J. Lopez, J. J. Ortega, and J. M. Troya, "Applying SDL to Formal Analysis of Security Systems", *11th International SDL Forum (SDL03)*, LNCS vol. 2708, pp. 300-317, 2003.
NICS Lab. Publications: https://www.nics.uma.es/publications

Applying SDL to Formal Analysis of Security Systems

Javier López, Juan J. Ortega, José M. Troya

Computer Science Department, E.T.S. Ingeniería Informática
University of Malaga, 29071 - Malaga - SPAIN
jlm@lcc.uma.es, juanjose@lcc.uma.es, troya@lcc.uma.es

Abstract. Nowadays, it is widely accepted that critical systems have to be formally analyzed to achieve well-known benefits of formal methods. To study the security of communication systems, we have developed a methodology for the application of the formal analysis techniques commonly used in communication protocols to the analysis of cryptographic ones. In particular, we have extended the design and analysis phases with security properties. Our proposal uses a specification notation based on MSC, which can be automatically translated into a generic SDL specification. This SDL system can then be used for the analysis of the desired security properties, by using an observer process schema. Apart from our main goal of providing a notation for describing the formal specification of security systems, our proposal also brings additional benefits, such as the study of the possible attacks to the system, and the possibility of reusing the specifications produced to describe and analyze more complex systems.

1 Introduction

Formal methods characterize the behavior of a system in a precise way and can verify its formal specification. In particular, the design and analysis of security systems can greatly benefit from the use of formal methods, due to the evident critical nature of such systems.

During recent years, the cryptographic protocol analysis research area [1] has experienced an explosive growth, with numerous formalisms being developed. We can divide this research into three main categories: logic-based [2], model checking [3–5], and theorem proving [6]. Although all three approaches have shown their applicability to simple problems, they are still difficult to apply in real, more complex environments such as distributed systems over the Internet.

Moreover, we believe that the results obtained in the analysis of cryptographic protocols do not have a direct application in the design of secure communication systems. Probably, one of the major reasons for that is the lack of a strong relationship between the analysis tools for security systems and the formal methods techniques commonly used in the specification and analysis of communication protocols. Trying to bridge this gap is one of the major contributions of our work.

We have developed a methodology [7, 8] for the specification of secure systems, which also allows us to check that they are not vulnerable against both well-known and originals attacks. Our approach uses a requirement language (SRSL) to describe security protocols, which can then be automatically translated into SDL [9], a widely used formal notation specifically well suited for the analysis of protocols. In addition, we have developed some verification procedures and tools for checking a set of security properties, such as confidentiality, authentication, and non-repudiation of origin. In our approach we use a simple but powerful intruder process, which is explicitly added to the specification of the system, so that the verification of the security properties guarantees the robustness of the protocol against attacks of such an intruder. This is known as the Dolev-Yao's method [10].

Because SRSL is an extension of MSC [11], available editors for MSC and SDL can be used for writing SRSL specifications, as well as standard code-generators and SDL validation tools. In particular, we have built our translators and analyzing tools using Telelogic's Tau SDL Suite.

The structure of this document is as follows. After this introduction, Sect. 2 defines the security concepts and mechanisms used throughout the paper. Then, Sect. 3 provides an overview of our proposal. The SRSL language is presented in Sect. 4, while Sect. 5 discusses how the SRSL descriptions can be automatically translated into SDL, and how the SDL specifications produced can be analyzed for proving security properties. Finally, Sect. 6 draws some conclusions and outlines some future work.

2 Specification of Security Properties

A security protocol [12] is a general template describing a sequence of communications, which makes use of cryptographic techniques to meet one or more particular security-related goals. In our context we will not distinguish between cryptographic and security protocols, considering both to be equivalent. The international organization ITU-T has defined Recommendation Series X.800 [13, 14] to specify the basic security services. Among these, the ones provided by the basic security mechanisms (cryptographic algorithms and secure protocols) are authentication [15], access control [16], non-repudiation [17], data confidentiality [18], and data integrity [19].

The notion of *authentication* includes both authentication of origin and entity authentication. *Authentication of origin* can be defined as the certainty that a message that is claimed to proceed from a certain party was actually originated from it. As an illustration, if Bob receives a message during the execution of a protocol, which is supposed to come from Anne, then the protocol is said to guarantee authentication of origin for Bob if it is always the case that, if Bob's node accepts the message as being from Anne, then it must indeed be the case that Anne has sent exactly this message earlier. Authentication of origin must be established for the whole message. Additionally, it is often the case that certain time constraints concerning the freshness of the received message must

also be met. *Entity authentication* guarantees that the claimed identity of an agent participating in a protocol is identical to the real one.

Access Control service. ensures that only authorized principals can gain access to protected resources. Usually, the identity of the principal must be established, hence entity authentication is also required here.

Non-repudiation. provides evidence to the parties involved in a communication that certain steps of the protocol have occurred. This property appears to be very similar to authentication, but in this case the participants are given capabilities to fake messages, up to the usual cryptographic constraints. Non-repudiation uses signature mechanisms and a trusted notary. We will distinguish two types of non-repudiation services: non-repudiation of origin (NRO) and non-repudiation of receipt (NRR). NRO is intended to prevent the originator's false denial of having originated the message. On the other hand, NRR is intended to prevent the recipient's false denial of having received the message.

Confidentiality. may be defined as the prevention of unauthorized disclosure of information. In communication protocols, this means that nobody who has access to the exchanged messages can deduce the secret information being transmitted.

Data Integrity. means that data cannot be corrupted, or at least that corruption will not remain undetected. Accepting a corrupted message is considered as a violation of integrity, and therefore the protocol must be regarded as flawed.

These services are commonly enforced using cryptographic protocols or similar mechanisms. It is worth noting that, To specify a security system, it is not necessary to know how the system is going to be analyzed, but it is essential to identify the security services required.

Now, considering the system from the attacker's perspective, additional security protocol vulnerabilities can be defined: (a) *man-in-the-middle*, where the intruder is able to masquerade a protocol participant; (b) *reflection*, where an agent emits messages and studies the system's answers; (c) *oracle*, where the intruder tricks an honest agent by inadvertently revealing some information (notice that such an attack may involve the intruder exploiting steps from different runs of the protocol, or even involve steps from an entirely different protocol); (d) *replay*, in which the intruder monitors a (possible partial) run of the protocol and, at some later time, replays one or more of the protocol's messages; (e) *interleave*, where the intruder contrives for two or more runs of the protocol to overlap; (f) *failures of forward secrecy*, in which the compromised information is allowed to propagate into the future; and (g) *algebraic attack*, where it is possible for intruders to exploit algebraic identities to undermine the security of the protocol. Please note that these kinds of attacks depend on the environment of the system (network, users, ...), and therefore not all of them are always achievable in a given context. However, we will study all potential situations, trying to cover them all in all cases.

To keep clear the focus of the paper, the following assumptions have been made. First, we suppose that cryptography is perfect, so no cryptanalysis techniques are used. Second, all agents may freely and perfectly generate random numbers. And finally, we do not consider interactions with any other protocols, because this is an open research topic.

3 Methodology Overview

Our approach (depicted in Fig. 1) performs the design and analysis of security protocols in the same way the design and analysis of a traditional communication protocols is accomplished, but including the security aspects.

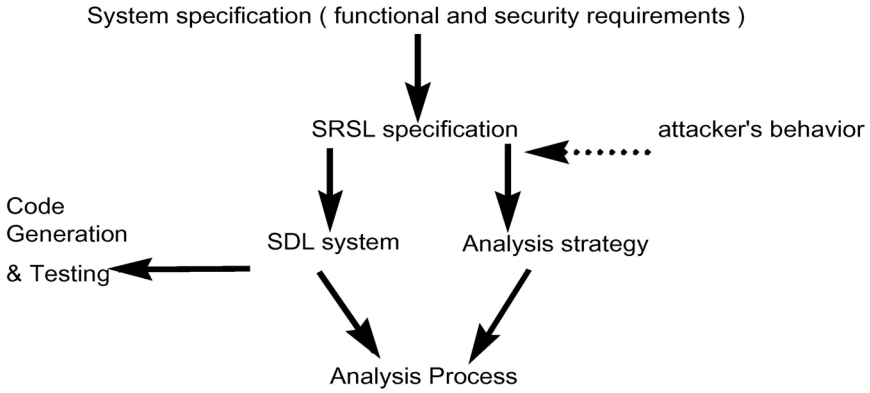

Fig. 1. Overview of Our Approach

In the first place, we need to gather the functional and security requirements of the system in any (usually informal) way. These informal specifications, together with the behavior about the kinds of possible attacks (if available), is the sort of information that can be described using our *Security Requirements Specification Language* (SRSL).

SRSL is an extension of MSC, augmented with textual tags. We make use of the MSC text area to include these tags, which are used to identify the security characteristics of the data being transmitted, the intruder's possible activities, and the security analysis goals. In case the attacker's behavior is not explicitly provided, we automatically generate a generic process that tries to examine all possible attacks.

For drawing the graphical SRSL specifications, any standard MSC and HMSC editor can be used. In our case, we have used Telelogic's TAU, which also allows

the automatic translation of the graphical MSC diagrams into their corresponding textual form. A translator program is then used to obtain the SDL system from the SRSL descriptions. (This program has been written in C, using plain LEX and YACC tools.) The SDL system produced is composed of: (1) a package with the data types of the system for the analysis; (2) a package with one process type for each protocol agent; and (3) a collection of process types ("observer" and "medium") for the analysis strategy.

To analyze the security properties, we evaluate the behavior of the SDL system under different kinds of attacks (as specified by the medium processes defined in the analysis strategy). The observer process provided by the TAU Validator tool is used for these checks. Thus, we can check whether a specific state is reached, or whether a particular data is ever stored into the intruder's database knowledge.

```
Security_information ::= definition_section security_service_section

Defintion_section ::= Definition var_definition  knowledge_section

var_definition ::= <varlist> : Agent ;
      | <varlist> : Text ;
      | <varlist> : Random ;
      | <varlist> : Timestamp ;
      | <varlist> : Sequence ;
      | <varlist> : Public_key ;
      | <varlist> : Symmetric_key ;
      | <varlist> : Shared_key ;
      | <varlist> : Session_key ;

Knowledge_section ::= Knowledge <listagent_id> : <varlistasig> ;

Security_service_section ::=  [intruder_strategy] security_property

intruder_strategy ::= Session instances [ <var>=<value> ] ;
      | intruder_knowledge [ <initial_knowledge> ];
      | intruder [ redirect | , impersonate | , eavesdrop ] ;

security_property ::= Security_service <security_service_list> ;

security_service_list ::= authenticated ( <agent> <agent> )
      | conf ( <data> )
      | NRO ( <agent> <data> )
```

Fig. 2. SRSL Security Section Syntax

We also make use of the TAU Validator *assert* mechanism, which enables observer processes to generate reports during the state space exploration. These reports are maintained by the Report Viewer, and can be examined to identify security flaws.

Currently, confidentiality and authentication [20] are the security properties usually analyzed. By analyzing confidentiality we prevent the intruder from being

able to derive the plaintext of messages passing between honest nodes. Our analysis consists of checking if the secret item can be deduced from the protocol messages and the intruder's database knowledge.

An authentication protocol is considered to be correct if a user Bob does not finish the protocol believing that it has been running with a user Alice unless Alice also believes that she has been running the protocol with Bob. Our analysis consists of looking for a reachable state where Bob has finished correctly and Alice will never reach her final state.

We also analyze non-repudiation of origin. For that we define the evidence of origin and who produces it (the origin). Our analysis consists of checking that the evidence is digitally signed by the origin agent, and that it cannot be created by any other agent.

In addition, the SDL system generated from the SRSL specifications can be used to automatically generate C or C++ code, which can interact with exiting applications. In order to generate this code we need to replace the data types package with a corresponding package that defines the data types in ASN.1 or C. This prototype can also be used for testing, which is part of our future work.

4 The SRSL language

The main aim of SRSL is to define a high-level language for the specification of cryptographic protocols and secure systems. As pre-requisites for this language we need to ask it to be modular to achieve reusability, to be easy to learn, and to incorporate security concepts.

As a natural base for SRSL we considered the requirements language most widely used in the telecommunications: the Message Sequence Chart (MSC) and its extension High-level MSC (HMSC). With MSC we can specify elementary scenarios, and compose them to define more complex protocols with HMSC. The version we have considered is previous to the MSC 2000 release [21], but we believe that some features of this release are very useful.

SRSL is divided into two main parts. The first one contains the definition of the protocol elements and the security analysis strategy. The second part describes the message exchange flow.

The first part is textual. The syntax of its main elements is shown in Fig. 2. These elements can be grouped into different categories, and are listed below (language keywords are written in *italics*):

- Main elements:
  - ⋆ Entities: *Agent*, principal identification;
  - ⋆ Message: *Text*, message text; *Random,* number created for freshness, also called nonce; *Timestamp*, actual time; *Sequence*, counter.
  - ⋆ Keys:
    Public_key public-key cryptographic, formed by a pair of public and private keys;
    Symmetric_key used for symmetric encipher;

Shared_key symmetric key shared by more than one entity;
Session_key a fresh symmetric key used to encrypt transmission.

- The "knowledge" section contains the information needed to describe the initial knowledge of each party of the protocol.
- The "security service" section is split into the "intruder strategy" section and the "security property" section. The first one defines a possible attack scenario. The second one describes which security property we try to achieve with this protocol. We have used three different security statements: *Authenticated(A,B)*, stating that B is certain of the identity of A; *conf(X)*, stating that the data X cannot be deduced (also called confidentiality); and *NRO(A,X)*, or non-repudiation of origin, which states that the data X (the evidence) must have been originated in A. These statements have a formal description which is used to analyze them.

The message exchange flow is described using the standard MSC and HMSC facilities. MSC references are used to achieve reusability. We have specified a set of standard protocols in SRSL, that can be easily re-used in different contexts, and combined together to describe more complex protocols using their MSC references.

Messages consist of an identification name (either a text string describing the meaning of the message, or a simple counter sequence), and the message parameters (which define the message data type format).

Some cryptographic operations can be applied to messages: Concatenate (",") for data composition; Cipher ({<plaintext>} <key>) to cipher data; Decipher("*decrypt*(<cipher_data>,<key>)") to extract the plaintext; Hash("<hash-function> (<data>)"), result of a one way algorithm; and Sign([<plaintext>] <Public_Private_key>"), for getting a hash encrypted message with the signer's private key. Further cryptographic functions can be defined if required.

In addition, the MSC expressions constructed using the inline MSC operators *alt*, *par*, *loop*, *opt* and *exc* can also be used.

The keyword *alt* denotes alternative executions of several MSCs. Only one of the alternatives is applicable in an instantiation of the actual sequence.

The *par* operator denotes the parallel execution of several MSCs. All events within the MSCs involved are executed, with the sole restriction that the event order within each MSC must be preserved. An MSC reference with a *loop* construct is used for iterations and can have several forms. The most general construct, *loop*<n,m>, where n and m are natural numbers, denotes iteration at least n and at most m times. The *opt* construct denotes a unary operator. It is interpreted in the same way as an *alt* operation where the second operand is an empty MSC. An MSC reference where the text starts with *exc* followed by the name of an MSC indicates that the MSC can be aborted at the position of the MSC reference symbol, and instead continued with the referenced MSC.

To illustrate our approach we will specify here a typical secure web access to a data bank portal via the Internet. Figure 3 shows the SRSL specification of the system in SRSL, that uses two agents: "User_Browser" and "Bank_Portal". The "Bank_Portal" agent has a secure web service via the HTTPS protocol, which

provides authentication of the server. This is represented by an MSC reference called "https_server_auth" that implements the server authentication and the key exchange protocol defined in HTTPS. This MSC reference is defined in a package of standard protocols. The results of this scenario is authentication of the server and a session key called "https_skey". The first security requirement "*Authenticated (User_Browser, Bank_Portal)*" is achieved with this protocol.

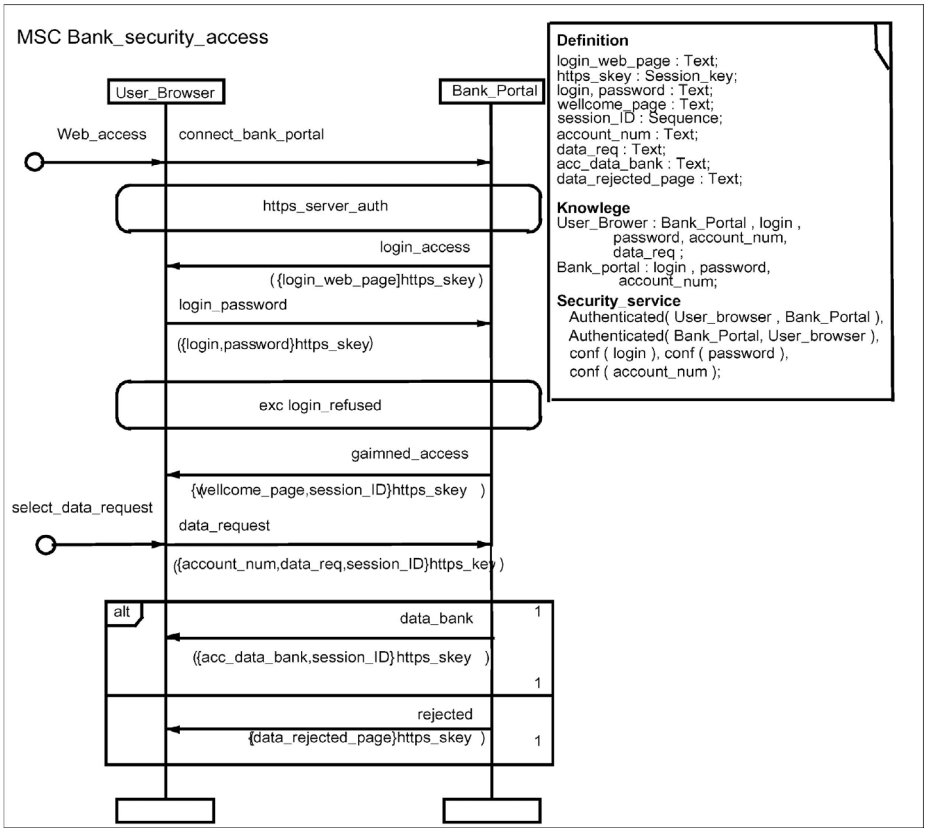

Fig. 3. SRSL Security Scenario of User's Web Access to Bank

The second requirement means that the user must authenticate itself to the bank's portal. This is accomplished by a mechanism that asks for the user's identification (login) and password, and subsequently validates it. The exception MSC reference called "login_refused" is active if the login-password authentication process of the "Bank_Portal" fails. Notice that all messages are ciphered by the https protocol session key.

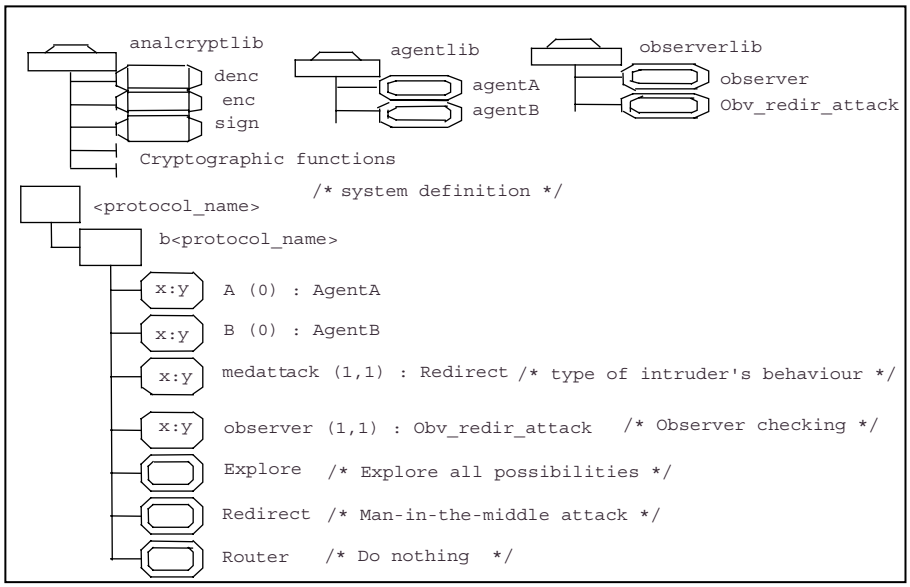

Fig. 4. Generic SDL System Overview

The last three requirements mean that the data transmitted is confidential. This goal is accomplished by making use of the session key established during the https connection.

Of course, other alternative security mechanisms could have been considered for specifying this system, which also met the five original requirements. The important point to note here is that we have chosen a form of specification that does not bind the developer to any particular security mechanisms, thus achieving separation of concerns and modularity. This is accomplished by allowing the security requirements to be defined at a higher level of abstraction, and independently from the system's functional requirements.

In the case of a system that is already implemented (a legacy system) that we want to analyze or document, we can describe instead the security mechanisms that have been implemented.

5 Security Analyses

We use SDL for the security analyses. In the first place, we need to build an SDL system from the SRSL specifications. We have developed a program that automates this process. The program is written in the C language, and uses LEX and YACC standard tools. The input file is a protocol specification written in SRSL, and the program produces a valid SDL system. The generated SDL system (depicted in Fig. 4) is composed of three packages, and contains several processes.

The SDL package that defines the system data types and their operators is called "analcryptlib". It also contains elementary security data types, and the message format definition used in the protocol. This information is used by the rest of the system.

Another SDL package defines a process type for each principal agent. They are implemented in a standalone fashion so they can be reused in different situations. Figure 4 shows two process types, which reference agents A and B respectively.

The last package is about the observer processes. They implement the assert mechanism used in the validation process, and depend on the medium process (called "medattack" in Fig. 4), and on the security services that will be evaluated.

The SDL system (shown in Fig. 5) is named after the protocol it defines, and consists of a single SDL block, which is composed of the process structure for analysis ("A", "B", "medattack"), an observer process ("observer") and several medium process types ("explore", "redirect", and "router"). The process structure for the analysis consists of a medium process that controls all transmissions among agent processes. This control implements the attacker's procedure.

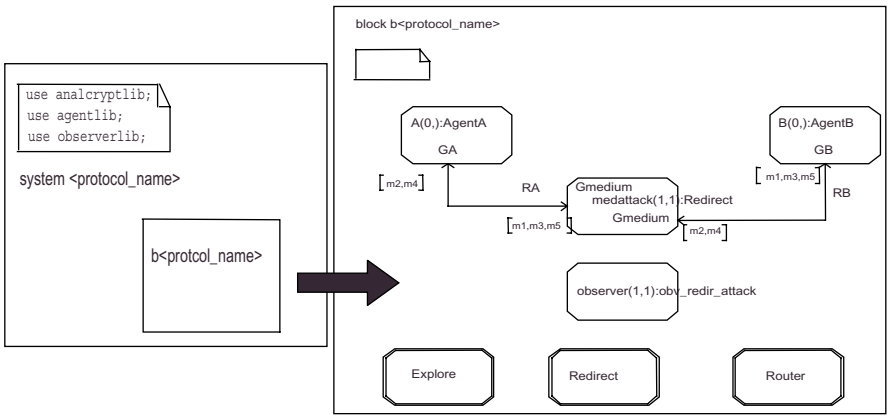

Fig. 5. SDL System Configuration

Please note that medium process types have to be created inside this block because they implement the intruder's behavior, and therefore they may create process agent instances. The TAU tool we use requires all process instances to be defined within the same block. The following describes in detail all the system parts, and how the SRSL specifications are mapped onto the SDL description of the system.

5.1 Data Types Package

An SDL package contains the data types and the cryptographic functions used in the SRSL specification of the system. We may consider an SDL package for performing the analysis, and other package (written in ASN.1) for code generation. All cryptographic data and operators are standardized using ASN.1 notation, following PKCS standards [22].

Since the SDL data types do not support recursive definitions, we make use of enumerated and structured data types. The elemental data types defined in Sect. 4 are then mapped to enumerated SDL *struct* sorts. An example is depicted in Fig. 6.

Fig. 6. Example of translation of security data to SDL *struct*

The messages, which are sent by protocol agents, are constructed by concatenation of elemental data types and cryptographic operations. We define a *struct* sort for each message, and set of elemental data types. The cryptographic functions are then applied to a set of elemental data types called "TENCMESS". This is shown in Fig. 7

Freshness or temporary secrets are implemented by adding an item that references the process instance values. In particular, we use the SDL sort *PID* for this purpose.

Furthermore, we define a "set of knowledge" type for each data type. The analysis methods use these types to store message knowledge in order to prove the specified security properties.

5.2 Agents Package

The generic model identifies each protocol agent with an SDL process type. All process types are stored in a package called "agentlib" so they can be used in other specifications. An agent specification is totally independent from the rest of the system, so they are generated in separate modules. In addition, the specification allows concurrent instances, so we can evaluate this behavior in the analysis phase.

The generic state transition of an agent process is triggered when it receives a correct message (a message accepted by the agent). Then, either the next message is composed to be sent to the receiver agent, or the process stops if the protocol's final state is reached for this process. If the message is not correct, the process returns to the state where it is waiting for messages.

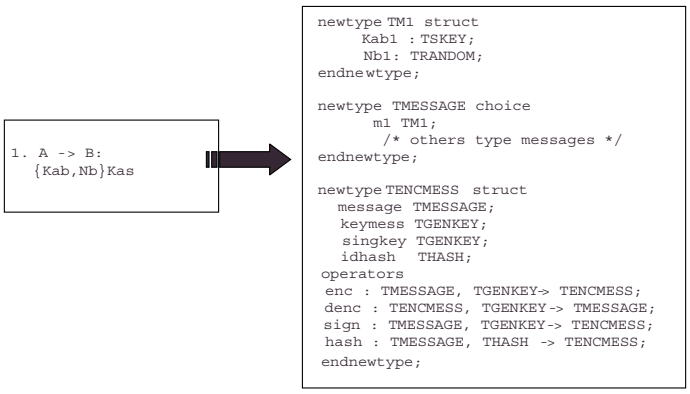

Fig. 7. Example of Translation Belonging to the First Message of a Security Protocol

The MSC expressions used in SRSL are mapped into SDL as follows: an *alt* expression produces several signal trigger states; a *loop* expression makes all next transitions return to the initial section state; an *opt* expression is implemented by a *continue* signal; and finally, an *exec* expression is translated into an asterisk state.

An SDL process is a finite state machine, and therefore it finishes when it executes a stop statement, or provides a deadlock if no signal arrives. Our model has to explore all possibilities. Hence, we need to develop a mechanism to ensure that all signals sent must be processed. Consequently, we have added a state called "final" to indicate the end of the protocol execution, and a general transition composed of a common "save" statement and a continuous signal, with less priority than the input statement, that checks whether there are signals still waiting to be processed. By means of this structure we are transforming a finite state machine into an infinite one, just for analysis purposes.

At this point, if we instantiate the medium process with a "Router" process type, we can specify a security protocol in the same way as we might specify a traditional communication protocol, and therefore we can analyze some of the liveness properties of the system in a traditional way. In the next subsection, we are going to explain how the security properties can be checked.

5.3 Model Medium-Observer Processes

In our approach, the intruder's behavior is divided into two main aspects, the exploration algorithm and the check mechanism. The first one is provided by a medium process, while an observer performs the check mechanisms.

We can consider two kinds of medium processes. The first one is characterized by an exploration mechanism that tries to explore all possibilities. It starts by examining all combinations of the different initial knowledge of each agent.

Afterwards, it checks the concurrent agents' execution, by first trying combinations of two concurrent sessions, and so on. Our algorithm finishes when an "out of memory" is detected, or when it detects that the significant intruder knowledge is not incremented. In general, the completeness problem [23, 24] is undecidable. Thus, the fact that the algorithm terminates without having found a flaw, is not a proof that there are no flaws.

The second kind of medium process uses an intruder process specialized in finding a specific flaw. If we are able to characterize a particular kind of attack, we can then evaluate the protocol trying to find such a specific flaw. Perhaps this is not the best solution in general (the only result we get is that a specific vulnerability does not occur in the cases we have examined), but it is very useful for a protocol designer that wants to be sure that the protocol is not vulnerable with respect to that kind of attack.

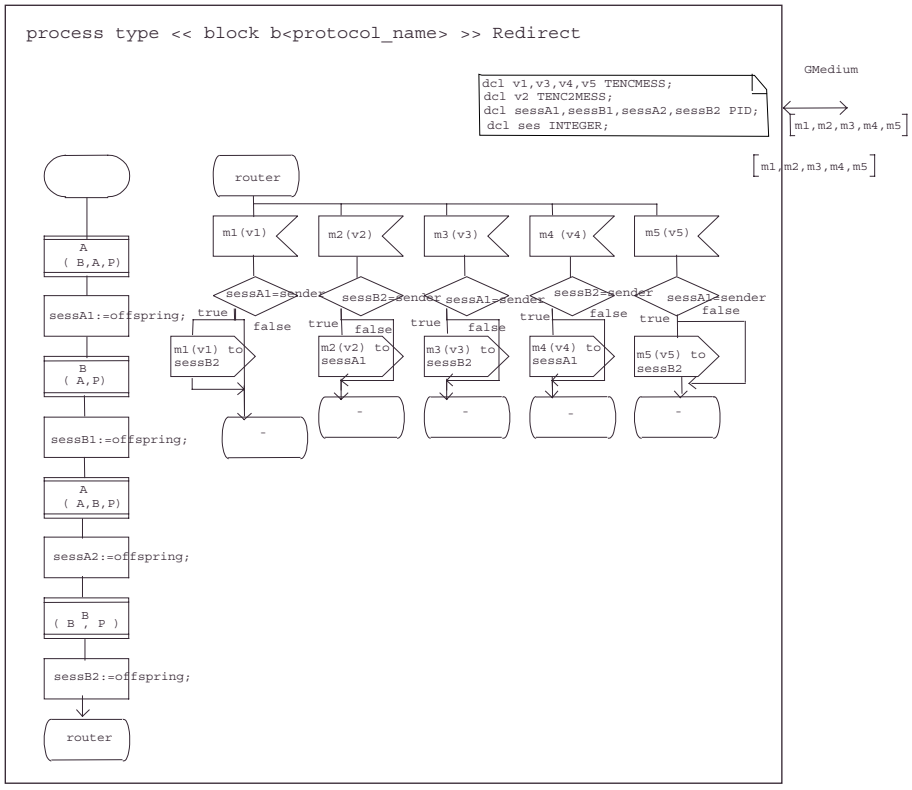

Fig. 8. Implementation of "redirect" Intruder's Behavior

The state transition of the medium process is triggered when it receives any message. After reception, the message is stored into the intruder's knowledge database. The intruder then decides which operation performs next, and proceeds to the next routing state. We have defined three different operations: *eavesdrop*, *redirect*, and *impersonate*. In an *eavesdrop* operation, the intruder intercepts the message but does not send it to any agent. A *redirect* operation means that the intruder intercepts the message but does not forward it to the original receiver. In an *impersonate* operation, the intruder sends a faked message to the original receiver.

Under the EU-funded project CASENET we are currently investigating the use of the protocol developer SAFIRE tool [25] to execute the medium processes. Even if we have to modify these processes to use the tool, we may easily obtain an environment for testing intruders' strategies. This is an open research item.

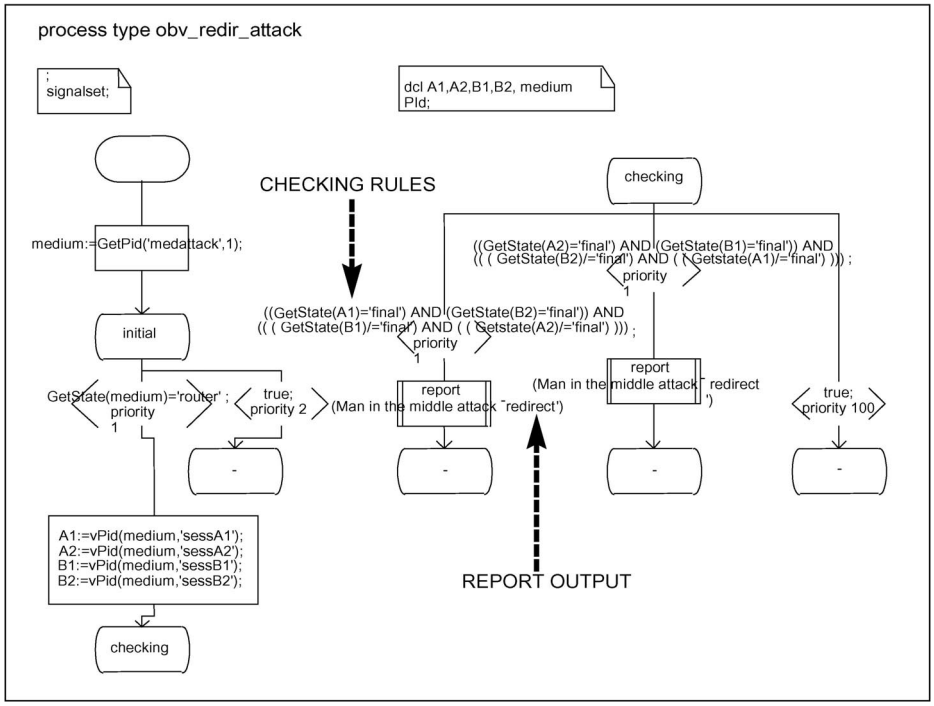

Fig. 9. Example of Observer Process Type that Checks Correspondence Flaw

The security properties are proved using condition rules. These rules check different situations where protocol vulnerability is possible. The observer process carries out the checking mechanism. This is a special SDL process type that is evaluated in each transition of the protocol specification. It has access to all variables and states of all process instances, so we can test it automatically.

To implement it we have to create an SDL *struct* sort with a "v" operator for each structured type that we want to evaluate. Figure 9 shows an example of an observer process. We can see the condition rules for checking the authentication property, and the report result. The report contains a security failure MSC scenario.

Currently, confidentiality, authentication, and non-repudiation of origin can be checked. For checking confidentiality, we examine whether a specific value (that we consider secret) can be deduced from the intruder's knowledge. Authentication is analyzed by checking that all the principal processes finish at the expected protocol step. Some authors [20] call this the correspondence (or precedence) property. Finally, non-repudiation of origin analysis consists of checking that, in the intruder's database knowledge, the evidence is signed digitally by the origin agent, and that it cannot be generated without this signature, and not even in another protocol run.

In order to validate our proposal we have carried out the analysis of some of the most classic cryptographic protocols, such as the Needham-Schroeder symmetric key, and the secure socket layer (SSL). Ref. [7] describes the results of the analysis for the authentication protocol Encrypted Key Exchange (EKE) [26]. This protocol was specified in SRSL, using a well-known "man in the middle" attack evaluating two executions running in parallel sessions. The attack followed the *redirect* intruder's behavior. The resulting scenario describes a situation where only one of the two agents in each session has finished (agent A of the first session, and agent of B of the second one), but not the other.

6 Conclusions

We have presented a new analysis method for analyzing and evaluating security protocols and their possible attacks. Security protocols are specified in SRSL, which can then be translated into a working SDL system. Attacks are implemented by SDL processes that specify the intruder's behavior and observer processes that check safety properties. One of the benefits of our approach is that protocol specifications are described independently from the analysis procedures, so they can be re-used in other environments as well.

Several kinds of security attacks can be analyzed using our approach. It is essential to study how they can be produced in a real environment. We examine the result scenario provided in an analysis procedure, and redesign the security protocol if necessary.

We have applied this method in complex systems, for instance in electronic contract signing. The SRSL specification helped to implement it and draw attention to security services and their related mechanisms. Furthermore, we have simulated several security critical scenarios in order to verify the security properties.

Currently we are extending SRSL so more complex protocols can be specified, and to analyze other properties. We are studying the use MSC-2000 features.

Furthermore, we are developing a framework to implement for testing protocol attacks in the Internet environment.

Acknowledgments. The work described in this paper has been supported by the European Commission through the IST Programme under Contract IST-2001-32446 (CASENET).

References

1. C. Meadows. Open issues in formal methods for cryptographic protocol analysis. Proceedings of DISCEX 2000,pages 237-250. IEEE Comp. Society Press, 2000.
2. M. Burrows, M. Abadi, R. Needham. A logic of authentication. In Proceedings of the Royal Society, Series A, 426(1871):233-271, 1989.
3. R. Alur, T. Henzinger, F. Mang, S. Qadeer, S. Rajamani, S. Tasiran. Mocha: modularity in model checking. CAV 98 Computer-aided Verification, Lecture Notes in Computer Science 1427, pages 521-525. Springer-Verlag, 1998.
4. W. Marrero, E. Clarke, S. Jha. Model checking for security protocols. DIMACS Workshop on Design and Formal Verification of Security Protocols, 1997.
5. J.C. Mitchell, M. Mitchell, U. Stern. Automated analysis of cryptographic protocols using Murphi. In Proceedings of IEEE Symposium on Security and Privacy, pages 141-151. IEEE Computer Society Press, 1997.
6. L. Paulson. The inductive approach to verifying cryptographic protocols. Journal of Computer Security, 6, 1998.
7. J. López, J.J. Ortega, J.M. Troya. Protocol Engineering Applied to Formal Analysis of Security Systems. Infrasec'02, LNCS 2437, Bristol, UK, October 2002.
8. J. López, J.J. Ortega, J.M. Troya. Verification of authentication protocols using SDL-Method. Workshop of Information Security, Ciudad-Real- SPAIN, April 2002.
9. ITU-T Recommendation Z.100 (11/99). Specification and Description Language (SDL), Geneva, 1999.
10. D. Dolev, A. Yao. On the security of public key protocols. IEEE Transactions on Information Theory, IT-29:198-208, 1983.
11. ITU-T, Recommendation Z.120(10/96). Message Sequence Charts (MSC). Geneva, 1996.
12. A. Menezes, P.C. Van Oorschot, S. Vanstone. Handbook of Applied Cryptography. CRC Press, 1996.
13. CCITT Recommendation X.800. Security Architecture for Open Systems Interconnection for CCITT Applications. 1991.
14. ITU-T Recommendation X.810 (ISO/IEC 10181-1). Information Technology – Open Systems Interconnection – Security Frameworks for Open Systems – Overview. 1995.
15. ITU-T Recommendation X.811 (ISO/IEC 10181-2). Information Technology – Open Systems Interconnection – Security Frameworks for Open Systems – Authentication. 1995.
16. ITU-T Recommendation X.812 (ISO/IEC 10181-3). Information Technology – Open Systems Interconnection – Security Frameworks for Open Systems – Access Control. 1995.
17. ITU-T Recommendation X.813 (ISO/IEC 10181-4). Information Technology – Open Systems Interconnection – Security Frameworks for Open Systems – Non-Repudiation. 1995.

18. ITU-T Recommendation X.814 (ISO/IEC 10181-5). Information Technology – Open Systems Interconnection – Security Frameworks for Open Systems – Confidentiality. 1995.
19. ITU-T Recommendation X.815 (ISO/IEC 10181-6). Information Technology – Open Systems Interconnection – Security Frameworks for Open Systems – Integrity.1995.
20. P. Ryan, S. Schneider. The Modelling and Analysis of Security Protocols: the CSP Approach. Addison-Wesley, 2001
21. ITU-T, Recommendation Z.120 (11/99). Message Sequence Charts (MSC-2000). Geneva,1999.
22. RSA Laboratory. Public-Key Cryptography Standards (PKCS), http://www.rsa.com/.
23. G. Lowe. Towards a Completeness Result for Model Checking of Security Protocols. 11th IEEE Computer Security Foundations Workshop, pages 96-105. IEEE Computer Society, 1998.
24. M. Rusinowich, M. Turuani. Protocol Insecurity with Finite Number of Sessions is NP-complete. 14th IEEE Computer Security Foundations Workshop June 11-13, 2001
25. Solinet GmbH. SAFIRE product, http://www.solinet.com/
26. S.M. Bellovin, M. Merrit. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In Proceedings of IEEE Symposium on Research in Security and Privacy, pages 72-84, 1992.

Development of Distributed Systems with SDL by Means of Formalized APIs*

Philipp Schaible and Reinhard Gotzhein

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

Abstract. Due to their intrinsic complexity, the development of distributed systems is difficult in general and therefore relies on careful and systematic development steps. This paper addresses the design and implementation of distributed systems, using SDL as the design language. In particular, the refinements during implementation design are examined, and it is shown how SDL interfacing patterns can support these steps, even in a heterogeneous environment. Then, tool support to automatically implement the interfacing patterns by generating tailored APIs for the system environment is presented. Finally, these technologies are illustrated in the context of a comprehensive development of a distributed light control system in a heterogeneous environment, using various communication technologies.

1 Introduction

The development of distributed systems in heterogeneous environments is a difficult issue - despite the use of customized design languages, development methods, and tool support in this area. One reason certainly is the intrinsic complexity of these systems due to concurrency, synchronization, and cooperation of system agents. Especially in cases of large systems, this requires suitable structuring mechanisms as well as a careful and systematic system design.

For the *functional design* (the design covering the overall functionality of a distributed system) SDL [1] is a suitable specification language that is widely used in industry. SDL supports the hierarchical structuring of a complex distributed system into agent modules. Furthermore, the interaction behavior as well as the internal behavior of these modules can be specified. For closed SDL systems, implementation code can be generated automatically, which has positive effects on quality, development costs, and time-to-market.

When it comes to *implementation design*, where, for instance, direct interaction of agents is replaced by message exchange through an underlying communication service, an SDL system may have to be partitioned into several

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

subsystems. Usually, such a communication service is provided by a local operating system, and thus is part of the environment from the view point of the SDL subsystems. This means that to interact, agents now send/receive messages to/from the environment.

To implement open SDL systems (systems interacting with their environment) only part of the code is generated automatically with the existing tools. In addition, an environment interface – also called *environment functions* – has to be supplied, requiring manual coding steps. On the one hand, this environment interface depends on the underlying communication service. On the other hand, it depends on the SDL subsystems using the interface. Thus, replacement of the underlying communication service or changes of the interaction behavior of SDL agents entail changes of the environment interface, which is a time consuming and error-prone task.
Our strategy to address this particular problem is twofold:

1. First, we define generic design solutions for the interaction of an open SDL system with different underlying communication services. This is done by defining, for each communication technology, an interfacing pattern, using the pattern description template and notation of the SDL pattern approach [2–6]. Interfacing patterns can then be applied during the implementation design, where the decision to use a particular technology is made. In this way, the interaction behavior of SDL agents can be controlled, reducing the need for modifications of the environment interface.
2. Second, we conceive and implement tool support for the automatic generation of the environment interface. This tool support is syntactically and semantically integrated with the interfacing patterns, and currently supports interfacing with TCP and UDP sockets, CAN, UART/TP, and QNX IPC.

In [7], two solutions for environment interfaces are described:

> The first solution is based on so-called *light-weight APIs* and datagram sockets. The idea is to emulate the datagram sockets by representing them as SDL abstract data types with suitable SDL operators (Socket, SendTo, RecvFrom, Close). To send a signal, it is first encoded into a character string, and then sent by evaluating the expression "SendTo (<parameterList>)". This strategy is straightforward, as there is a one-to-one relationship between SDL actions and the service primitives of datagram sockets. However, it differs from the SDL communication paradigm that is based on explicit signal exchange.
> The second solution is based on so-called *full-weight APIs* and again datagram sockets. The communication is based on specific SDL signals outPacket, bindPort, and inPacket. To send a signal, it has to be encoded into a character string, and is then sent by an explicit SDL output action, as one parameter of outPacket.

Both solutions have the disadvantage that the SDL designer has to specify the coding and decoding of SDL signals and signal parameters in the design. Another drawback is that there exists only one set of operations for different SDL signals,

therefore, the receiver has to decode the packets first in order to distinguish between these signals. Finally, only datagram sockets are currently supported.

The paper is structured as follows. In Sect. 2, we elaborate on the systematic design of distributed systems, illustrated by a running example. In particular, we explain the refinements during implementation design, and show how these steps can be supported by SDL interfacing patterns. In Sect. 3, we introduce the tool APIgen for the automatic generation of environment interfaces. We show how the tool complements existing code generators, and explain the generated code. Section 4 presents a survey of the comprehensive development of a distributed light control system in a heterogeneous environment, where various interfacing patterns have been applied in the design phase, and APIgen has been used to generate the environment functions. Conclusions are drawn in Sect. 5.

2 Systematic Design of Distributed Systems with SDL

2.1 Stepwise Design

The design of distributed systems is often done in several steps, especially in cases of large systems. This requires that the system requirements be partitioned into subsets that can be dealt with one-by-one.

Horizontally The separation of system functionalities can lead to a proper partitioning, such that with each requirement subset, more functionality is added. For instance, phases of a communication service (connection setup, data transfer ...) or functionalities of a communication protocol (flow control, error control ...) can be identified.

Vertically The requirements can be partitioned into different levels of abstraction. For instance, on a high level of abstraction, direct reliable interaction between groups of system agents is assumed, while this assumption is later relaxed to unreliable, indirect interaction between pairs of system agents. This of course may influence the behavior of the system agents, which now may have to deal with loss or group management.

For the remainder of the paper, it is sufficient to consider *vertical partitioning*, by distinguishing two levels of abstraction:

- *Functional design* deals with the overall system functionality — a high level of abstraction.
- *Implementation design* addresses the mechanisms used to implement the system, in particular, the replacement of direct interaction between system agents by concrete communication services.

To illustrate these steps, we start with a simple example: the system pingPong. Figure 1 shows a *functional design*, where two agents called pingAgent and pongAgent interact via a common channel pingPongTable, directly exchanging signals ping and pong, each carrying a parameter of type Integer.

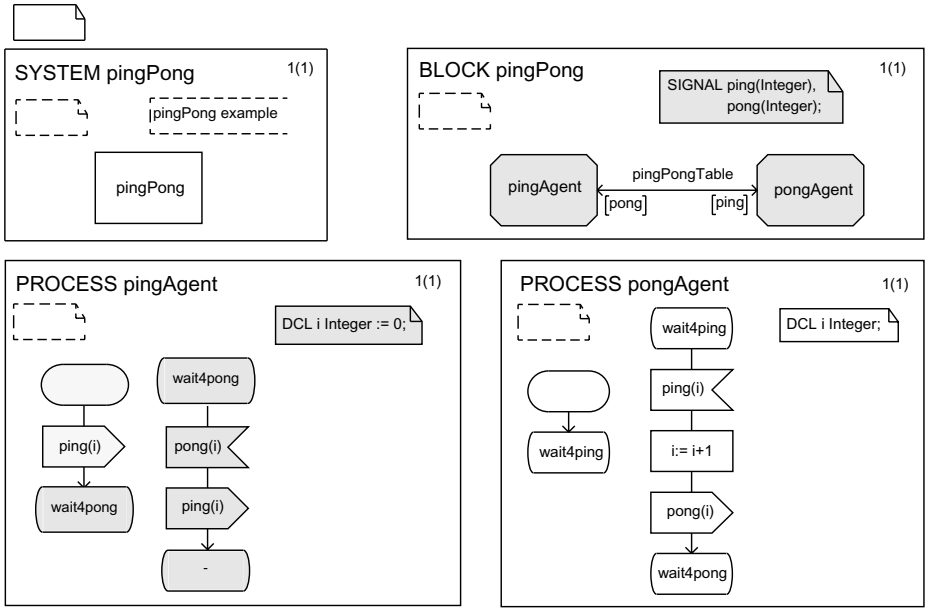

Fig. 1. Example "pingPong": Functional Design

In a distributed environment, it is intended that the agents of the system pingPong be placed on different hosts. Therefore, the direct interaction of the functional design has to be replaced by an underlying communication service. This leads to an *implementation design*, where this underlying service is made explicit, and the behavior of the interacting agents is modified such that the functional design is realized correctly.

Figure 2 shows an *implementation design* that is based on the decision to use the communication service provided by the transport protocol TCP. For this purpose, an SDL component called TCPserviceProvider is added, and interaction between pingAgent and pongAgent, now acting as service users, is redirected via this process. As TCP supplies specific service primitives, the behavior of the service users requires modification. For instance, to send a signal ping, a socket has to be created, and a connection must be established. Furthermore, the signal ping and its parameter have to be encoded before they can be sent, and to be decoded upon reception. Figure 2 shows the additional behavior of pingAgent resulting from these design decisions. The correspondence between Figs. 1 and 2 is highlighted by the shaded SDL symbols.

2.2 Pattern-based Implementation Design

Analysis of several implementation designs has shown similarities in those parts where common channels have been replaced by an explicit service provider. To capture these similarities, we have defined *generic solutions* using the *SDL pat-*

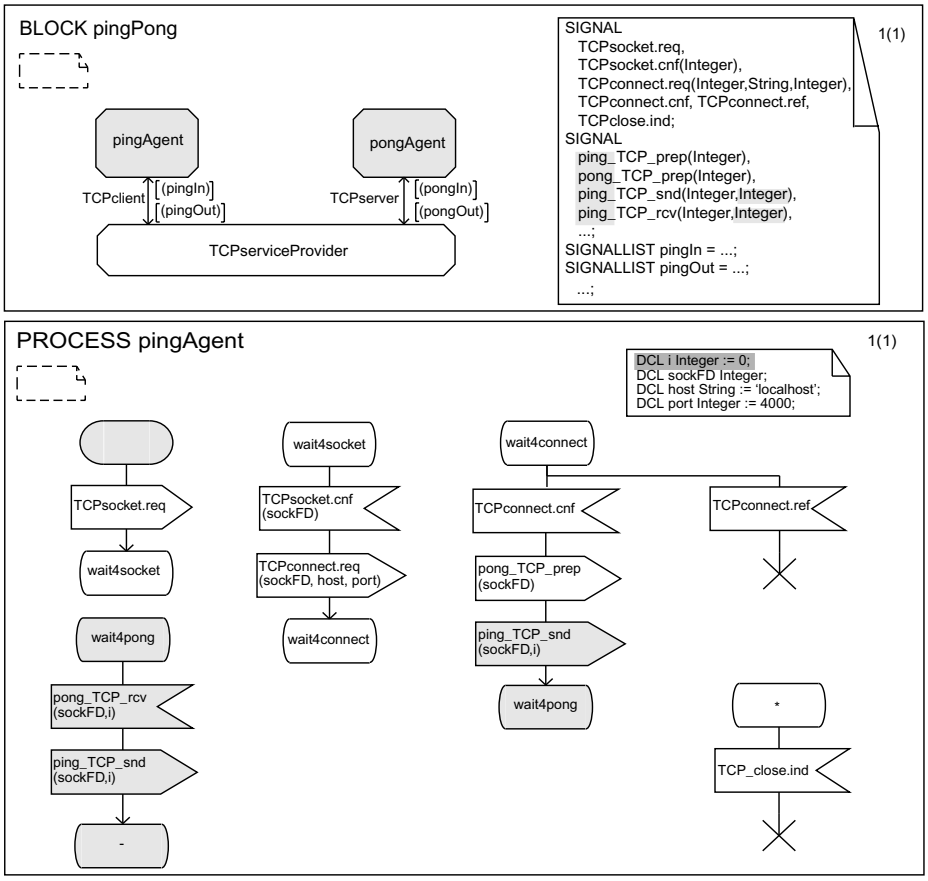

Fig. 2. Example "pingPong": implementation design

tern approach [2, 6, 8]. In particular, we have defined, for each type of communication service, an *SDL interfacing pattern*. Currently, interfacing patterns exist for the services provided by TCP, UDP, CAN, Bluetooth, UART/TP, AAL5, and QNX interprocess communication. Furthermore, SDL components representing these services have been defined in order to have complete implementation designs that can be simulated.

In Fig. 3, we show an excerpt of the generic solution defined by the SDL pattern TcpInterfacing. The excerpt is taken from the *SDL Fragment*, the syntactical part of the design solution defined by the pattern, and shows the context, the adaptation, and the embedding for a TCP client (EFSM TCPclientAutomatonA) and the enclosing scope unit (SU TCP). To define the generic design solution, a language called PA-SDL (Pattern Annotated SDL, see [6] for details) is used. Solid symbols denote design elements that are added to the context specification as a result of the pattern application. As a general rule, names may be

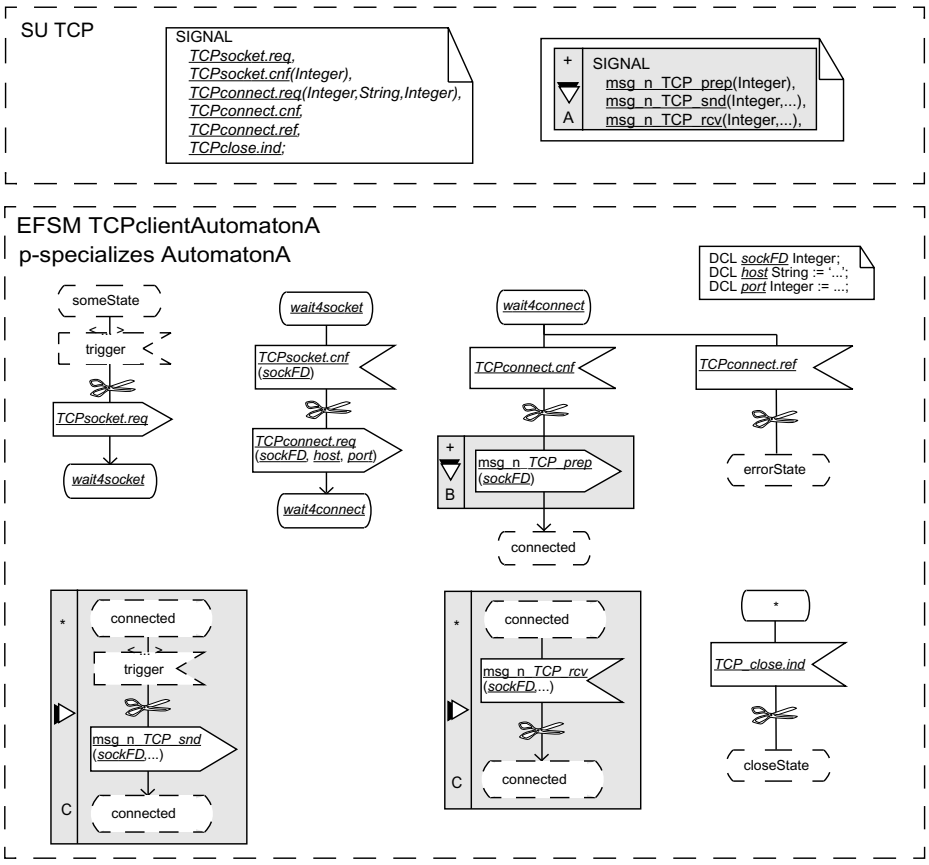

Fig. 3. TcpInterfacing pattern (SDL Fragment, excerpt)

changed. However, names in italics must be fresh, and if underlined, renamed in a unique way when adapting the pattern. SU refers to a structural SDL unit, for instance, a system, a block, a process, or a service. Scissor symbols indicate the possibility of refinements, for instance, by adding further actions to a transition, without disrupting the control flow. Finally, the shaded part called *border symbol* is an annotation denoting replications. The direction of replication (horizontal or vertical) is given by the arrow, the number of replications is specified by the multiplicity.

To apply a pattern, the context has to be identified first. In case of the TcpInterfacing pattern a choice must be made of the enclosing structural unit, two active components (SDL processes), and matching transitions of these components. The SDL fragment then defines how to adapt the pattern, and how to embed it into the SDL context specification. Application of the TcpInterfacing pattern to the functional design in Fig. 1 yields the implementation

design in Fig. 2. In a similar way, interfacing with other types of communication services can be achieved.

3 Automatic Implementation of SDL Designs with APIgen

In this section, we introduce the tool *APIgen* (API generator), which supports the automatic code generation for distributed systems designed in SDL. APIgen is a supplement for *Cadvanced*, the SDL-to-C compiler that is part of the Telelogic TAU SDL suite [9]. In its present form, APIgen is syntactically and semantically integrated with the interfacing patterns of the SDL pattern pool (see Sect. 2.2). Starting point for the code generation is an implementation design, as shown in Sect. 2.1.

3.1 Architecture of APIgen

To run a system in a distributed environment, several implementation decisions have to be made. For instance, the target environment is determined, and the logical distribution given by the implementation design is mapped to physical components. In fact, this is also a decision between light and tight integration, in the sense that SDL processes are mapped to different OS processes or a single one. This may lead to a modification of the implementation design, such that components to be implemented on the same physical node are collected into one SDL system that is syntactically complete and therefore can be compiled.

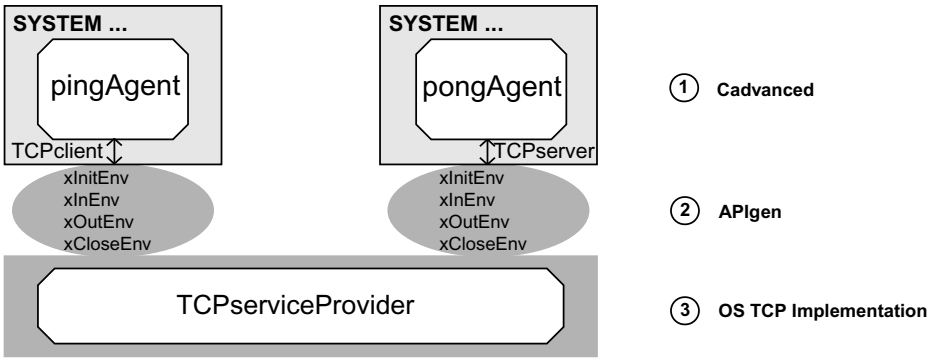

Fig. 4. Example "pingPong": light integration

Figure 4 shows the result of these implementation decisions for the pingPong system: pingAgent and pongAgent are assigned to separate nodes, therefore, separate SDL systems that interact with their local environment are introduced, each containing the corresponding declarations and process specifications. The

resulting SDL systems are open in the sense that they interact with their environment. Here, interaction is by means of signals. Furthermore, it is decided to replace TCPserviceProvider by an OS TCP implementation.

From the implementation decisions, it follows that, from the view point of pingAgent and pongAgent, the communication service now belongs to the *environment*. This is important when it comes to automatic code generation: it is straightforward to generate C-code from SDL specifications, using *Cadvanced*, the SDL-to-C compiler [9]. However, interaction with the environment is not directly supported. For interaction between pingAgent and pongAgent with the OS TCP implementation, for instance, a set of tailored environment functions has to be provided: xInitEnv and xCloseEnv are called at system start and termination, respectively; xOutEnv is called when a signal is sent to the environment, and xInEnv is called periodically, polling for events in the environment leading to signals to be sent to the SDL system.

Conceptually, the environment functions can be understood as an API. They abstract from internal details, and serve as an interface to be used by SDL process implementations whenever interaction with the environment is necessary. Because the code of this API depends on both the environment (the type of communication service and its implementation) and the SDL system (the signal types to be exchanged with the environment, and, in particular, the signal parameters), it is commonly hand-coded. In our experience, this is a very time consuming and error prone task, due to several fundamental and conceptual differences between the physical environment representing a "real" world and abstract SDL specifications.

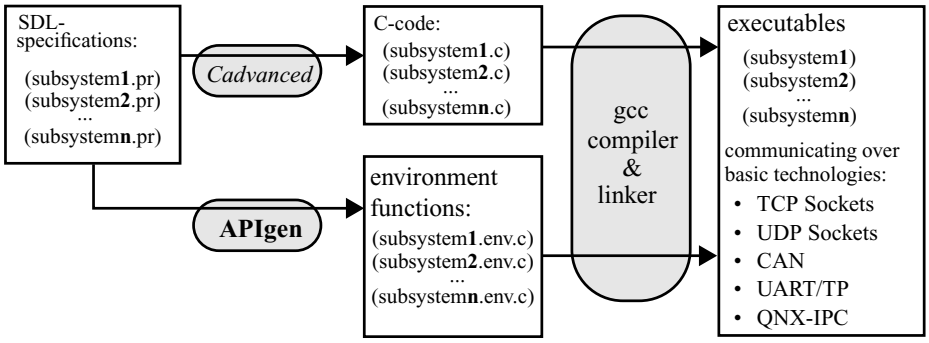

Fig. 5. Tool chain for automatic code generation

In the pingPong example, both SDL systems are compiled using Cadvanced, generating C-code that assumes the existence of this API. Furthermore, the TCP implementation is part of the operating system. To fully automate the code generation, we have conceived and implemented *APIgen*, a tool to create the environment functions (see Figure 5). APIgen takes the SDL specifications as input, and automatically generates C-code, supporting a variety of communi-

cation technologies (e.g., TCP sockets, UART/TP, CAN, QNX IPC). This fills the gap between code generation performed by Cadvanced and existing communication technologies, without requiring subsequent manual modifications or additions. Generated environment functions are stored in separate files named subsystem**n**.env.c (see Fig. 5). They can be compiled separately and are linked with core modules subsystem**n**.c generated by Cadvanced.

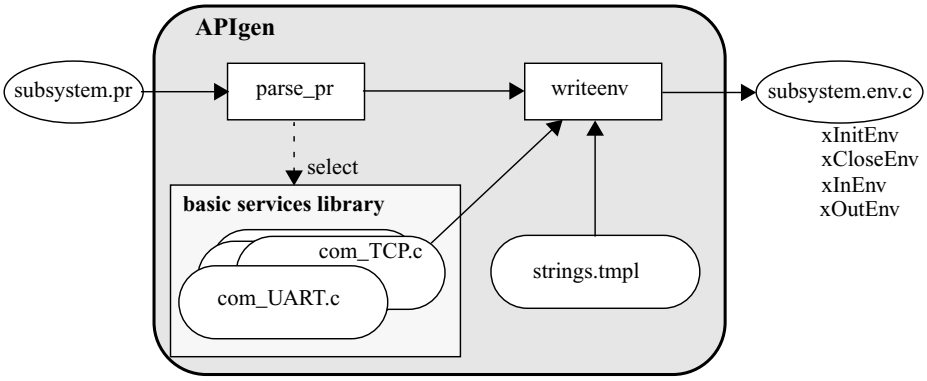

Fig. 6. Architecture of APIgen

When APIgen is started, there is no further interaction with the system developer. All communication specific information (such as selection of communication technology, host addresses, port numbers) is contained in the SDL implementation design specification, and is obtained from interfacing pattern applications (see Sect. 2.2). More specifically, APIgen takes SDL pr-files as input, and outputs env.c-files (see Fig. 6). Pr-files are processed in order to identify and collect signals sent to the environment. Furthermore, it is determined what basic technology is used to exchange these signals with other SDL systems (module *parse_pr* in Fig. 6). For this purpose, specific naming conventions have been defined. For instance, if an original signal sig is to be sent, then the corresponding signal to be sent via TCP sockets is to be named sig_TCP_snd (see Fig. 2). Please note that these naming conventions are enforced when applying the TCPInterfacing pattern (see Fig. 3).

The actual generation of environment functions is performed by the module *writeenv*, which uses general purpose strings from the library *strings.tmpl* and basic technology specific strings from the selected module of *basic services library* (see Fig. 6). Arranging technology specific strings into separate modules supports the extension of APIgen to incorporate further basic technologies, as well as its maintainability.

3.2 Auxiliary signals

Before signals between processes of different open SDL systems can be exchanged, the underlying basic technology may have to be configured. For instance, a connection setup may be required, and messages to be exchanged may have to be registered. For these purposes, we introduce a specific set of SDL signals called *auxiliary signals*. These signals have predefined names and parameters, and must be sent to the environment prior to message exchange in accordance with the underlying service.

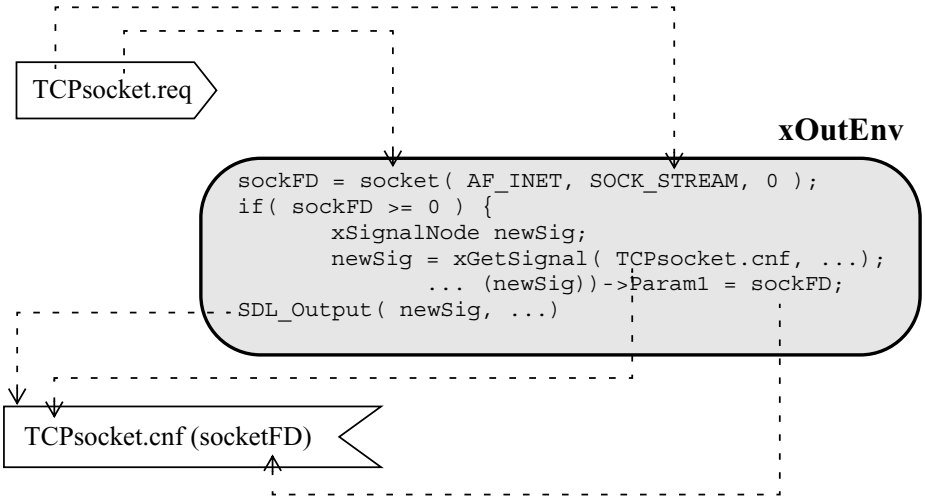

Fig. 7. API generation for auxiliary signals (example)

In Fig. 2, pingAgent first requests a socket, using SDL signals TCPsocket.req and TCPsocket.cnf, and then establishes a connection, using further signals TCPconnection.req, TCPconnection.cnf, and TCPconnection.ref. These are auxiliary signals following obvious naming conventions that are known to APIgen. Based on these naming conventions, APIgen will generate, for each signal, suitable environment functions xInEnv and xOutEnv. An example is shown in Fig. 7: an output of TCPsocket.req triggers the creation of a stream socket, where the file descriptor sockFD is returned as result.

The return of a result causes a problem in the context of SDL, because the output of a signal does not yield a return value. This is due to different interaction paradigms: in SDL, interactions are asynchronous notifications, while C-procedure calls are based on synchronous inquiry. To solve this problem, we model synchronous inquiry in SDL: after requesting a socket by output of a TCPsocket.req signal, the SDL process enters a waiting state until the result is returned. This result is received as a parameter of the predefined signal TCPsocket.cnf (see Fig. 2). To ensure that both naming and behavior conventions

are followed in the SDL design, we have defined the TCPINTERFACING pattern (see Sect. 2). Please note that by using these conventions, it is straightforward to create multiple sockets, and to distinguish them by their file descriptors.

Figure 7 shows an excerpt of the API code that will be generated for the pingPong example. As the result will be returned upon completion of the socket request, the library function SDL_Output is called to return the TCPsocket.cnf signal to the SDL process. Usually, SDL_Output would be called in the environment function xInEnv, as this is an input to the SDL process. However, for efficiency reasons, it has been incorporated into xOutEnv.

3.3 Transfer Signals

Once the underlying basic technology has been configured, SDL signals can be exchanged between remote SDL processes. For this purpose, a specific set of SDL signals called *transfer signals* is derived from the original signals: that is, those signals that had been directly exchanged between SDL processes of the functional design (see Fig. 1). For instance, for an original signal ping, transfer signals ping_TCP_snd and ping_TCP_rcv are added to the signal definitions of the sending and the receiving process, respectively. The suffix distinguishes whether a signal is sent or received. The use of these signals in the SDL implementation design is shown in Fig. 2 and is in fact quite obvious.

To establish the interfacing between SDL processes and TCP sockets, several strategies are possible. For instance, a signal TCP_snd, parameterized with the file descriptor of the socket and a byte sequence containing the signal type and the signal parameter values, could be used for sending different SDL signals (see [7], full-weight API). This approach requires that coding and decoding of SDL signal type and parameter values is performed in the SDL implementation design. Without appropriate tool support, for instance, by Cadvanced, this strategy requires a very detailed design specification.

Alternatively, coding and decoding could be shifted into the application programming interface. We have adopted this strategy for two reasons. First, it relieves the SDL implementation designer of the tedious and error-prone details. Second, the routines for coding and decoding of signals can be and in fact is automatically generated by the tool APIgen. During the parsing phase, APIgen extracts all information necessary for this purpose from the input file subsystem**n**.pr. In particular, the naming conventions support the identification and collection of signals to the environment, and the assignment of the corresponding basic technology. We point out that different basic technologies may be used at the same time, therefore, this strategy also supports the development of large systems in heterogeneous environments. Furthermore, the naming conventions are again supported by interfacing patterns, e.g., TCPINTERFACING.

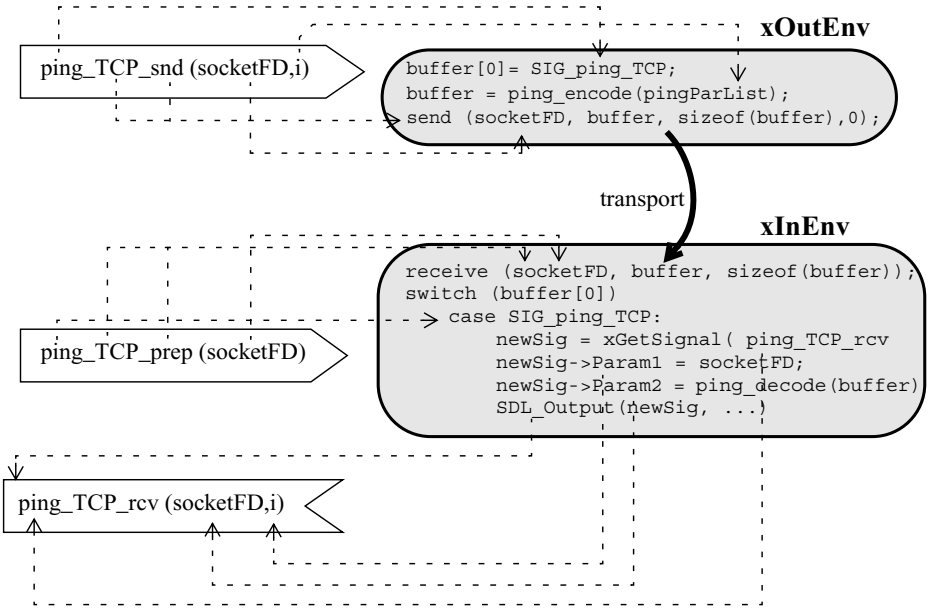

Fig. 8. API generation for transfer signals (example)

Figure 8 shows another excerpt[1] of the API code that will be generated for the pingPong example. In this example, an output of a signal ping_TCP_snd is mapped to the corresponding part of the environment function xOutEnv, and sent via an existing TCP connection. This TCP connection has already been established during the configuration of the underlying service, using auxiliary signals (see Section 3.2). The receiver prepares the reception of signals ping_TCP_rcv by first calling ping_TCP_prep, thus indicating where signals of this type should be delivered. Afterwards, polling for actual receptions is done (environment function xInEnv). To support signal parameters, APIgen generated coding and decoding procedures, to be used by xOutEnv and xInEnv, respectively (ping_encode, ping_decode, see Fig. 8).

Please note that this is an excerpt of the code generated as part of xOutEnv. For each SDL signal sent via the environment interface, approximately 1 page of C-code is added to this function.

Some implementation details are worth mentioning:

– To generate environment functions (pingAgent.pr and pongAgent.pr), *APIgen* is started with a complete list of communicating SDL systems. This way, corresponding _snd and _rcv signals, (such as ping_TCP_snd and ping_TCP_rcv) can be associated and obtain a unique, system wide signal identification

[1] For each SDL signal, approximately 1 page of C-code is generated and added to the environment functions. In addition, functions for encoding and decoding of signal parameters may be generated, depending on whether ASN.1 is used to define these parameters.

(such as SIG_ping_TCP, see Fig. 8) as well as matching coding and decoding procedures (ping_encode and ping_decode in Fig. 8) to be incorporated into xInEnv and xOutEnv as shown.
- Coding and decoding of signal parameter values is supported in different ways:
  - ⋆ For certain basic SDL data types and some composite data types (including structures and arrays), APIgen automatically generates coding and decoding routines. The standard strategy is to encode all parameter values of a signal into a string.
  - ⋆ For ASN.1 data types, coding and decoding routines are automatically generated with the ASN.1 utilities provided with the Telelogic TAU tool suite. Both basic and packed encoding rules (BER and PER) are supported. These routines are then linked with the code generated by APIgen.
  - ⋆ When developing heterogeneous systems, it may be necessary to use specific routines for coding and decoding of signals. These routines can not be generated by APIgen. However, APIgen supports the system developer by creating template files (signaldefs.h and signaldefs.c) for coding and decoding routines, which are then completed by the system developer.

Development of *APIgen* started in 1998. After a first prototype with restricted functionality (few basic technologies, restricted use of signal parameters), it has been continuously improved and extended, and has been used in several case studies. The most comprehensive case study has been SILICON, where a distributed application in a heterogeneous environment has been developed from scratch. In the next section addresses this case study — especially the use of SDL interfacing patterns and APIgen.

4 The SILICON case study

In 1999, the problem description of a distributed light control system [10] was sent out with an international call for papers, soliciting the application of requirements engineering methods, techniques, and tools to this case study. The results have been published in [11]. We have taken a subset of this problem description as the starting point for another case study called SILICON (System development for an Interactive LIght CONtrol), covering not only the requirements phase, but all development activities. In particular, the objective of this case study has been to develop a complete, customized solution starting from an informal problem description, applying generic methods in all development phases.

The problem description of the SILICON case study consists of the building description and the informal needs from the view points of the user and the facility manager. Fig. 11 is an excerpt of the ground-plan, showing two offices and a hallway section as well as a number of installations. Each office is equipped with two light groups and switches to turn the lights on and off. The windows

can be darkened by sun blinds that are controlled by further switches. Another light group is placed in the hallway section, it can be switched on and off. In addition, the light in the hallway is triggered by a motion detector.

In the following, we will focus on the design and implementation phases, and, in particular, on the interfacing with basic communication technologies at design and implementation level. According to the design steps explained in Sect. 2, we start with the functional design, which deals with the overall system functionality on a high level of abstraction. Nevertheless, we introduce a high-level system structure by identifying application components. For instance, we distinguish sensors, actuators, and control cells. Control cells are structured hierarchically, starting with the building level down to the room level, they receive sensor values and trigger actuators. This leads to a functional design with a logical system structure that follows the physical building structure.

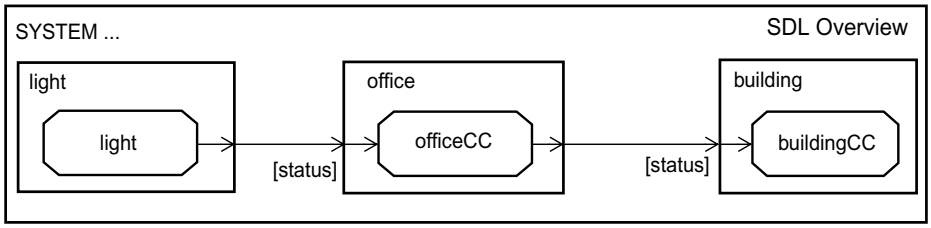

Fig. 9. SDL functional design (excerpt): system architecture (SDL overview diagram)

In Fig. 9, an excerpt of the logical system structure as specified in SDL is shown. It contains three application components, namely a light sensor light, an office control cell officeCC, and a building control cell buildingCC. On the functional design level, these components interact directly through SDL channels. An interaction occurs, for instance, when the light sensor sends a signal status, which is received by officeCC and then forwarded to buildingCC. The status signal may then trigger further signals (not shown here), for instance, to display the light status on the facility manager control panel, or to trigger the light actuator.

In the implementation design, the channels between application components have been replaced by underlying communication services. In the SILICON case study, it was decided to group sensors, actuators, and control cells hierarchically, and, depending of the required throughput and real-time performance, to use different communication technologies on each level of the hierarchy. For instance, it was decided to use UART/TP and CAN to interconnect components of one room and one floor, respectively.

Figure 10 indicates how these design decisions have been incorporated into the SDL implementation design. While keeping the application components unchanged, further components representing the communication middleware (switchPE, UART_Codex_light ...) and the basic communication technologies (UARTservice Provider, CANserviceProvider) as well as the necessary channels are added, refining the functional design without modifying the behavior of the application components. In contrast to the example in Section 2, the communi-

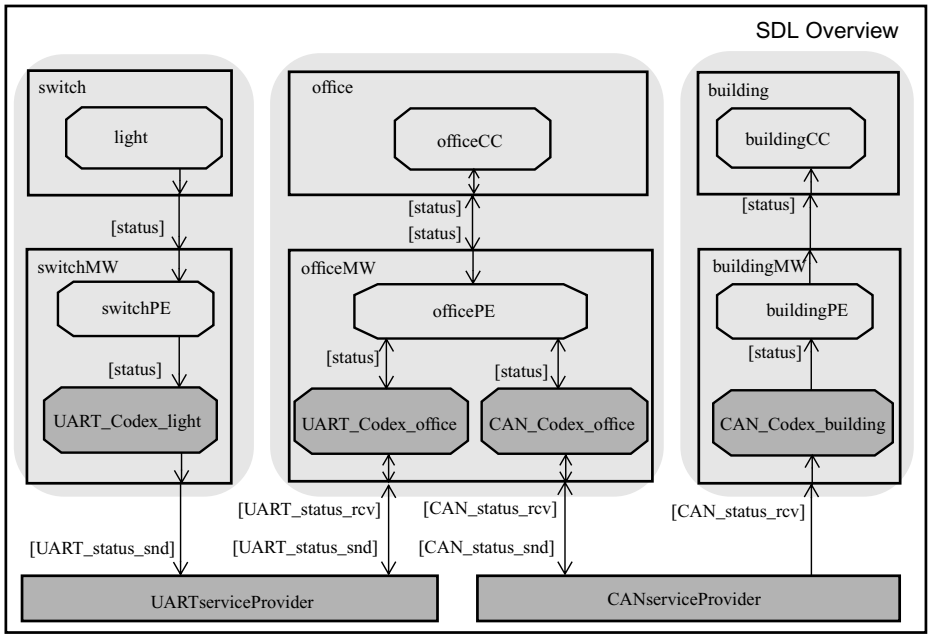

Fig. 10. SDL implementation design (excerpt): refined system architecture

cation middleware (switchMW, officeMW, buildingMW) is made explicit and not incorporated into the application components, in order to enclose all communication specific functionality. Otherwise, the approach is just the same: select an underlying service provider and apply interfacing patterns to replace direct communication. In the case study, we have applied several such patterns, including UARTInterfacing, CANInterfacing, TCPInterfacing, and Bluetooth-Interfacing.

Based on the implementation design, we have developed a complete, customized implementation, consisting of a physical building model, a new tailored communication technology, application hardware, communication middleware, and application software. The implementation of the application components and the communication middleware has been produced as described in Sect. 3, using Cadvanced and APIgen. This provides evidence that the approach scales, and that it works well also in the context of heterogeneous communication systems. Figure 11 shows the building model topology laid out around the ground-plan:

- On the room level, the UART/TP-bus technology (Universal Asynchronous Receiver Transmitter/Token Passing) provides interconnection between micro controllers implementing switches, light groups, sun blinds, motion detectors, and user control panel, and embedded PCs implementing office and hallway controllers. We have installed 3 separate UART-buses, each associated with the devices of a single room.
- On the floor level, the CAN-bus technology (Controller Area Network) interconnects embedded PCs that act as office or hallway controllers.

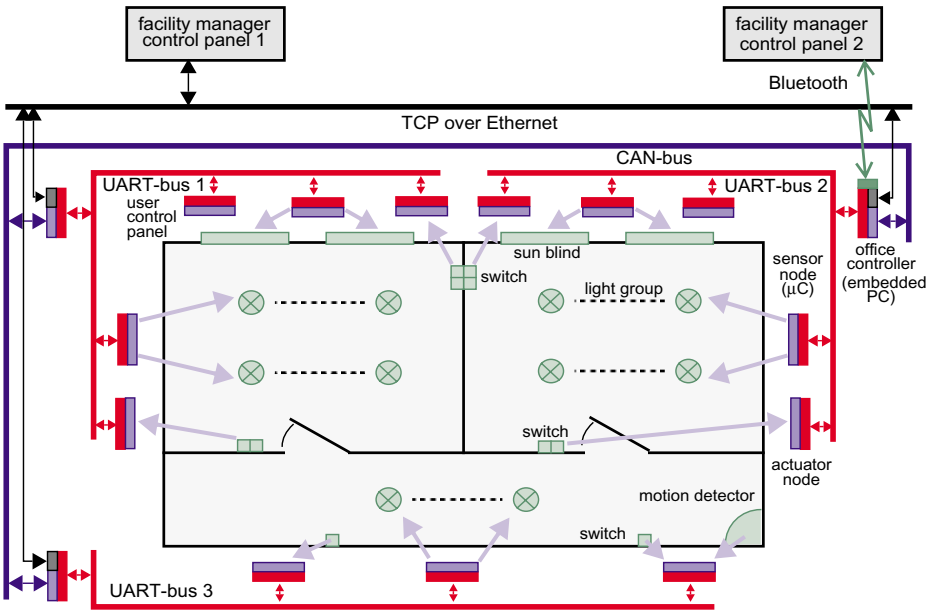

Fig. 11. Topology of the building model implementation

– On the building level, Ethernet technology interconnects one or more embedded PCs of each floor. In this case study, this option is not fully exploited, however, we use this technology to download executables, and to visualize the communication system in operation. Furthermore, the facility manager control panel can be attached to this technology.
– In addition, Bluetooth technology provides wireless interconnection between the facility manager control panel and office controllers.

The hierarchical structure of the communication system in combination with field bus technologies requires special purpose protocols with routing support, as TCP/IP-solutions are not feasible here. Furthermore, these protocols have to support real-time communication.

The physical building model is shown in Fig. 12. It has been demonstrated at several industrial fairs including Embedded Systems 2001 in Nuremberg, and CeBIT 2001 in Hannover, in cooperation with Telelogic. In the front part of the building model, the windows equipped with sun blinds are visible. Below, micro controllers implementing the switches, sun blinds, and light groups are installed. User control panels are mounted in front of the building, and can be used to control each light group and sun blind separately. Furthermore, the panel can be used to set and recall one or more light scenarios. The embedded PCs are hidden under the building. To the left of the building model, two computer screens are placed. The front screen belongs to a laptop that acts as the facility manager control panel, and is used for control purposes as well as for visualization on the application level. The flat screen above provides the visualization on the communication level.

Fig. 12. SILICON building model

5 Conclusions

We have presented a methodology for the development of distributed systems through refinement of the functional design supported by SDL interfacing patterns. Interfacing patterns define generic solutions for recurring design problems, as do SDL patterns in general. In accordance with the underlying communication service, interface patterns are selected from the pattern pool, adapted, and embedded into the context specification. This is particularly valuable, as the quality of the design is improved, leading to a significant reduction of rework due to defects. Also, incorporating further basic technologies is straightforward.

In order to generate code from implementation designs, we have developed the tool APIgen, which complements existing SDL-to-C code generators. APIgen automatically generates environment interfaces for a variety of communication technologies, without the need of further user interaction, and is syntactically and semantically integrated with the corresponding SDL interfacing patterns. This way, several basic technologies can be used together, without the need for manual coding. This closes a gap in the development of distributed systems.

The methodology presented here is intended to support the communication system developer, who needs to preserve some control over the basic communication technology, even at the design level. For instance, the ability to make a proper selection between different technologies such as CAN, UART, or TCP, and to control their configuration (setting host addresses and port numbers) are crucial. Furthermore, when a particular selection, say TCP, has been made, the designer can control at which point in execution, connections are established and closed. This is different from other methodologies (such as the normal approach

using Telelogic TAU) where interfacing with the environment is transparent on design level.

Currently, APIgen supports SDL-96, a pragmatic decision based on the available commercial tool support. It would be desirable to use the additional language features of SDL-2000, in particular, hierarchical states and exception handling. With these features, further abstractions can be introduced into design specifications, for example encapsulating generic design decisions captured by SDL patterns in general or SDL interfacing patterns in particular.

SDL interfacing patterns may be interpreted as design-level APIs, to be used during SDL implementation design. This way, the generic solutions captured by interfacing patterns may eventually lead to *standardized* design-level APIs, enabling interoperability. Furthermore, based on standardized APIs, tools for the automatic generation of program-level APIs (environment functions in the context of SDL implementations) can be developed. In fact, APIgen has been conceived and implemented for precisely this purpose.

References

1. ITU-T. Recommendation Z.100 (08/02), Specification and Description Language (SDL). International Telecommunication Union, Geneva.
2. B. Geppert, R. Gotzhein, F. Rößler. Configuring Communication Protocols Using SDL Patterns. SDL'97 - Time for Testing, Proceedings of the 8th SDL Forum, Elsevier, Amsterdam, 1997, pp. 523-538.
3. D. Cisowski, B. Geppert, F. Rößler, M. Schwaiger. Tool Support for SDL Patterns. Proceedings of the 1st Workshop on SDL and MSC (SAM'98), Berlin, 1998.
4. F. Rößler, B. Geppert, P. Schaible. Re-Engineering of the Internet Stream Protocol ST2+ with Formalized Design Patterns. Proceedings of the 5th International Conference on Software Reuse (ICSR5), Victoria, Canada, 1998.
5. B. Geppert, A. Kühlmeyer, F. Rößler, M. Schneider. SDL-Pattern based Development of a Communication Subsystem for CAN. Formal Description Techniques and Protocol Specification, Testing, and Verification, Proceedings of FORTE/PSTV'98, Kluwer Academic Publishers, Boston, 1998, pp. 197-212.
6. B. Geppert. The SDL-Pattern Approach - A Reuse-Driven SDL Methodology for Designing Communication Software Systems. Ph.D. Thesis, University of Kaiserslautern, 2000.
7. T. Kim, R.L. Probert, I. Sales, A. Williams. Rapid Development of Network Software via SDL/Socket Interfaces, Proceedings of the $3^r d$ Workshop on SDL and MSC (SAM'2002), Aberystwyth, Wales, LNCS 2599, Springer, 2003.
8. R. Gotzhein. Consolidating and Applying the SDL-Pattern Approach: A Detailed Case Study. Journal of Information and Software Technology, Elsevier Sciences, 2003 (in print).
9. Telelogic. Tau 4.4 SDL Suite, 2002.
10. S. Queins, G. Zimmermann, M. Becker, M. Kronenburg, C. Peper, R. Merz, J. Schäfer. The Light Control Case Study: Problem Description. Journal of Universal Computer Science (J.UCS), Special Issue on Requirements Engineering 6(7), pp. 586-596, Springer, 2000 .
11. E. Börger, R. Gotzhein (Guest Eds.): Requirements Engineering: The Light Control Case Study. Special Issue of the Journal of Universal Computer Science (J.UCS), Numbers 6(7), Springer, 2000.