *hazardousCondition* (see [PeGoKr96, GoKrPe96]). This has led us to the hypothesis that by developing a customized communication subsystem for each such pattern, we will be able to *reuse* the development process as well as its products with some adaptations. As a result, both development process and products will be highly *generic*. Note that the patterns may be different in other application areas.

3. System architecture

Based on a set of requirements supplied by the customer, Team 2 has developed a system architecture as part of the system design. The architecture shown in Figure 2 is extracted from that system architecture and contains only those components relevant to the application requirement above, which are:

- A control cell *ccSafety*. One of the tasks of *ccSafety* is to realize the above requirement by appropriate interaction with other components. Interaction could, for instance, occur by message passing, shared variables, or procedure call.
- A (logical) sensor *senHazCon* perceiving the predicate *hazardousCondition*. There may be several physical sensors and also some processing involved in determining the current value of *hazardousCondition*. This is a matter of implementation and not considered at this stage. It is, however, important how much time may pass from *hazardousCondition* being satisfied until this is recorded by sensors, processed and ready for being communicated via *ipHazCon*. This time T_{sen1} is part of *T*.
- A (logical) actuator *actUSash* controlling the upper sash. There may be several physical actuators involved in closing the upper sashes. As before, this is an implementation matter. Again, some time will pass from the reception of a signal to close the upper sashes via *ipUSash* until they are finally closed. This time T_{act1} is part of *T*.
- Components may interact via common interaction points [Got90] *ipHazCon* and *ipUSash*. Thus, the ability of components to interact is architecturally modeled.

In order to realize the requirement, suitable cooperation of components is necessary. This cooperation has to be such that firstly, the specified relative order of events is realized (a functional requirement), and secondly, the specified timing constraints are observed (a non-functional requirement). If the times T_{sen1} and T_{act1} are determined, we have $T - T_{sen1} - T_{act1}$ time units from the recording of a hazardous condition until the order to close all upper sashes must be given.

In general, $T - T_{sen1} - T_{act1}$ may be shared among the components in an arbitrary way. In this exposé, we will make the assumption that each component has a fixed share, and that the sum of these shares be equal to or less than the available time span. This assumption reduces flexibility, but allows to consider each component independently.

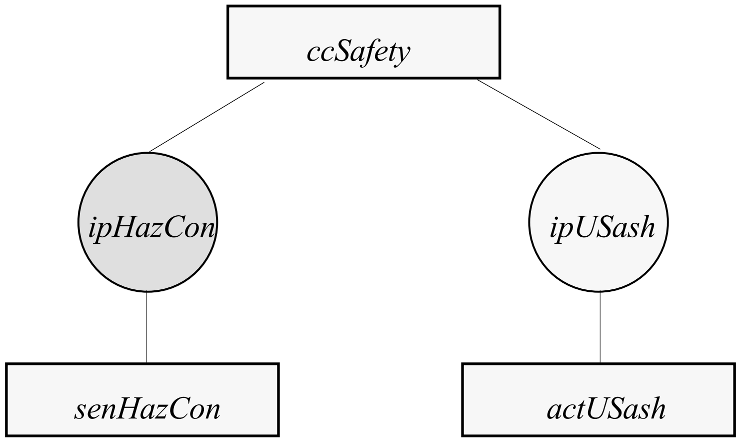

Figure 2: Extract of the conceptual system architecture

Note that the architecture in Figure 2 still allows for a variety of design decisions. For instance, in a centralized solution, interaction between components could be by procedure call, interrupt or shared variables. In a decentralized solution, interaction may occur, for instance, through message passing. The same or a different communication system may be introduced for each of the interaction points. The sensor and actuator components may be refined into smaller components including physical sensors and actuators, which communicate via some shared medium. This medium may also be shared among the components shown in Figure 2, or be a separate medium. These are design decisions which are by no means predetermined at this stage of development. What the architecture shows is a conceptual design that may be implemented in many different ways.

4. Interaction properties

In the following, we will focus on the interaction points *ipHazCon* and *ipUSash*, which determine the kind of interaction between the agents *ccSafety*, *senHazCon* and *actUSash*. *ipHazCon* and *ipUSash* will be customized according to the needs of the application. Therefore, interaction point properties have to be defined first. To formalize these properties, we use a first-order real-time temporal logic. This logic is explained in [Got93] and [KrGoPe96], respectively.

The component *ipHazCon* is the conceptual location where interactions between *ccSafety* and *senHazCon* occur.

ipH_1 $\square\, \forall v,ts.\ ([\mathit{ccSafety.?\ value.resp(v,ts)}]$

$\supset \mathit{now}\text{-}\mathit{ts} \leq T_{ipH1} \wedge \blacklozenge\ [\mathit{senHazCon.!\ value.resp(v,ts)}])$

If *ccSafety* accepts (receives, reads) the value *v* with timestamp *ts*, where *v* is assumed to be the value of *hazardousCondition* at time *ts*, *v* must satisfy two conditions:

- At the time of acceptance, v is not older than $T_{sen1} + T_{sen2} + T_{ipH1}$. This relies on the assumption (specified as a local property of the agent *senHazCon*, see Section 5) that the timestamp *ts* is correctly set.
- v has been offered (sent, written) by *senHazCon* before. In other words, values must not be created by the interaction point.

ipH_2 $\Box$ (*at ccSafety.? value.resp* $\supset \Diamond_{\leq T_{ipH2}}$ [*ccSafety.? value.resp*])

If *ccSafety* is ready to accept (receive, read) the value of *hazardousCondition*, it will accept (receive, read) some value within T_{ipH2}.

Properties ipH_1 and ipH_2 taken together restrict interactions occurring between *ccSafety* and *senHazCon* without determining a particular interaction paradigm. For instance, shared variables, interrupts, procedure calls or message passing may be used. If shared variables are used, then *! value.resp* corresponds to writing, *? value.resp* to reading. If interaction is by procedure call, then *value.req* (not constrained by ipH_1 and ipH_2) corresponds to the procedure call, *value.resp* to the return. Another option for implementation is the rate of value acceptance of *ccSafety*, which may be different from the rate of value offering of *senHazCon*.

At this point, we make the design decision to use message passing for interaction between *ccSafety* and *senHazCon*, which is in line with the goal of developing large systems, as message passing is a frequent interaction paradigm in distributed systems. As a consequence, *!* corresponds to sending, *?* to receiving a message. With this design decision, we can understand ipH_1 and ipH_2 as the specification of a communication service. This service is the starting point for the customization of a communication protocol.

The component *ipUSash* is the conceptual location where interactions between *ccSafety* and *actUSash* occur.

ipU_1 $\Box$ ∀s. (#[*ccSafety.! set.req(s)*] ≥ #[*actUSash.? set.req(s)*])

The number of set requests with parameter value *s* passed to *actUSash* does not exceed the number of set requests by *ccSafety* with the same parameter value.

ipU_2 $\Box$ (*at ccSafety.! set.req* $\supset \Diamond_{\leq T_{ipU1}}$ [*ccSafety.! set.req*])

When *ccSafety* is ready to request the closing of the upper sashes at *ipUSash*, the request occurs within T_{ipU1}.

ipU_3 $\Box$ (*at actUSash.? set.req* ∧ #[*ccSafety.! set.req*] > #[*actUSash.? set.req*]
$\supset \Diamond_{\leq T_{ipU2}}$ [*actUSash.? set.req*])

When *actUSash* is ready to accept a request to close the upper sashes at *ipUSash*, and there is a pending request, the request is accepted within T_{ipU2}.

Similar to *ipHazCon*, the properties of *ipUSash* leave many options for implementation.

We expect that for application requirements following the same syntactic *pattern*, we may define similar communication services. Thus, we will be able to *reuse* the development process as well as its products with small variations. As a result, both development process and products will be highly *generic*.

Furthermore, we expect that similar communication services can be integrated to form a single service, which could - in a broad sense - be understood as the *multiplexing of application requirements* over a single communication system.

5. On the proof of the application requirement

So far, we cannot be certain whether it is possible to customize a communication system that satisfies the specification of *ipHazCon* and *ipUSash*. With the interaction paradigm being determined, we still have to make certain assumptions concerning the behaviour of the agents *ccSafety* and *senHazCon*. This is, however, typical in the context of open systems and follows the rely/guarantee pattern. One such assumption, for instance, is that if *senHazCon* offers a value *v* at *ipHazCon*, *v* is a value of *hazardousCondition* that has a determined maximum age and is associated with a correct timestamp *ts*. Assumptions sufficient for proving the application requirement are listed below.

The component *senHazCon* processes data probed from the physical environment in order to evaluate the environment predicate *hazardousCondition*. Since this evaluation may take some time, we introduce a predicate *sSenHazCon* associated with *senHazCon*, which represents the current status of processing. This predicate can be understood as an auxiliary predicate to make the specification more readable. There is no need to represent it in an implementation.

sen_1 $\square\ (\mathit{hazardousCondition} \Rightarrow_{\leq T_{sen1}} \mathit{sSenHazCon})$

Each time *hazardousCondition* holds continuously for at least T_{sen1}, *sSenHazCon* must become true within that time span and must remain true as long as *hazardousCondition* holds[4] .

sen_2 $\square \Diamond_{\leq T_{sen2}} [\mathit{at\ senHazCon.!\ value.resp}]$

senHazCon offers value responses with a minimum rate. Note the use of the action operator, which has the effect that the parameters of the value response do not age beyond T_{sen2}.

sen_3 $\square\ \forall v,ts.\ ([\mathit{at\ senHazCon.!\ value.resp(v,ts)}] \supset v = \mathit{sSenHazCon} \wedge ts = \mathit{now})$

A value response is parameterized with the current value of *sSenHazCon* and the current time.

The component *ccSafety* is responsible for accepting (processed) data about the physical environment from *senHazCon*, and for requesting appropriate action by *actUSash*. To make the specification (and the proof) more readable, an auxiliary predicate *sCcHazCon* is associated with *ccSafety*. The following constraints apply:

---

[4] If $T_{sen1} \leq T3$ (see Section 2), the requirement simplifies to: each time *hazardousCondition* holds, *sSenHazCon* must become true within T_{sen1} and must remain true as long as *hazardousCondition*.

cc_1 $\Box \Diamond_{\le T_{cc1}}$ *at ccSafety.ipHazCon.? value.resp*

ccSafety must be ready to accept the value of *sSenHazCon* frequently enough.

cc_2 $\Box\, \forall ts.\ ([sCcHazCon] \supset \Diamond_{\le T_{cc2}}\ at\ ccSafety.ipUSash.!\ set.req(close))$

If a hazardous condition is reported, *ccSafety* is ready to request the closing of the upper sashes within T_{cc2}.

cc_3 $\Box\, \forall v,ts.\ ([ccSafety.ipHazCon.?\ value.resp(v,ts)]$
$\supset (sCcHazCon = v\ \mathcal{W}\ \exists v',ts'.\ ([ccSafety.ipHazCon.?\ value.resp(v',ts')] \wedge v \neq v')))$

sCcHazCon always equals the last reported value of *sSenHazCon*.

cc_4 $\Box\, (sCcHazCon \supset \neg\ [at\ ccSafety.ipUSash.!\ set.req(open)])$

If the last reported value of *sSenHazCon* has been true, then no request to open the upper sashes must be issued.

The component *actUSash* is responsible for opening and closing the upper sashes. To make the specification (and the proof) more readable, an auxiliary predicate *sActUSash* is associated with *actUSash*. The constraints are:

act_1 $\Box\, \forall v.\ ([actUSash.ipUSash.?\ set.req(v)]$
$\supset (sActUSash = v\ \mathcal{W}\ \exists v'.\ ([actUSash.ipUSash.?\ set.req(v')] \wedge v \neq v')))$

sActUSash always equals the value of the last set request.

act_2 $\Box\, ((sActUSash = close \Rightarrow_{\le T_{act1}} upperSashClosed) \wedge$
$(sActUSash = open \Rightarrow_{\le T_{act1}} \neg\ upperSashClosed))$

Each time *(sActUSash = close)* holds continuously for at least T_{act1}, *upperSashClosed* must become true within that time span and must remain true as long as *(sActUSash = close)* holds (analogously in case *(sActUSash = close)* does not hold).

act_3 $\Box \Diamond_{\le T_{act2}}$ *at actUSash.? set.req*

actUSash offers to accept set requests with a minimum rate.

From the properties stated in Sections 4 and 5, we have formally proven the application requirement. More precisely, we have proven the following:

$$\models_i (sen_1 \wedge sen_2 \wedge sen_3 \wedge ipH_1 \wedge ipH_2 \wedge cc_1 \wedge cc_2 \wedge cc_3 \wedge cc_4 \wedge ipU_1 \wedge ipU_2 \wedge ipU_3 \wedge act_1 \wedge act_2 \wedge act_3 \wedge aa_1) \supset ar_1$$

The existence of this proof provides evidence that the communication service is indeed suitable and sufficient.

6. Protocol specification

In this section, we will explain how a customized communication subsystem for the service as specified by ipH_1 and ipH_2 is configured. Before the configuring can be done, a pool consisting of elements called *protocol building blocks* has to be created. Based on the required service, a subset of these building blocks will be selected, adapted and composed to yield the customized communication subsystem. The result of this configuration process is listed in the appendix.

We specify protocol building blocks and the configured subsystem using SDL-92, which is an internationally standardized FDT [ITU92, Ols94]. Unlike most other FDTs, SDL is widely used in industry, and well-supported by tools. The language contains a set of graphical constructs, which improves readability of specifications and thus the acceptance of the language itself. Unlike the temporal logic used so far, SDL does not support the specification of non-functional aspects. However, being an operational technique, SDL is more appropriate for specifying the mechanisms occurring in communication protocols as compared to temporal logic.

Figure 3: Service architecture: *SystemControl1*

Figure 4: Protocol architecture: *SystemControl3*

Figure 3 shows an SDL-specification of an open system with two *blocks* named *ipHazCon* and *ipUSash*. These blocks are linked to the environment by so-called *channels*. The relationship to the conceptual architecture of Figure 2 is obvious: the blocks represent the interaction points named *ipHazCon* and *ipUSash*. *ccSafety*, *senHazCon*, and *actUSash* are not part of the communication system and therefore not included in the SDL-specification.

It is straightforward to refine this structure into the usual protocol architecture. However, this would result in separate components representing the basic technology, which is not the way we want to realize the communication services represented by *ipHazCon* and *ipUSash*. Instead, we want to place the communication subsystems related to these services on a single component representing the basic technology. As a consequence, the structure obtained after the refinement of the service architecture needs some modification.

We have devised a small number of transformation rules (not stated in this exposé) stating how the internal system structure may be modified without affecting the external behaviour of the open system. Suitable application of these transformation rules leads to the protocol architecture shown in Figure 4. This architecture consists of a single component representing the basic technology, and four components that together form the communication subsystem to be configured. Please note that the external appearance of this system is identical to the system in Figure 3.

The following examples refer to the block *SensorPart* (see Figure 4).

Figure 5: Protocol building block: *Service Type HandleData*

Figure 6: Protocol entity: *Process Type ReliableTransport_S*

SDL supports the specification of protocol building blocks through self-contained units called *services* [Ols94]. Note that these so-called services are not related to communication services as defined earlier, rather they define transition systems. A simple service (or protocol building block), consisting of an automaton with a single transition, is shown in Figure 5. In this transition, an input signal is accepted, and a PDU is partially assembled.

Apart from the fact that non-functional behaviour can not be expressed in SDL, there are other limitations needing further attention. From a semantical point of view, it should be noted that services in SDL are incomplete specifications and therefore only become formally meaningful when being part of a complete specification. Another important limitation is that with SDL, it cannot be expressed how and when protocol building blocks may be composed. This "composition intelligence" has therefore to be specified using some meta-language.

Given a pool of protocol building blocks, a protocol architecture, and a service specification, suitable protocol building blocks that together form the customized protocol have to be selected. This certainly is a highly creative step of the design process which can be guided by heuristics and other means.

Building blocks may still be generic in the sense that service-specific information needs to be added, leading to an adaptation of building blocks. This can, for instance, be supported in SDL by suitable parameterization, or by redefinition of transitions and types.

Once the protocol building blocks are selected and adapted, they have to be composed to yield a communication subsystem consisting of protocol entities. In SDL, a protocol entity can be represented by a process, which is configured by composing services. An example is shown in Figure 6, which includes the protocol building block *HandleData* shown earlier. A process of that type is then contained in *SensorPart* (see Figure 4).

The entire process of configuring a communication subsystem, consisting of the selection, adaptation and composition of protocol building blocks, provides for a high degree of *reusability* and *genericity*. Protocol building blocks may be combined in many ways, leading to customized communication subsystems. We have shown how certain language features of SDL-92 may be used to support this process. Additional language features such as inheritance, redefinition, and procedures offer further potential for a generic layout of communication subsystems.

7. Implementation issues

With the protocol subsystem being specified in SDL, it is in principle possible to use one of the commercially available SDL-compilers to automatically or semi-automatically *generate* executable protocol code. There exists, however, only little experience with generating code for real-time protocols. As SDL-92 does not support the specification of timing constraints[6], it

[6] SDL-92 supports the concept of timers. This, however, is done on a syntactical basis only and has no semantical foundation.

cannot be expected that existing commercial SDL-compilers can be used here. This is an issue for further study and can only be solved in the long run.

Another implementation issue is optimization. There exists potential for integrating communication subsystems of a single node into one subsystem. This integration may be performed in a reusable way on the specification level such that the integrated subsystem is again specified in SDL. In Figure 4, for instance, the blocks *ControlPart1* and *ControlPart2* may be integrated into a single block, and processes be fused.

The implementation of the communication subsystem interfaces with the underlying basic technology. We have already built an experimental network based on a token ring protocol and have developed a driver capable of certain deterministic timing guarantees. Future plans foresee the replacement of this basic technology by an ATM network. Due to the generic layout of the communication subsystem, the adaptation effort caused by this replacement is expected to be small.

8. Concluding remarks

We have identified potential for the generic layout of communication subsystems, illustrated by a case study. This potential can be classified into: genericity of products and of the development process.

- In building automation, we have so far encountered a relatively small number of syntactic patterns of application requirements. This may lead to reuse through adaptation. The syntactic patterns may, of course, be different in other application areas.
- Application requirements following the same syntactic pattern may be supported by similar communication services. Again this may lead to reuse through adapation.
- Also, we have addressed the configuring of customized communication subsystems, which leads to reusability based on a predefined set of protocol building blocks, their adaptation, and composition.
- Genericity of the development process has been identified in all stages of protocol development. We have argued that in the early design stages and for the given application domain, products follow a relatively small number of syntactic patterns. Since these products are therefore similar, the design processes will be largely reusable. With respect to the protocol development, strategies and heuristics to support the configuring still have to be investigated. We are only just beginning to understand the systematics underlying this process.
- As far as the derivation of executable code is concerned, semi-automatic code generation from the operational formal specification seems to be promising.

It should be pointed out that broad use of FDTs has been made in all stages of development. Also, non-functional aspects have been formally treated as illustrated by a case study. In summary, there is large potential for generic methods and techniques, which we have only just started to exploit.

References

[CeCa83] Cerf, V., Cain, E.: The DoD Internet Architecture Model, Computer Networks, Vol. 7, 1983

[Got90] Gotzhein, R.: The Formal Definition of the Architectural Concept "Interaction Point", in: S. T. Vuong (ed.), Formal Description Techiques, II, North-Holland, 1990, pp. 67-81.

[Got93] Gotzhein, R.: Open Distributed Systems - On Concepts, Methods and Design from a Logical Point of View, Vieweg Wiesbaden, 1993

[GoKrPe96] R. Gotzhein, M. Kronenburg, and C. Peper: Specifying and Reasoning about Generic Real-Time Requirements - A Case Study, SFB 501 Report 15/96, University of Kaiserslautern, Germany, 1996

[ISO84] ISO/CCITT: Information Processing Systems - Open Systems Interconnection - Basic Reference Model, ISO 7498/CCITT Recommendation X.200, 1984

[ITU92] ITU, Geneva: Specification and Description Language (SDL), 1994

[KrGoPe96] M. Kronenburg, R. Gotzhein, and C. Peper: A Tailored Real-Time Temporal Logic for Specifying Requirements of Building Automation Systems, SFB 501 Report 16/96, University of Kaiserslautern, Germany, 1996

[Ols94] Olsen, A., et al.: Systems Engineering Using SDL-92, North-Holland, 1994

[PeGoKr96] C. Peper, R. Gotzhein, and M. Kronenburg: A Generic Approach to the Formal Specification of Real-Time Requirements of Building Automation Systems, SFB 501 Report 1/97, University of Kaiserslautern, Germany, 1997

[SFB94] Sonderforschungsbereich 1496, Entwicklung großer Systeme mit generischen Methoden, Finanzierungsantrag, University of Kaiserslautern, 1994

Appendix