Micro Protocol Design: The SNMP Case Study [1]

Reinhard Gotzhein[a], Ferhat Khendek[b], Philipp Schaible[a]

[a]Computer Science Department, University of Kaiserslautern
Postfach 3049, D-67653 Kaiserslautern, Germany
{gotzhein,schaible}@informatik.uni-kl.de

[b]ECE Department, Concordia University,
1455, de Maisonneuve W., Montreal (P.Q.), Canada H3G 1M8
khendek@ece.concordia.ca

Abstract. Today, reuse in software engineering is usually supported by component libraries, such as Java packages. Components are self-contained, ready-to-use building blocks, which are selected and composed. They are usually associated with the implementation phase, a result of practical experience rather than of existing limitations. In this paper, we shed some new light on the concept of components from the protocol engineering point of view. In particular, we describe a conceptual framework for the protocol design phase, and introduce a specific type of protocol design components called *micro protocols*. We then instantiate and apply this framework to a subset of SNMP, the Simple Network Management Protocol, using SDL as design language.

Keywords: protocol engineering, reuse, components, micro protocols, formal methods, SDL

1. Introduction

Reuse of solutions and experience for recurring system development problems plays a key role for quality improvement and an increase in productivity. As a prerequisite, the problems and their solutions have to be in some sense "similar". These similarities should not be understood as purely syntactical, rather, semantical and conceptual similarities should be considered as well, which requires precise domain knowledge and some creativity. Reuse has been studied thoroughly in software engineering, which has led to the distinction of three main reuse concepts [5], namely components, frameworks, and patterns. It should be emphasized that these reuse concepts can be applied together, for instance, by defining a component framework [11] such as CORBA or DCOM+ and adding components, or by using patterns to define components used in a component framework. In this paper, we are concerned with components.

Components are self-contained, ready-to-use building blocks, which are selected from a component library and composed. As a component is usually defined by a piece of code, this reuse concept is strongly implementation-oriented. Being a syntactically complete code fragment, a component has to be used as it comes. Therefore, *design for independence* [11], for instance, by suitable parameterization, is an important issue here, as well as the definition of suitable "glue" to compose components.

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

The focus of our current work is on reuse in *protocol engineering*. It is a matter of fact that reuse in this otherwise well-understood domain has not been very successful in the past. Still, protocol design usually starts from scratch. In [1] and [2], we have shown how the design pattern idea can be applied to communication protocols. More specifically, we have introduced the notion of *SDL pattern*, which combines the advantages of the pattern idea and the use of a formal design language, we have developed a pattern pool and a pattern-oriented design process, and have applied our approach to the design and re-design of several real-life communication protocols.

In this paper, we describe and apply the concept of *protocol components*, i.e., self-contained, ready-to-use building blocks for communication systems development. Earlier results on protocol components have been reported in [8] (F-CCS), [6] (Da Capo) and [12] (HORUS). All these approaches are on *code* reuse. The early development phases including protocol design are not supported. In HORUS, the components are functions. Some of these functions are self-contained, but others are not self-contained and are always composed with other components. In the following, we describe a conceptual framework for the protocol design phase, and define a specific type of protocol component called *micro protocol* (Section 2). In Section 3, we survey the support of SDL-2000 for the definition and composition of micro protocols. We then apply our framework to a subset of SNMP, the Simple Network Management Protocol (Section 4), and draw some conclusions (Section 5).

2. The concept of ‘Micro Protocol’

In this section, we introduce a specific type of protocol (design) component called *micro protocol*, and a corresponding communication system architecture. To start with, we revisit the notion of *communication protocol*, which then is refined into *protocol components*. We draw a clear distinction between a *concept* (such as *service* or *protocol*) and its *specification*.

- A *communication system* is a system providing a meaningful, self-contained communication service to a set of users, using a suitable communication protocol. Depending on the amount of visible detail, we distinguish between service level and protocol level.
- A *communication service* consists of a service architecture, service users constraints, service provider guarantees and service formats for the exchange of data from the service users’ point of view.
- A *communication protocol* consists of a protocol architecture, a set of protocol rules, and protocol formats for the exchange of data from the service provider’s point of view:
    - A *protocol architecture* defines the agents (service users, protocol entities, lower service provider) as well as their common interaction points. The protocol architecture refines the service architecture.
    - *Protocol rules* capture the legal behavior of protocol entities in a self-contained way, i.e., without relying on any assumptions about the environment, consisting of the service users, the lower service provider and the other protocol entities.
    - *Protocol formats* define the set of values that may be exchanged between protocol entities and their local environment, consisting of service users and lower service provider.

From this discussion, it follows that a communication protocol can be defined independently of the lower communication service. However, the service provided by the set of protocol entities depends on the lower communication service as well as on the service constraints.

Communication systems can be structured into smaller constituents in different ways. A *communication component* is a self-contained, ready-to-use building block of a communication system. Various kinds of communication components can be distinguished, depending on the point of view of the communication systems engineer:

- A *protocol functionality* is a single aspect of internal protocol entity behavior (*operational structuring*), e.g., flow control, error control. It is realized by a particular *protocol mechanism* (e.g., sliding-window, rate control, checksum), and generally distributed among a set of protocol entities. We demand that there be no further decomposition into smaller functionalities.
- A *protocol collaboration* is a self-contained subset of synchronization and causality relationships of a set of protocol entities.
- A *protocol phase* is a stage of protocol execution (*temporal structuring*), e.g. connection setup, data transfer, connection release.
- A *micro protocol* is a communication protocol with one protocol functionality and the required protocol collaboration. Because a protocol functionality covers only one single aspect of protocol behavior, a micro protocol is not decomposable.
- A *macro protocol* is the composition of micro and/or macro protocols.
- A *(micro/macro) protocol entity* is an agent following a (micro/macro) protocol.
- A *protocol layer* is a complete set of protocol entities following the same protocol.

The composition of communication components may require specific kinds of "glue". For instance, collaborations can be composed sequentially, concurrently, or exclusively (see [7]). The variety of communication components gives rise to corresponding communication system architectures, including the aforementioned service and protocol architecture:

- *protocol-functionality-based architecture*: protocol functionality instances as building blocks (fine granularity)
- *protocol-collaboration-based architecture*: protocol collaboration instances as building blocks (medium granularity, system view)
- *protocol-phase-based architecture*: structuring into protocol phases
- *protocol-entity-based architecture*: protocol entities as building blocks
- *micro-protocol-based architecture*: micro protocols as building blocks
- *layered communication system architecture ("protocol stack")*: sequence of protocol layers, e.g., TCP/IP-protocol stack
- *hierarchical communication system architecture:* nested set of communication systems

A *protocol specification* describes a protocol either *explicitly*, i.e., by directly stating the protocol architecture, the protocol rules and the protocol formats, or *implicitly*, for instance, by operationally defining protocol entity types that follow the protocol rules. In the following, we specify protocols operationally, as this is more common.

3. Language Support of SDL-2000

Design components are directly supported by SDL types contained in SDL packages, which follows the approach of object-oriented class libraries. With SDL being an object-oriented description technique itself, this is straightforward. So, is this already the language solution we are looking for?

The answer to this question is best given by expressive feasibility studies. What we need is the ability to specify a particular kind of component, namely *micro protocols*. Can these be captured by SDL types? And if yes, can we compose them into functionally complete communication protocols? What kind of glue will be needed, and is SDL sufficiently expressive in this respect, too?

In this section, we explain in a generic way how we intend to apply SDL-2000 [4] to specify and compose micro protocols. As it turns out, this has substantial advantages as compared to SDL-96. However, due to the lack of tool support, we have used SDL-96 in the case study in Section 4.

A micro protocol is specified operationally by defining protocol entity types that follow the protocol rules such that one protocol functionality and the required collaboration among protocol entity instances are covered. Methodologically, protocol functionalities and corresponding collaborations are isolated first, and then represented in SDL.

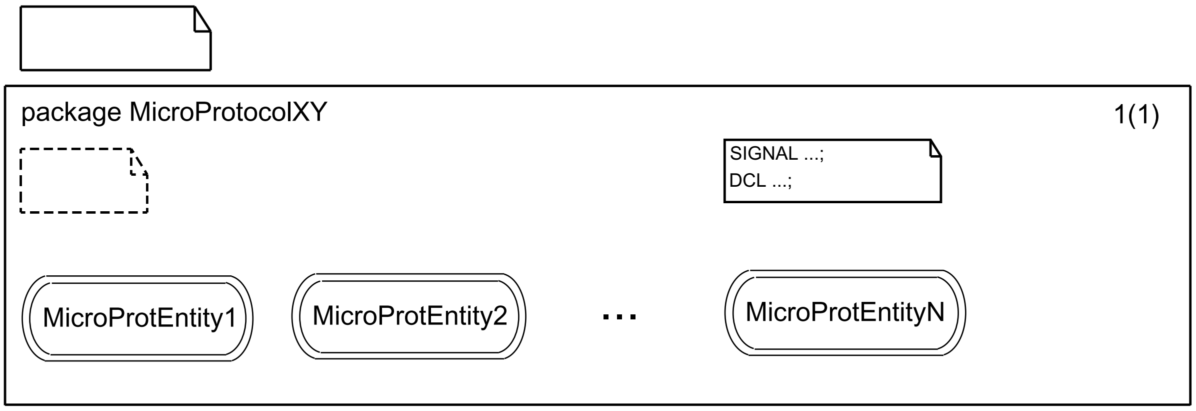

Figure 1: Generic SDL representation of a micro protocol

Our first conceptual design decision is to model protocol entities by asynchronously communicating extended Mealy machines. Obviously, there are several ways to represent them in SDL, for instance, by specifying SDL block types, SDL process types, or SDL composite state types. Which one to use depends on the composition of micro protocols, which depends on the protocol that is to be configured. For instance, composite states can be used in state aggregations, which can express a mutually exclusive composition while sharing local variables. They can also be used in composite state graphs, expressing a sequential composition, possibly with iterations. To express pipelining among micro protocol entities, blocks and processes connected by channels can be used.

As the library of micro protocols should be as generic as possible, the composition of components should not be constrained at this point. Therefore, we choose the most general form and decide to define, for each micro protocol, a set of composite state types grouped in a package with the name of the micro protocol. These state types can then be instantiated in SDL blocks, processes, composite states and state aggregations. Figure 1 shows a generic example, where MicroProtocolXY consisting of composite state types MicroProtEntity1 through MicroProtEntityN is defined. In case of a symmetrical micro protocol, there is just one composite state type. Additionally, some declarations may be needed. Packages can be hierarchically structured, thus forming a library of individual micro protocols.

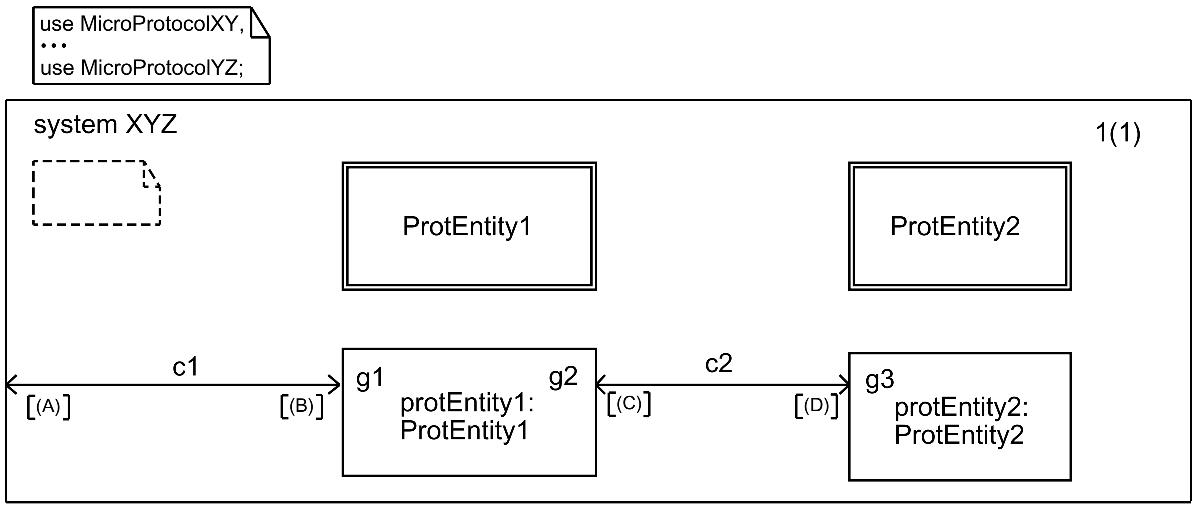

Figure 2: Generic micro protocol composition

Once a micro protocol library is available, we can select and compose suitable micro protocols such that the resulting protocol is self-contained and functionally complete. As we are dealing with components, no adaptation will be necessary. However, the composition will be highly dependent on the functionality to be provided, which requires background knowledge of the protocol engineer that can not simply be expressed in SDL. Once the necessary composition has been determined, we can again use SDL as the language means to express it. Figure 2 illustrates the proceeding: the packages defining the selected micro protocols are imported, and the types contained in these packages are instantiated “suitably”. As discussed before, this could mean to use the composite state types in SDL block types, process types, or to define aggregate states or composite state graphs, depending on the composition style. The additional SDL constructs used here can be understood as “composition glue”. This glue includes block types and process types acting as micro protocol containers, as well as SDL channels introduced to express pipelining. In our case studies so far, we have found that even further transitions may be needed, which may be triggered by *exceptions* raised by specific micro protocol entities [3].

4. The SNMP Case Study

In this section, we will illustrate our methodology through the SNMP (Simple Network Management Protocol) [9] [10]. This protocol has been developed and is being maintained by IETF. It has evolved from version one to version three through the addition of new functionalities. In this paper, we focus on SNMPv1.

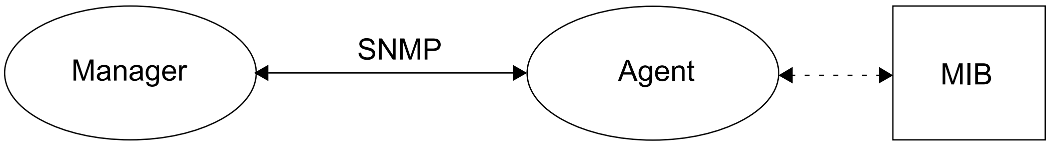

Figure 3: SNMPv1 architecture

The SNMP protocol as such defines the collaborations and exchange formats between a manager and an agent. It is based on a certain number of MIBs (Management Information Bases) that define the management information, i.e. the semantics of data exchanged between a manager and an agent. This information is organized into groups, forming a tree structure. The leaves represent variables to be exchanged between managers and other agents. An agent is responsible of the part of the tree that corresponds to its resource(s).

An SNMP manager communicates with a certain number of agents. A manager may request information from an agent, which basically consists of obtaining the current values of specific variables. The manager may also set the values of certain variables to control the corresponding resource. For instance, the manager may ask the agent to set the values of variables in a routing table in order to influence routing decisions, or to respond to network congestion. An agent may send traps to a manager to inform him of an emergency, such as a severe fault. An architectural view of the protocol is given in Figure 3.

SNMP Functionalities

Based on the informal definition, we can identify three functionalities of the SNMPv1 protocol. The first one is a monitoring functionality, where a manager requests variable values from an agent. The second one is a control function, where a manager asks an agent to set specific variables. The third functionality is the ability of the agent to send traps to a manager. The monitoring functionality can be decomposed further into even smaller, self-contained functionalities. The difference between these functionalities are the parameters carried in the messages, and their interpretation.

The collaborations corresponding to these functionalities are shown in Figure 4, expressed with MSC. For the monitoring functionalities, we have two similar collaborations initiated by a getReq and a getNextReq message, respectively. While in the first collaboration, the variables are listed explicitly, they are referred to implicitly in the second one. In both cases, the agent responds with a getResp message carrying the variables and their values, or with errors, for instance, in case the agent is not able to supply the value of at least one variable. The third MSC defines a typical collaboration of the control functionality, initiated by a setReq message, which carries the variables and their new values. On completing the MIB update, the agent replies with a setResp message. In case of traps, the collaboration is an asynchronous notification, with no response required. Here, the agent sends a trap message carrying a predefined trap type and more specific information about the nature of the trap.

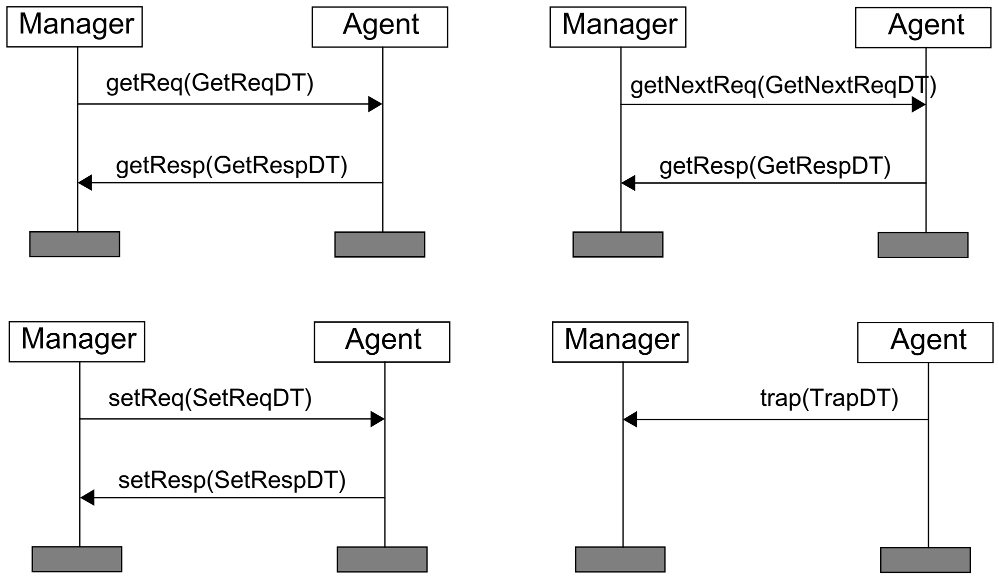

Figure 4: Collaborations of SNMPv1

These four functionalities lead to four micro protocols, namely SNMPget, SNMPgetNext, SNMPset and SNMPtrap. We have specified these micro protocols with SDL-96, the latest version currently supported by commercial tools. Excerpts of these specifications are shown below.

SNMPv1 micro protocols

All definitions belonging to one micro protocol are grouped into one SDL package. In Figure 5, the top level of packages SNMPget and SNMPset specifying the corresponding micro protocols are shown. Packages are imported when a protocol using the contained micro protocol are configured (see below). The package definition slightly differs from the generic case in Section 3, as we have used SDL-96 for reasons already explained. With SDL-2000, a more uniform treatment will be possible.

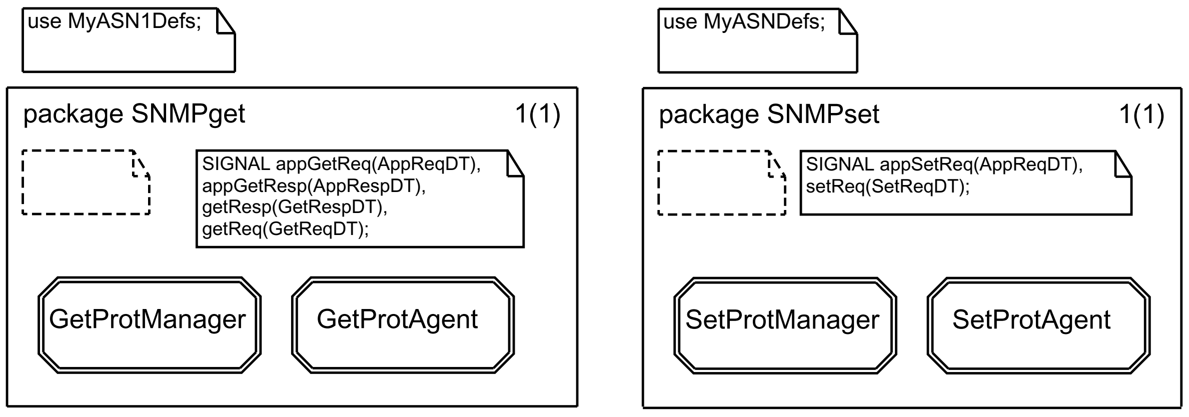
use MyASN1Defs;
package SNMPget
1(1)
SIGNAL appGetReq(AppReqDT), appGetResp(AppRespDT), getResp(GetRespDT), getReq(GetReqDT);
GetProtManager
GetProtAgent
use MyASNDefs;
package SNMPset
1(1)
SIGNAL appSetReq(AppReqDT), setReq(SetReqDT);
SetProtManager
SetProtAgent

Figure 5: SDL packages collecting the micro protocol definitions of SNMPget and SNMPset

SNMPget micro protocol

The SNMPget micro protocol is collected in the SNMPget package. We define two micro protocol entity types, GetProtManager and GetProtAgent, and signals to interact with the management application, i.e. the application using the SNMP service. Notice that our micro protocols are expressed as SDL process types rather than composite state types, which is due to the use of SDL-96.

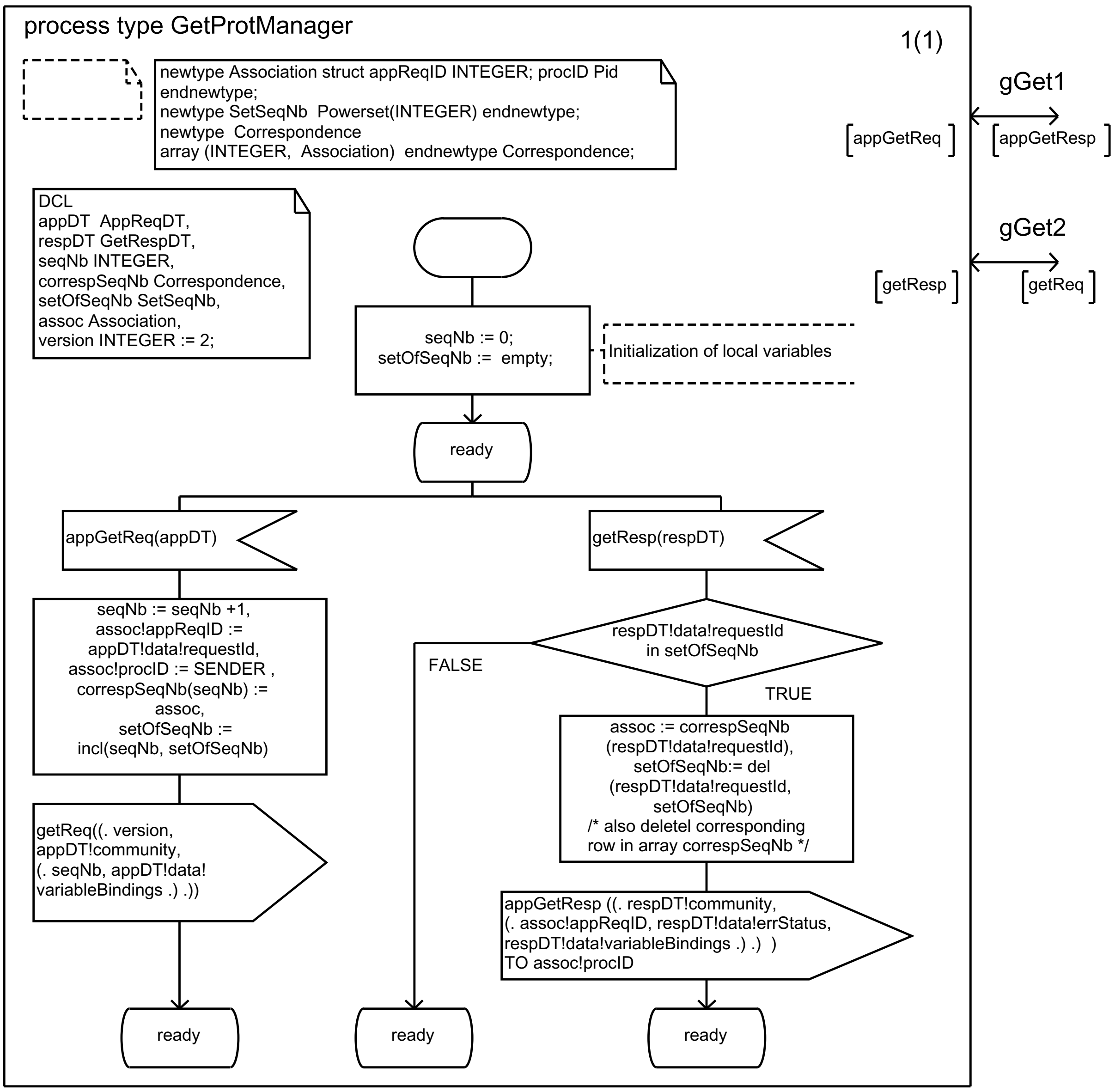

Figure 6: SDL specification of the GetProtManager entity type

The behaviour of the GetProtManager micro protocol entity (see Figure 6) consists of two transitions. In the first transition, GetProtManager receives a request from the management application, and then sends getReq signal. This signal contains a sequence number, which is used for associating the correct response to this request later. Furthermore, it contains parameters such as community, which defines the access rights of the manager. The data types used in the micro protocol are defined in the package MyASN1Defs, which is derived from the predefined data type package in the SNMP standard. In the second transition, the manager receives a response from the agent and checks if this is a response to a pending request using the sequence number, and then forwards the response to the management application.

The behaviour of the GetProtAgent micro protocol entity (see Figure 7) consists of a single transition. On receiving a getReq signal, the entity checks the SNMP version number. If this check is positive, it responds either with the list of requested values or an error.

Composition of SNMPv1 micro protocols

The packages containing the micro protocol definitions are now used to configure a subset of the SNMPv1 protocol. In Figure 8, the packages SNMPget and SNMPset are therefore imported. Process types contained in these packages are instantiated and connected by signal routes. The "composition glue" (see Section 3) is represented by the block types SNMPManager and SNMPAgent, and by the channels and signal routes. Similarly, further micro protocol such as SNMPgetNext and SNMPtrap can be defined and composed.

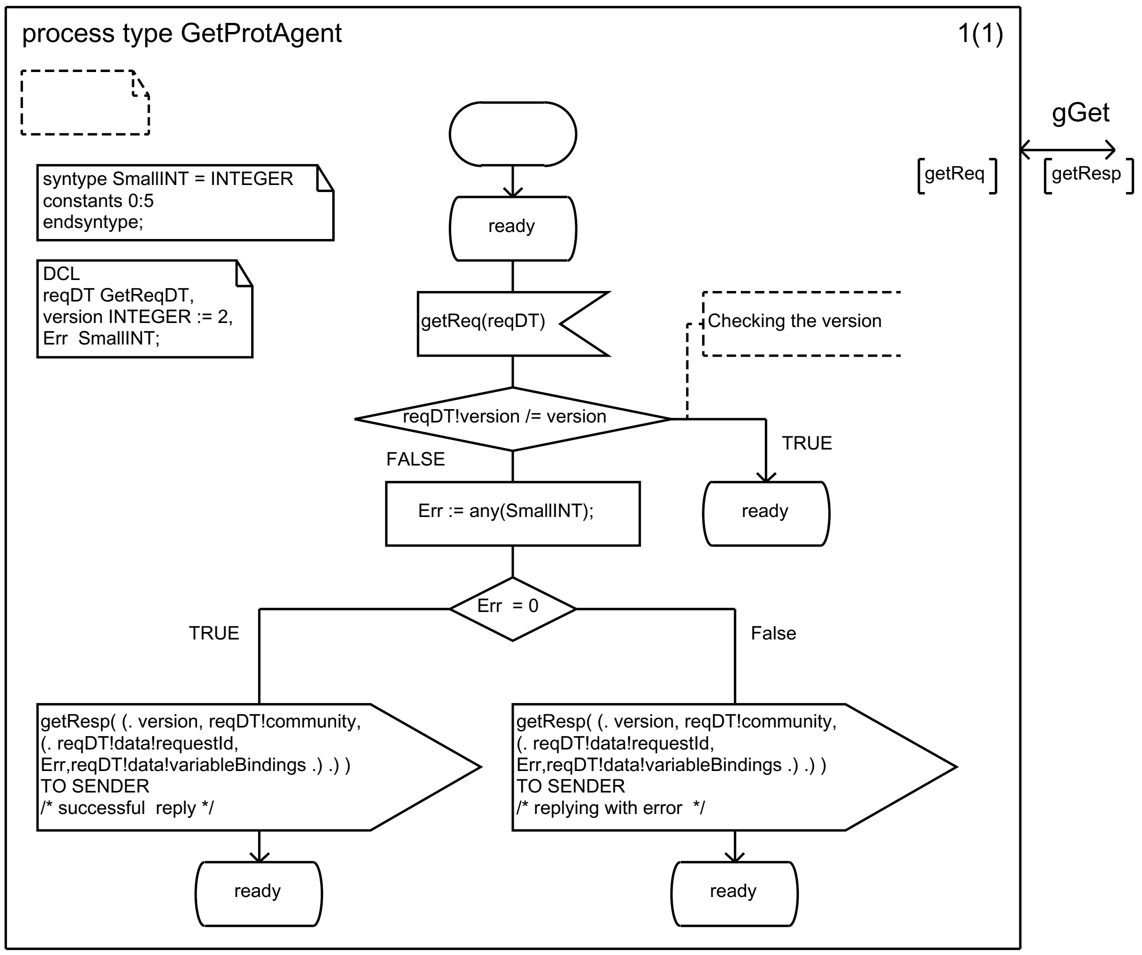

Figure 7: SDL specification of the GetProtAgent entity type

The micro protocols used in this case are very specific to SNMP. They can be used to define SNMPv1, SNMPv2 and SNMPv3 as well as other protocols containing the get and set functionalities. To improve reusability, we can think of *generic micro protocols*. For instance, the micro protocol SNMPtrap could be abstracted to a generic micro protocol OneWayHandshake that has signal parameters. More specific micro protocols are then obtained by instantiating generic micro protocols, making them reusable. Since micro protocols are seen as components here, suitable language support is needed to define generic micro protocols as syntactically complete units.

In the specification examples so far, we have focused on the core SNMP functionalities to demonstrate the concept of micro protocols. Here, SNMPManager and SNMPAgent communicate directly through an SDL channel, which models a perfect network. In a subsequent step, this channel can be replaced by a block modelling a UDP type network. Further micro protocols, for error handling for instance, can be defined and composed with the core functionalities.

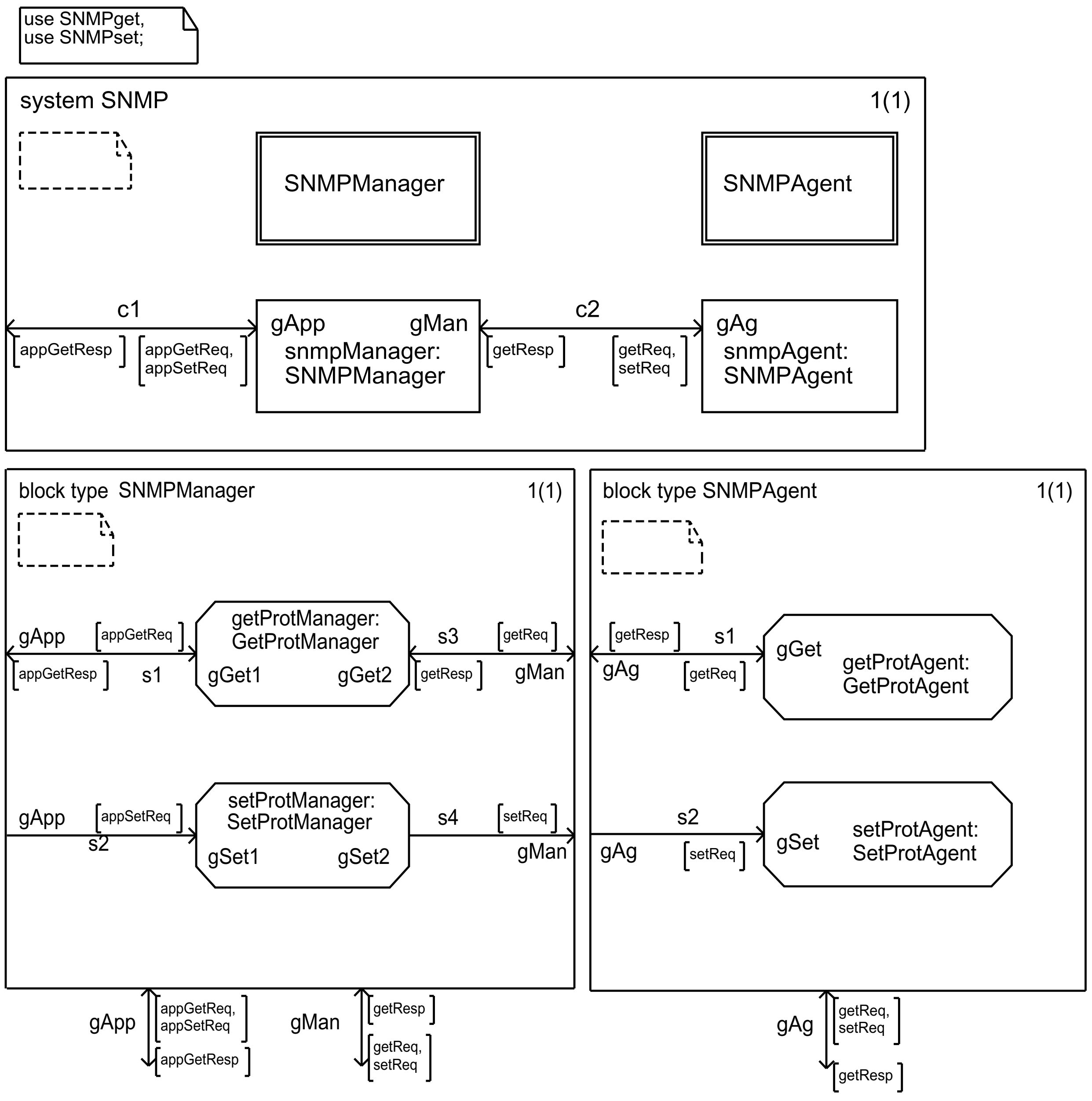

Figure 8: Configuration of a subset of SNMPv1 based on micro protocols

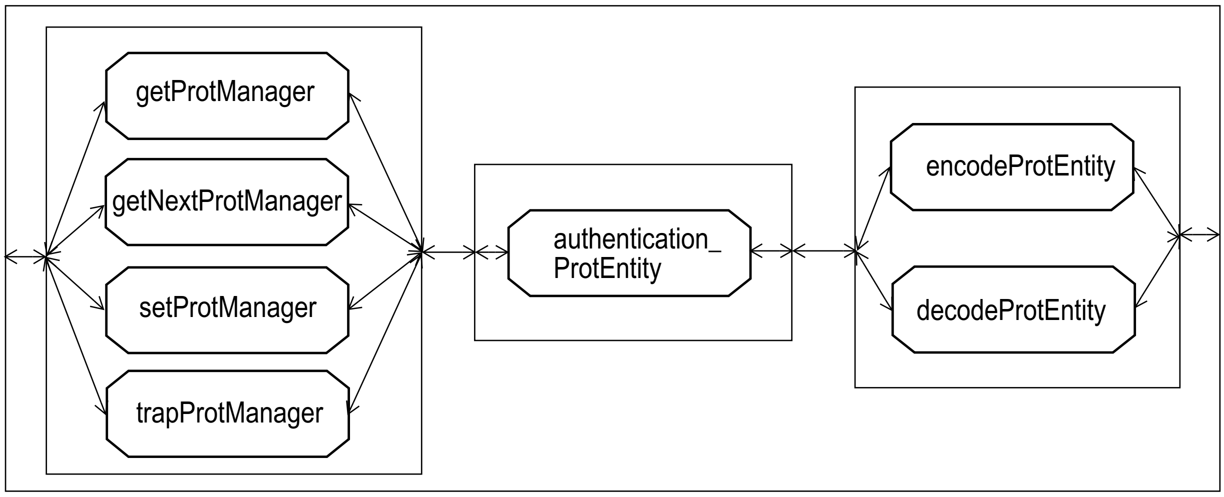

Figure 9: Configuration of an SNMPv1 manager based on micro protocols

Furthermore, the SNMP protocol requires authentication and encoding/decoding of messages. For a more complete SNMP specification, an SNMP entity (SNMPAgent or SNMPManager) is composed in a pipeline with an authentication micro protocol entity and an encoding or decoding micro protocol entity. When an SNMP message is to be transmitted, it is first passed to the authentication micro protocol entity, which performs some transformations on the message. The message is then passed to the encoding micro protocol entity, which performs the encoding of the message using the ASN.1 basic encoding rules. For the reception, the message is passed from the decoding entity to the authentication entity and finally to the responsible SNMP micro protocol entity. The composition of the SNMP manager is shown in Figure 9.

5. Conclusions and Outlook

In this paper, we have explained and applied the concept of *micro protocols*, i.e., self-contained, ready-to-use building blocks that can be applied to configure communication systems designs. In particular, we have examined the suitability of SDL-2000 w.r.t. language support, have specified several micro protocols using SDL-96, and have composed them to yield the functionality of a subset of SNMPv1.

We expect that micro protocols can foster reuse in the protocol engineering domain. As in other areas, reuse of solutions and experience for recurring system development problems plays a key role for quality improvements and an increase in productivity. Furthermore, micro protocols are a natural way of structuring communication systems, which may enable compositional testability and verification.

The micro protocol types we have specified so far are generic in the sense that they can be composed in different ways and combinations in order to design a communication system. However, there is potential to further boost reusability. The micro protocol type SNMPtrap, for instance, can be abstracted to a more generic type OneWayHandshake that has signal parameters. Thus, OneWayHandshake would still be syntactically complete and qualify as a component, while it can be specialized to yield SNMPtrap just by signal renaming and specialization of parameters. In general, more specific micro protocols are obtained by instantiating generic micro protocols. Work in this direction is in progress.

Work done so far shows that this micro protocol approach to protocol design is feasible. We have applied the ideas to a subset of SNMP, the Simple Network Management Protocol used in the Internet, and have achieved encouraging results. Yet, more experience is needed, and a library of micro protocols has to be built.

References

[1] B. Geppert, R. Gotzhein, F. Rößler: *Configuring Communication Protocols Using SDL Patterns*, in: A. Cavalli, A. Sarma (eds.), SDL'97 - Time for Testing, Proceedings of the 8th SDL Forum, Elsevier, Amsterdam, 1997, pp. 523-538

[2] R. Gotzhein, P. Schaible: *Pattern-Based Development of Communication Systems*, in: Annals of Telecommunications, Special Issue on Protocol Engineering, Vol. 54, No. 11-12, 1999, pp. 508-525

[3] R. Gotzhein, F. Khendek: *Conception avec Micro-Protocoles*, Colloque Francophone sur l'Ingenierie des Protocoles (CFIP'2002), Montreal, Canada, May 27-30, 2002

[4] ITU-T Recommendation Z.100 (11/99): *Specification and Description Language (SDL)*, International Telecommunication Union (ITU), 1999

[5] R. E. Johnson: *Frameworks = (Components + Patterns),* in: Object-Oriented Application Frameworks (Special Issue), Communications of the ACM, Vol. 40, No. 10, 1997, pp. 39-42

[6] T. Plagemann, B. Plattner, M. Vogt, T. Walter: *Modules as Building Blocks for Protocol Configuration*, Proceedings of the International Conference on Network Protocols (ICNP'93), San Francisco, 1993

[7] F. Rößler, B. Geppert, R. Gotzhein: *Collaboration-based Design of SDL Systems*, Proceedings of the 10th SDL FORUM, June 2001

[8] M. Zitterbart, B. Stiller, A. Tantawy: *A Model for Flexible High-Performance Communication Subsystems, IEEE Journal on Selected Areas in Communications*, Vol. 11, No. 4, 1993, pp. 507-518

[9] J. Case, M. Fedor, M. Schoffstall, J. Davin: *A Simple Network Management Protocol*, RFC 1157, May 1990

[10] W. Stallings: *SNMP, SNMPv2, SNMPv3, and RMON 1 and 2*, Addison Wesley, Third Edition, 1999

[11] C. Szyperski: *Components and architecture*, Beyond Objects column, Software Development. Vol. 8, No. 10, October 2000

[12] HORUS at http://www.cs.cornell.edu/Info/Projects/HORUS/