On the use of Pragmatics for Model-based Development of Protocol Software

Kent Inge Fagerland Simonsen[1,2]

[1] Department of Computer Engineering, Bergen University College, Norway
kifs@hib.no
[2] DTU Informatics, Technical University of Denmark, Denmark
kisi@imm.dtu.dk

Abstract. Protocol software is important for much of the computer based infrastructure deployed today, and will remain so for the foreseeable future. With current modelling techniques for communication protocols, important properties are modelled and verified. However, most implementations are being done by hand even if good formal models exist. This paper discusses some of the challenges in modelling and automatically generating software for protocols. The challenges are discussed using the Kao-Chow authentication protocol as a running example by outlining an approach for generating protocol software for different platforms based upon Coloured Petri Nets (CPN). The basic idea of the approach is to annotate the CPN models with pragmatics which can be used in a code generator when mapping the constructs of the CPN model onto the target platform.

1 Introduction

Much work has been done to model and verify protocols using a wide range of formalisms [7]. Petri nets [22] and Coloured Petri Nets (CPNs) [12,13] in particular are widely used formal modelling languages for behavioural modelling and verification of industrial-sized protocols [6]. There exist, however, relatively few examples [15,16,20] where CPN models have been used as a basis for automatically obtaining implementations of the modelled protocols.

This paper describes challenges with automatically generating code from protocol models and proposes some avenues for solving them. A concept of *pragmatics* is introduced for protocol models which holds information useful for generating an implementation. This paper also proposes the use of separate models to describe the configuration and platform for protocol software. This allows the protocol models to be at a high level of abstraction while specific implementations can be derived using configuration and platform models.

Figure 1 illustrates our approach to generating protocol software. The Protocol Model is a model in a language that is not yet fully designed, but it could be based on CPN or another High Level Petri Net (HLPN) language. The Configuration Model contains information on which design choices should be made for the implementation of the protocol. For example, the configuration can contain information on exactly which underlying network layer service to use for communication between protocol entities. The Platform Model is intended to contain information on how operations should be implemented on the specific platform in question. For example the details on what is needed to set up and transmit messages over the User Datagram Protocol (UDP) [8] or the Transmission Control Protocol (TCP) [9]. The Protocol Model, together with a Configuration Model and Platform Model is fed into a Generator in order to obtain an implementation of the protocol. Finding a good separation between the Protocol, Configuration and Platform Models is one of the important challenges in this approach.

The evaluation of our approach will be based on the software we are able to generate using our approach. If we are able to generate software for a wide range of protocols with high quality, this will be considered a success. We will also evaluate the confidence we can gain that the generated software maintains the properties of the Protocol Model.

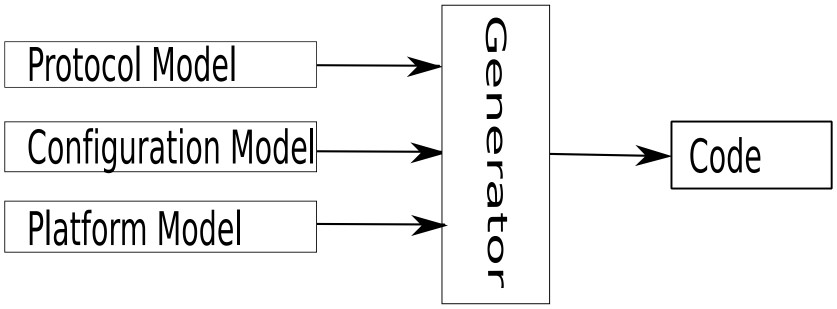

Fig. 1: Protocol software generation approach.

In order to include information that will help with code generation, we introduce the concepts of *pragmatics* and *scope* to the Protocol Model. Pragmatics assign special behaviour and meaning to model elements. This means that we are able to differentiate between transitions, places and data according to their function in the protocol. In the protocol models, pragmatics are encapsulated in << and >>. We will provide more details on these pragmatics in the following.

This paper is structured as follows. Section 2 focuses on elements that are missing from CPNs in order to model and generate code for protocols. This section also introduces the Kao-Chow (KC) authentication protocol [14] which is used as a running example throughout this paper. The concepts of pragmatics and scope are also introduced in this section. Section 3 discusses the need for configuration and platform models and identifies some elements that should be contained in those models. Finally, Section 4 discusses future work and identify criteria for evaluating our approach to model based development of protocol software. The reader is assumed to be familiar with the basic concepts of CPNs.

2 Protocol Model

To illustrate our approach we use the KC protocol. KC is a protocol that makes it possible for two entities, A and B, to authenticate each other using uncertified symmetric key [3] cryptography and an authentication server, S. The authentication server is assumed to have pre-shared keys with each of the authenticating entities. Listing 1 shows the basic sequence of messages exchanged in KC using Alice and Bob notation [19]. First some entity, A, wants to authenticate with another entity, B. To this end, A sends its and Bs identity together with a nonce[4], Na, to the authentication server, S (1). S then generates a session key, Kab, for use between A and B. This session key and A and B's identities, together with A's nonce is encrypted with the pre-shared key, Kbs, between B and S. S also creates a copy of the same data encrypted with the pre-shared key, Kas, between A and S and sends both copies to B (2). B then sends the part of the message it got from S encrypted with the key, Kas, shared by A and S to A together with As nonce encrypted with the session key, Kab, and a new nonce Nb (3). Finally, A responds to B with B's nonce, Nb, encrypted with the session key, Kab, (4). A considers B to be authenticated when the nonce, Na, it receives from A encrypted with Kab is identical to the Na which A created at the beginning of the exchange. Similarly, B considers A to be authenticated when B receives its nonce, Nb, encrypted with Kab from A.

Listing 1: Kao-Chow message sequece from [23]

```
1. A -> S: A, B, Na
2. S -> B: {A, B, Na, Kab}Kas, {A, B, Na, Kab}Kbs
3. B -> A: {A, B, Na, Kab}Kas, {Na}Kab, Nb
4. A -> B: {Nb}Kab
```

CPNs and other types of Petri Nets are widely used and have a well documented capability for modelling and verifying protocols and aiding in the implementation of protocol software. Our approach is to use HLPNs, and CPNs in particular, as a starting point for modelling protocols.

The top page of a CPN model of KC is shown in Fig. 2. Here the participants, A, B and Server, in the protocol are modelled as substitution transitions on the top level module. The places make it explicit that A send messages to Server, Server send messages to B, and that A and B send messages to each other.

Another effort to model KC using Petri Nets is presented in [2]. In this paper KC is first defined in the Security Protocol Language [5] and then translated into S-nets [3]. KC is also modelled using several different tools and languages in [4].

[3] Uncertified keys are keys that are not accompanied with information such as proof of who is the keys owner and issuer and how the key should be used.

[4] A nonce is a number or bit-string that is used only once.

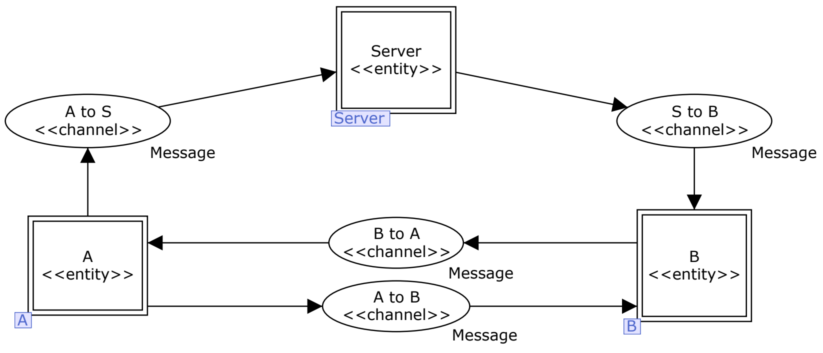

Fig. 2: Top level module of the Kao-Chow model.

2.1 Scope Pragmatics

Implicit in the KC model is information about different protocol entities that have different roles. Our approach is to make this information more explicit is to add the entity pragmatic to substitution transitions that indicate that the module is an entity in the protocol. In this approach, the top level of a model typically consists only of substitution transitions and network nodes, which is the case in the KC model in Fig. 2.

An alternate approach could be to tag all model elements that are part of the same protocol entity or to encompass all elements inside some form of field that delimits the entities from each other. One problem with this approach is that since several parties can exist on the same module some elements may interact without going through a network node. On one hand this could make models more error-prone. On the other hand such back-channels may be used to represent out of band communication that is relevant to the protocol and not properly network traffic. Since such out of band traffic could also be represented by non-network nodes in the top level anyway, this is not a strong argument against the chosen approach as explained in the previous paragraph.

2.2 Communication Channel Places

Network places, which have the channel pragmatic, represent the network and firing adjacent transitions corresponds to sending some data over the network. The sender and recipients are identified by the transitions on either side of the network places. Pragmatics on network places could, for example, include constraints on the network channel which corresponds to the assumptions made on the network used by the protocol. Such assumptions could be that package are guaranteed to arrive in order without duplicates, as TCP channels guarantee, or that there are no such guarantees, as is the case with UDP. Another example may be a constraint indicating that the channel is secure from an attacker being able to read the data which is provided by the Transport Layer Security (TLS) protocol [10]. How the communication channel should be initialized and used specifically should be specified in the configuration and platform models.

2.3 API Pragmatics

Figures 3 and 4 show the two modules for the A entity in the KC model. The behaviour of A is somewhat complex despite the simplicity of the protocol, because many steps are taken for each message. In the figures, pragmatics have been added to several elements in the model. Figure 3 shows how the protocol is initiated by placing a token on the place Init at the top of the figure. The token contains the addresses for A and B. These addresses are then combined with a nonce from the place Nonce and put on the place A to S which represent sending the message to the authentication server. At the same time, a copy of the nonce is placed on the WaitAuthenticate place. This place represents a state where A is waiting for a response from B.

When a message is placed on the place B to A, the transitions on the submodule associated with the Authenticate substitution transition become enabled. This submodule is shown in Fig. 4. Here the transition Receive Authentication (when enabled) stores As original Nonce and Bs Nonce in StoreA and StoreB, and then places a token on the Wait Decrypt place. Then the Decrypt Key transition can use the key in the place Key Store to decrypt the session key and nonce. The Authenticate transition is now able to perform the actual authentication of B. In the KC protocol, the authentication involves simply to check that the stored and the received and decrypted nonces are identical. The model does not explicitly show what should be done if the nonces are not the same. In practice, this would typically cause an error to be raised and the protocol would terminate. This is left out of the model in this paper for simplicity. Assuming that the authentication step is successful, Bs nonce is encrypted with the session key, which is generated by the authentication server and stored at the place Session Key Store, at the Encrypt Nonce transition, and finally put on the A to B place.

In the upper part of Fig. 3 there is a transition with the API pragmatic. This pragmatic symbolizes an entry point where the outside environment can interact with the protocol software. For target languages in the object-oriented paradigm this would typically be translated into a method with public access. In the KC example, the API pragmatic is the starting point of the protocol. It is given the name `kcAuthenticate` and takes two arguments; `toAddr` and `fromAddr`. Listing 2 shows an example of how the API transition could be translated into the signature of a Java method.

Listing 2: API method signature

```
public void kcAuthenticate(Id fromAddr, Id toAddr)
```

2.4 Operation Pragmatics

The operation pragmatic means the implementation should performing an operation such as printing data to the screen or calling a system library when this

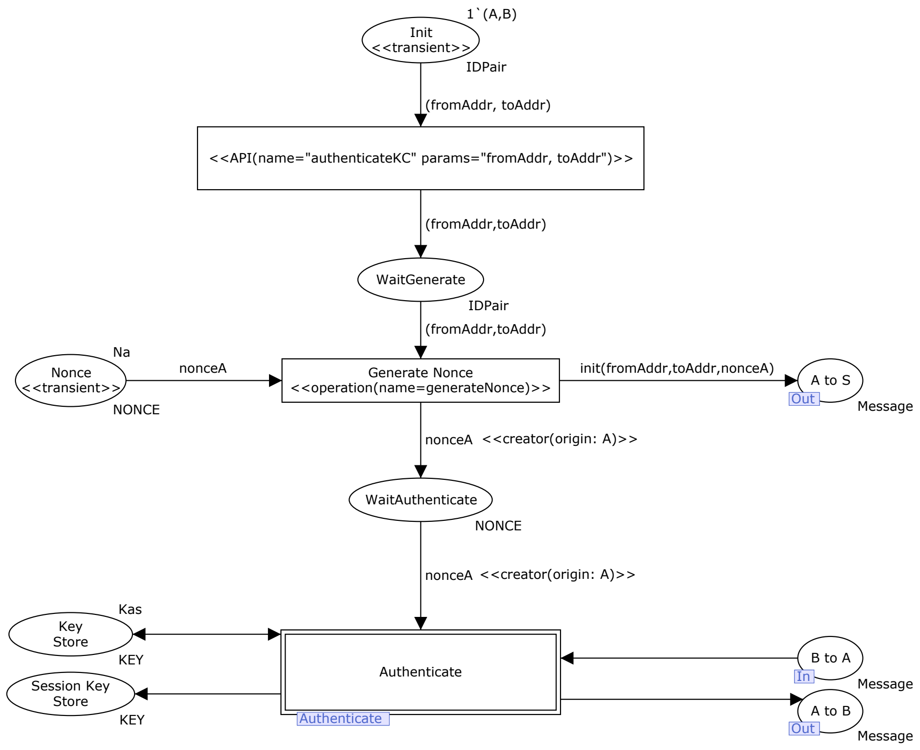

Fig. 3: Module A of Kao-Chow model.

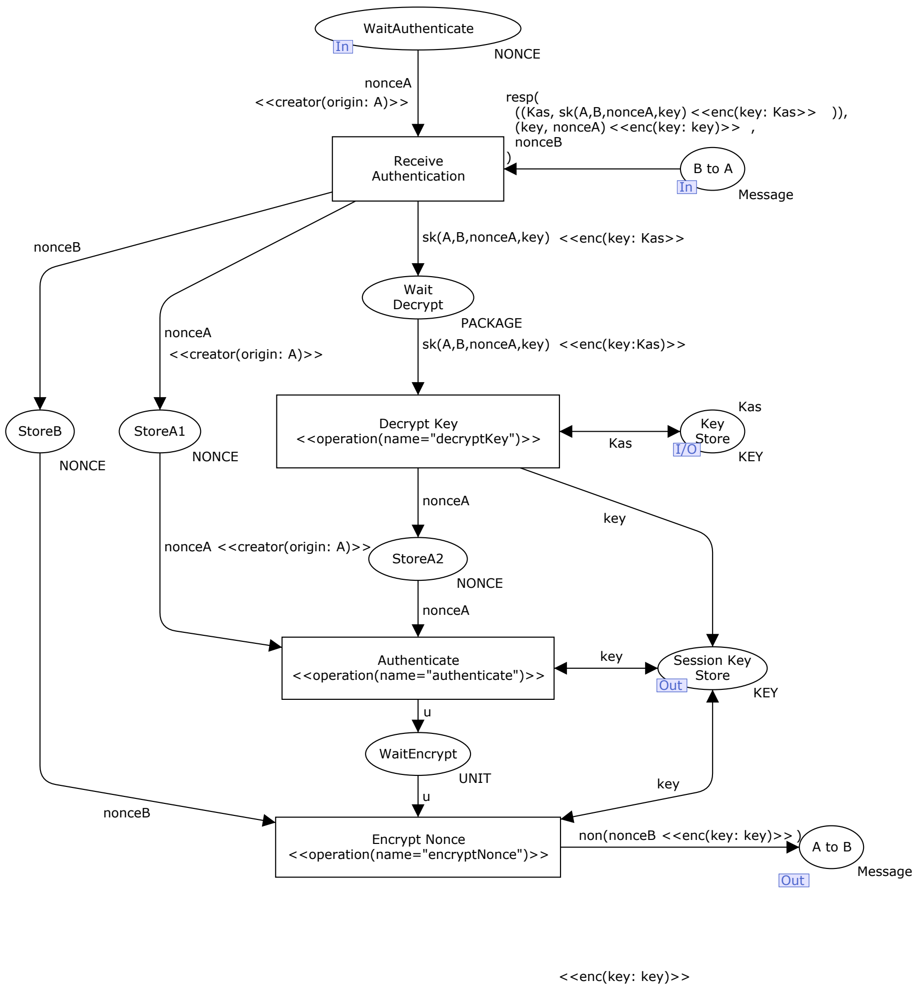

Fig. 4: Module Authenticate of Kao-Chow model.

pragmatic is encountered. Operation pragmatics are typically attached to transitions. The specific code that results from an operation in the Protocol Model is defined by the configuration and platform models.

In the KC protocol for entity A, there are four operation pragmatics. These are to generate a nonce (Generate Nonce in 3), encrypt (Encrypt Nonce in 4), decrypt (Decrypt Key in 4) and authenticate (Authenticate in 4). These pragmatics help to make explicit what operations are to be done for the transition with this pragmatic. Also it makes it possible for the Generator (see Fig. 1) to know how to generate code for these transitions, even if they are not modelled at the same level of detail as the implementation.

In an implementation, the Encrypt Nonce operation in the lower part of Figure 4 could be translated into what is shown in listing 3 on the Java platform where the encrypt method is already defined.

Listing 3: Encryption operation

```
String nonceReply = serverNonce.nonce.toString();
nonceReply = encrypt( nonceReply.bytes, sessionKey);
```

2.5 Transient Entities

Two places with the transient pragmatic are present in Figure 3. Model elements with the transient pragmatics are elements that are not considered by the generator, but may be useful for other uses of the model such as simulation and verification. The transient places in Figure 3 provide an initial state in the model which is necessary for simulation of the CPN model.

2.6 Data Pragmatics

In Fig. 4 several pieces of data have an enc pragmatic, for example on the arc between the Wait Decrypt place and the Decrypt Key transition in the middle of the figure. This pragmatic indicates that the data is encrypted with a given key. Encrypted data should only be used (read or manipulated) in transitions where the encryption key is available. In the KC example, encrypted data is only available after passing through a transition with a decryption operation where the correct key is available.

The enc pragmatic as shown here only takes symmetric encryption schemes into account. However, extending the pragmatic to also be able to model asymmetric encryption should be relatively simple. The enc pragmatic is an example of a domain specific pragmatic which is specific to the area of security protocols.

3 Configuration and Platform Models

Pragmatics in the model bring the model closer to an implementation by adding information that is useful for generating an implementation. Still the model is too abstract to generate code without making many assumptions about design choices and the underlying platform. We propose to use configuration and platform models to provide information so that the generator can generate an implementation.

The configuration model contains information about how to implement the protocol. It is likely to be highly dependent on both the protocol model and the platform model. It therefore seems possible that configuration models will not be reusable for other protocols or platforms. A typical design choice that will be represented in the configuration model is the choice of underlying network service to be used for communication between protocol entities. For example if for a protocol that has no constraints on the network layer service, a configuration would be whether to use UDP or TCP for the implementation.

The platform model should hold specific implementation details. In the example with the underlying network layer service, the platform model would hold information on how to set up, send and receive messages over UDP and TCP. The platform models are general in the sense that a platform model can be used to generate implementations of several protocols for the specific platform. In order to achieve this, the platform models, of course, need to support a wide range of features for different protocols and configurations.

Separating the configuration and platform models in this way makes it possible to reuse the models. Protocol models can be reused for different platforms and configurations. Platform models can also be reused to create protocol software for different protocols with different configurations for a specific platform.

4 Discussion

This paper has discussed some initial ideas for generating protocol software from models in a general way by annotating the model with pragmatics and adding configuration and platform information. This paper has also introduced a few specific pragmatics for protocol models that are exemplified by a model of the KC protocol. The list of pragmatics is by no means exhaustive, but provides a starting point for creating the first generation of technologies for protocol software modelling and generation using our approach. Additional information to be specified in configuration and platform models has also been introduced and argued for.

4.1 Related Work

In [18] a method for annotating CPNs is described. This method makes it possible to add auxiliary information to tokens in CPNs in layers of annotations. This approach is similar to the pragmatics presented in this paper in that both add information to CPNs. The approaches are different in that the pragmatics are added directly to the CPNs whereas the annotations in [18] are created and maintained separate from the underlying CPN model. Another difference is that the annotations are only concerned with tokens, while pragmatics can be added to places and transitions as well.

In [17] a restricted version of CPNs, called Colored Control Flow Nets (CCFN), are used to generate Java programs. This is done by first translating the CCFN to an intermediate model called a Annotated Java Workflow Net (AJWN) which is annotated by Java snippets derived from arc inscriptions in the corresponding CCFN.

In [16] a subclass of CPNs called Process-Partitioned CPNs (PP-CPNs) is introduced and used to automatically generate an implementation of the Dynamic MANET On-demand (DYMO) [11] routing protocol. The approach in [11] to generate code is to first translate the PP-CPN model into a control flow graph. The control flow graph is then used to construct an abstract syntax tree (AST) for an intermediate language which in turn is used to generate the AST of the target language. One difference to our approach is that in [16] information about the target platform and how translate model concepts to target language is contained in the generator instead of configuration and platform models. The method of [16] also defines a new subclass of CPNs instead of extending CPNs with annotations such as the pragmatics described here.

The notion of using different models for different layers of abstraction is also present in the Model Driven Architecture (MDA) [21] methodology of software engineering. In MDA three models are defined for a system. A Computation Independent Model (CIM) defines what a system is supposed to do and roughly corresponds to the protocol model as described in this paper. A Platform Independent Model (PIM) describes behaviour and structure of a system independent of the platform it is implemented on and a Platform Specific Model (PSM) combines the information in the PIM with all the details that are needed to generate an implementation of the system for the specified platform. The PIM and PSM are quite different from the configuration and platform models in this paper which do not include information on the software system itself, but rather design choices and how to implement these choices on the target platform for the given protocol model.

4.2 Future work

In the near future, we plan to use the KC model and manually simulate the code generation and then compare the implementation that is obtained through this simulation to an implementation that we have already created independently from the model. After that we will produce the first set of tools to automatically generate protocol software from HLPNs using the concepts of pragmatics and scope discussed in this paper as well as configuration and platform models.

Code generation will be done by model transformations. A significant challenge will be to gain confidence in the output of the generator. Formal verification of the generator will likely not be possible, but it is critical that we can maintain a high degree of confidence in the generated software. One technique that can be used to validate both the generator and the software it produces it to generate test suits based on the state space of the protocol code. Another technique is to rigorously test and examine several generated protocol implementations from several different protocol domains.

The protocol model itself should be verifiable. One approach to verifying protocols using CPNs has been described in [1]. We will study whether this and other approaches are applicable to protocol models with pragmatics as described in this paper. We will also look into how pragmatics can be used to help verify more properties about a protocol such as verifying that secret data is never places on a network channel in plain text and that the correct keys are always present to decrypt encrypted data.

References

1. Jonathan Billington, Guy Edward Gallasch, and Bing Han. A coloured petri net approach to protocol verification. In *Lectures on Concurrency and Petri Nets*, pages 210–290, 2003.
2. Roland Bouroulet, Raymond R. Devillers, Hanna Klaudel, Elisabeth Pelz, and Franck Pommereau. Modeling and analysis of security protocols using role based specifications and petri nets. In Kees M. van Hee and Rüdiger Valk, editors, *Petri Nets*, volume 5062 of *Lecture Notes in Computer Science*, pages 72–91. Springer, 2008.
3. Roland Bouroulet, Hanna Klaudel, and Elisabeth Pelz. A semantics of security protocol language (spl) using a class of composable high-level petri nets. In *ACSD*, pages 99–110. IEEE Computer Society, 2004.
4. Manuel Cheminod, Ivan Cibrario Bertolotti, Luca Durante, Riccardo Sisto, and Adriano Valenzano. Tools for cryptographic protocols analysis: A technical and experimental comparison. *Computer Standards & Interfaces*, 31(5):954–961, 2009.
5. Federico Crazzolara and Glynn Winskel. Events in security protocols. In *ACM Conference on Computer and Communications Security*, pages 96–105, 2001.
6. CPnets – Industrial Use.
   `http://cs.au.dk/cpnets/industrial-use/`.
7. G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall International Editions, 1991.
8. Internet Engineering Task Force. *RFC768: User Datagram Protocol*, August 1980. `http://tools.ietf.org/html/rfc768`.
9. Internet Engineering Task Force. *RFC793: Transmission Controll Protocol*, September 1981. `http://tools.ietf.org/html/rfc793`.
10. Internet Engineering Task Force. *RFC5246: The Transport Layer Security (TLS) Protocol, Version 1.2*, August 2008. `http://tools.ietf.org/html/rfc5246`.
11. Internet Engineering Task Force. *Dynamic MANET On-demand (DYMO) Routing*, July 2010. `http://datatracker.ietf.org/doc/draft-ietf-manet-dymo/`.
12. K. Jensen and L.M. Kristensen. *Coloured Petri Nets – Modelling and Validation of Concurrent Systems.* Springer, 2009.
13. K. Jensen, L.M. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(3-4):213–254, 2007.
14. I-Lung Kao and Randy Chow. An efficient and secure authentication protocol using uncertified keys. *Operating Systems Review*, 29(3):14–21, 1995.
15. L.M. Kristensen, P. Mechlenborg, L. Zhang, B. Mitchell, and G.E. Gallasch. Model-based Development of COAST. *STTT*, 10(1):5–14, 2007.

16. L.M. Kristensen and M. Westergaard. Automatic structure-based code generation from coloured petri nets: A proof of concept. In *Proc. of Int. Workshop on Formal Methods for Industrial Critical Systems*, volume 6371 of *Lecture Notes in Computer Science*, pages 215–230. Springer, 2010.
17. K. B. Lassen and S. Tjell. Translating colored control flow nets into readable java via annotated java workflow nets. In *Proc. 8th Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN 2007)*, pages 39–58, 2007.
18. B. Lindstrøm and L. Wells. Annotating coloured petri nets. In *Proc. of the Fourth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, pages 39–58, 2002.
19. Sebastian Mödersheim. Algebraic properties in alice and bob notation. In *ARES*, pages 433–440. IEEE Computer Society, 2009.
20. K.H. Mortensen. Automatic Code Generation Method Based on Coloured Petri Net Models Applied on an Access Control System. In *Proc. of ATPN'00*, volume 1825 of *LNCS*, pages 367–386. Springer, 2000.
21. OMG Model Driven Architecture. *Web Site*. `http://www.omg.org/mda/`.
22. W. Reisig. *Petri Nets - An Introduction*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.
23. Security Protocols Open Repository. Kao chow authentication v.1. `http://www.lsv.ens-cachan.fr/Software/spore/kaoChow1.html`.