Modelling the Configuration/Management API Middleware using Coloured Petri Nets*

Paul Fleischer
Department of Computer Science
University of Aarhus
pf@daimi.au.dk

Lars M. Kristensen
Department of Computer Science
University of Aarhus
kris@daimi.au.dk

ABSTRACT

The Configuration/Management Application Programming Interface (CMAPI) is a vendor-specific API and middleware-layer for configuration and management of components in embedded systems. CMAPI is used by TietoEnator Denmark in the implementation of a controller for the Generic Access Network (GAN) architecture. This paper presents a hierarchical Coloured Petri Net model of CMAPI that was developed in the process of applying Coloured Petri Nets and CPN Tools for the specification of the GAN controller.

Categories and Subject Descriptors

I.6.5 [**Simulation and Modelling**]: Model Development; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Petri nets*

General Terms

Design, Experimentation, Verification

Keywords

Coloured Petri Nets, CPN Tools, modelling, software specification, middleware.

1. INTRODUCTION

The Generic Access Network (GAN) [1] architecture specified by the 3rd Generation Partnership Project (3GPP) [2] allows access to common telephone services such as SMS and voice-calls via generic Internet Protocol (IP) networks. The operation of GAN is based on a *mobile station* (e.g., a cellular phone) opening an encrypted tunnel to a *security gateway* via an IP network. A *GAN controller* is responsible for relaying the commands being sent via this tunnel to the telephone network, which in turn allows mobile stations to access the services on the telephone network. The encrypted tunnel is provided by the Encapsulating Security Payload (ESP) mode of the IP security layer (IPSec) [7] and the Internet Key Exchange v2 (IKEv2) protocol [6].

TietoEnator Denmark [10] is working on providing solutions to support the GAN architecture which includes implementation of the GAN controller. Coloured Petri Nets (CP-nets or CPNs) [4] and CPN Tools [11] are currently being used by TietoEnator for specification and validation of the GAN controller design. In earlier work [3] we developed a CPN model of the GAN architecture with the purpose of specifying the usage scenarios and the overall architecture. Based on the work presented in [3], TietoEnator decided to use CPNs for specification of the GAN controller to be implemented. This paper presents the first step in the specification of the GAN controller using CPNs and is concerned with modelling the Configuration/Management Application Programming Interface (CMAPI). CMAPI is a vendor-specific API for configuration and management of components in embedded systems. It is a thin middleware-layer implemented in C that is compiled together with components using it. CMAPI is configured at compile-time with a static number of classes and parameters that will be supported at run-time. A CMAPI class consists of a C structure with pointers to functions implementing various aspects of the class: constructor, destructor, set/get of parameters, and actions. Besides this object-like behaviour, CMAPI also supports message exchanges through *packets*. Objects in CMAPI are structured in a hierarchy, which allows grouping of related elements.

In the context of the GAN controller, CMAPI is being used for controlling two network-layer components: IP Security (*IPSec*) [7] and Internet Key Exchange (*IKE*) [6]. IPSec enables encryption and authentication of data on a low level and IKE is used to negotiate parameters for IPSec. CMAPI is a key part of the implementation of the GAN controller and modelling of CMAPI is required before the detailed controller procedures can be specified using CPNs. CPNs have previously been used for modelling middleware. In [5] UML is used in conjunction with CPN to model middleware for pervasive healthcare systems. In [8, 9] CPNs were used for modelling and specification of selected parts CORBA. CMAPI being a vendor-specific middleware has not previously been exposed to formal modelling.

The rest of the paper is organised as follows. Section 2 gives an overview of the constructed CPN model. Section 3 explains how the objects, functions, and classes of CMAPI have been modelled. Section 4 shows how the CMAPI model component is used in the controller model and how it inter-

*Supported by the Danish Research Council for Technology and Production and TietoEnator Denmark.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PNTAP 2008, March 03, Marseille, France
Copyright © 2008 ICST 978-963-9799-20-2
DOI 10.4108/ICST.SIMUTOOLS2008.3043

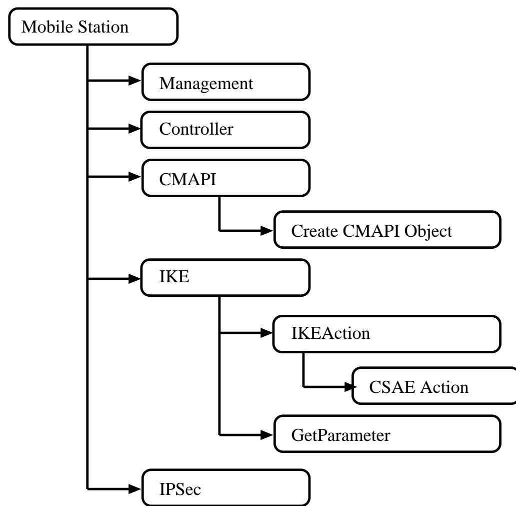

Figure 1: Module hierarchy for the CPN model.

acts with the modules modelling the controller logic and the IKE component. Finally, section 5 concludes and describes directions of future work. The reader is assumed to be familiar with the basics of hierarhical Coloured Petri Nets and the basics of C programming

2. MODELLING OVERVIEW

The CPN model is a hierarchical model organised in 10 modules. Figure 1 shows the *module hierarchy* of the CPN model. Each node in figure 1 corresponds to a *module* and Mobile Station represents the top-level module of the CPN model. An arc leading from one module to another module means that the latter module is a *submodule* of the former module. The model has been organised into four main parts: a Management part modellint the management plane, a Controller part modeling the logic of the GAN controller, a CMAPI part modelling CMAPI, an IKE part modelling the IKE modules, and an IPSec part modeling the IPSec modules. We have adopted the convention that a *substitution transition* and its associated submodule have the same name.

Figure 2 shows the top-level module of the CPN model of the GAN controller design. Each of the rectangles in figure 2 are substitution transitions. Each substitution transition has an associated submodule modelling the behaviour of the component in detail. The Controller is responsible for controlling the IPSec and IKE components, which are both represented by accordingly named substitution transitions. The modelling reflects how the CMAPI middleware connects the controller to the IPSec and IKE components.

The Controller itself is configured by the Management component which is also represented by a substitution transition. The Controller uses Application Programming Interfaces (*API*s) to communicate with these components. In general, the APIs consist of a number of C functions and structures. An API invocation corresponds to calling a C function. This is modelled as a place with an API colour set. In figure 2 there are four APIs. The Management API is modelled by the `MGMT_API` colour set and the place `Mgmt API`. The IKE API is modelled by the `IKE_API` colour set and the place `API`. Finally, CMAPI has two APIs. The Top API is modelled by the `CMAPI` colour set and the place `CMAPI Top`, and the Bottom API, modelled by the `CMAPI_LOW` colour set and the place `CMAPI Bottom`. The top API is used by the caller of CMAPI functions, while the bottom API is used by the code which implements the CMAPI classes. The top and bottom names have been introduced during the modelling of CMAPI to make the difference between the two APIs explicit.

This main purpose of the CMAPI modelling presented is to facilitate the later design of the controller component. The model is designed to model CMAPI such that it can be used as a component in other CPN models of software using the CMAPI middleware. This is of interest to TietoEnator as the CMAPI middleware is widely used in the development of their mobile solutions. The main focus has, however, been to support the modelling of the controller component. As the controller component does not make use of the message exchanges in CMAPI, these are currently not included in the model.

3. CMAPI MODELLING

This section describes the CMAPI middleware in more detail and explains how it has been modelled. As mentioned in the previous section, CMAPI has two APIs: Top and Bottom. All CMAPI calls from the Top API have a CMAPI object as argument. When a CMAPI function is called from the Top API, CMAPI performs simple sanity checks and calls the proper function registered for the class of the object. It is important to notice that this dispatching of CMAPI entails that the functions of the CMAPI classes are executed in the same thread as the caller. This has to be maintained in the model, in order to avoid accidentally turning a single-threaded system into a multi-threaded system.

Figure 3 shows the top-level of the CMAPI CPN modules. Tokens representing CMAPI calls are removed from the place `CMAPI Top` by one of five transitions: `Dispatch Object Create`, `Dispatch Object Destroy`, `Dispatch CMAPI Action`, `Dispatch CMAPI Set`, and `Dispatch CMAPI Get`. `Dispatch Object Create` is the only substitution transition, and will be described later. The other transitions work in the following fashion: Together with a token representing a CMAPI call from the place `CMAPI Top`, a token representing the CMAPI object is consumed from the place `Objects`. A token is then produced on the place `CMAPI Bottom` with enough information for the correct CMAPI classes to pick it up and process the CMAPI command.

In the next sections the modelling of CMAPI is described in more detail. Section 3.1 explains what CMAPI objects are and how they are modelled. Section 3.2 describes how calls to CMAPI from the top API are modelled. Finally, section 3.3 describes how CMAPI classes are modelled, and how the bottom API of CMAPI works.

3.1 CMAPI Objects

CMAPI keeps little state about its objects. Basically, only the class and its parent object are stored. All other information about the object is kept by the CMAPI class the object belongs to. Each object is uniquely identified by a pointer to its CMAPI structure. All this is captured in the model by the `CMAPI_OBJECT` colour set defined in listing 1.

The `instance`-field of the `CMAPI_OBJECT` record captures the unique identifier of the CMAPI object and corresponds to a pointer in C. This has been modelled by a simple inte-

Management
Management
Mgmt
API
MGMT_API
Controller
Controller
CMAPI
Top
CMAPI
CMAPI
CMAPI
API
IKE_API
CMAPI
Bottom
CMAPI_LOW
IPSEC
IPSEC
IKE Library
IKE

Figure 2: Top-level module of the controller model.

CMAPI
Top
I/O
CMAPI
Objects
CMAPI_OBJECT
(s,Return(ret))
Disptach
Object Create
Create CMAPI Object
obj
[#instance(obj) = #1(destroy_cmd)]
(obj,s,Call(DestroyObject
destroy_cmd))
Dispatch
Object Destroy
(s, Call(DestroyObject
destroy_cmd))
Return
obj
[#instance(obj) = #1(act)]
(s,Call(Action act))
obj
(obj,s, Call(Action act))
Dispatch
CMAPI Action
[#instance(obj) = #instance(set)]
obj
(s,Call(Set set))
obj
(obj,s, Call(Set set))
Dispatch
CMAPI Set
(obj,s,Return(ret))
[#instance(obj) = #instance(get)]
obj
(s, Call(Get get))
obj
(obj, s, Call(Get get))
Dispatch
CMAPI Get
CMAPI
Bottom
I/O
CMAPI_LOW

Figure 3: Top-level of the CMAPI CPN modules.

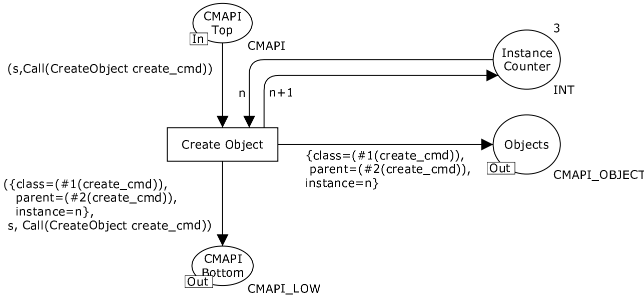

Figure 4: Object creation module of CMAPI.

ger, which is increased for each creation of CMAPI objects. The `parent`-field is the identifier of the parent object, while the `class`-field is the identifier for the object's class. In CMAPI, the classes are defined in a C enumeration. This is represented in the model by a ML enumeration, as can be seen in listing 1.

```
colset CMAPI_CLASS = with
                       cmapi_class_root        |
                       cmapi_class_main        |
                       cmapi_class_ikev2_main;

colset CMAPI_INSTANCE = INT;

colset CMAPI_OBJECT = record
                    class   : CMAPI_CLASS    *
                    parent  : CMAPI_INSTANCE *
                    instance: CMAPI_INSTANCE;
```

Listing 1: Colour sets for describing CMAPI objects

Figure 4 shows the object creation module of the CMAPI model. The transition `Create Object` consumes a token from the place `CMAPI Top`, which is a request to create a new object. The place `Instance Counter` is used to keep track of unique object identifiers and is increased by one for each object creation. Two tokens are produced by the transition `Create Object`. One is placed in the place `Objects`. This is used to keep track of the object hierarchy and a mapping between objects and classes. The other token produced is put in the place `CMAPI Bottom`, and is later consumed by the relevant CMAPI class as will be explained in section 3.3.

When constructing CMAPI objects, parameters can be passed along in a parameter container. In C, parameters are identified by structures. Each parameter container can contain a number of CMAPI parameters, but only a single value for each parameter. In the CMAPI model, a parameter consists of a pair of strings, where the first is the parameter name, and the second is the parameter value. CMAPI containers are modelled as lists of parameters.

Listing 2 shows the definition of the `CMAPI_PARAM` and `CMAPI_PARAM_CONTAINER` colour sets. As CMAPI only allows one instance of each parameter within a parameter container, it is possible to create ML functions that manipulate the parameter list under the assumption that each key exists only once.

```
colset CMAPI_PARAM = product STRING * STRING;
colset CMAPI_PARAM_CONTAINER =
                         list CMAPI_PARAM;
```

Listing 2: `CMAPI_PARAM_CONTAINER` colour set

3.2 CMAPI Functions

Keeping in mind that the execution stack keeps track of the return-point in C during function calls, CMAPI calls are modelled by the colour set `CMAPI_CMD` as can be seen in listing 3.

```
colset APP       = with controller;
colset CMAPI_CMD = union Call: CMAPI_CALL   +
                       Return: CMAPI_RETURN;
colset CMAPI     = product APP * CMAPI_CMD;
```

Listing 3: The `CMAPI_CMD` and related colour sets

The colour set is a product of an application (`APP`) and a CMAPI command (`CMAPI_CMD`). The idea behind the `APP` colour is to be able to identify the caller of the CMAPI function, modelling the return-point on the stack in C. Only the transitions representing the caller component are allowed to produce CMAPI calls for the given application and consume CMAPI returns for it. In the controller model there is only a single caller of CMAPI functions, namely the controller itself. A colour belonging to `CMAPI_CMD` is either a CMAPI call (of colour `CMAPI_CALL`) or a return from a CMAPI call (of colour `CMAPI_RETURN`). Listing 4 shows the `CMAPI_CALL` colour set. It is a union of the four basic operations supported by CMAPI.

The constructor `CreateObject` is used to create a new CMAPI object. The `CMAPI_CREATE` colour set is a product of the three parameters used to construct a new CMAPI object: Its class, its parent, and a list of parameters. The constructor `DestroyObject` is used to destroy an existing CMAPI object. The destroy operation takes two parameters, as represented by the `CMAPI_DESTROY` colour set. The first parameter is the CMAPI object that is to be destroyed, and the second is a parameter list.

The `Action` constructor is used to represent a CMAPI action. Actions in CMAPI correspond to method calls in object-oriented languages. The `CMAPI_ACTION` colour set is a product of the CMAPI instance to invoke an action on, the action to invoke, and finally a number of parameters. Actions are in CMAPI identified by integers defined by each CMAPI class. These integers are named by using C enumerations. Thus, actions are modelled as ML enumerations.

In order to set and get parameters of CMAPI objects, the `Set` and `Get` constructors model the parameter setting and getting of CMAPI. The common `CMAPI_SETGET` colour set used by both `Set` and `Get` constructors, is a record with the `instance`-field identifying the CMAPI object instance and the `params`-field identifying the parameter list to get or set. Note, that the CMAPI API has many more functions for e.g., browsing the object hierarchy which are not modelled.

```
colset CMAPI_CREATE  = product
                        CMAPI_CLASS          *
                        CMAPI_INSTANCE       *
                        CMAPI_PARAM_CONTAINER;

colset CMAPI_DESTROY = product
                        CMAPI_INSTANCE       *
                        CMAPI_PARAM_CONTAINER;

colset CMAPI_CLASS_ACTION = with
                          ikev2_csae_start;

colset CMAPI_ACTION = product
                        CMAPI_INSTANCE       *
                        CMAPI_CLASS_ACTION   *
                        CMAPI_PARAM_CONTAINER;

colset CMAPI_SETGET = record
              instance: CMAPI_INSTANCE       *
              params  : CMAPI_PARAM_CONTAINER;

colset CMAPI_CALL = union
                CreateObject : CMAPI_CREATE  +
                DestroyObject: CMAPI_DESTROY +
                Action       : CMAPI_ACTION  +
                Set          : CMAPI_SETGET  +
                Get          : CMAPI_SETGET;
```

Listing 4: The `CMAPI_CALL` colour set

3.3 CMAPI Classes

As mentioned in section 3.1, classes are identified by a unique number (modelled as a ML enumeration). This section explains how the class functionality is modelled using CPNs.

When defining a CMAPI class in C, a structure is filled with all the necessary information, e.g., pointers to functions handling the different CMAPI operations for objects of the class. Hence, classes basically consist of a class struct and a number of functions to handle operations. In the CPN model, classes consist of a submodule which consumes and produces tokens of the `CMAPI_LOW` colour set. They are consumed when the class is being invoked by CMAPI, and produced when the class returns control to CMAPI.

Listing 5 shows the `CMAPI_LOW` colour set, which is used to model the communication between CMAPI and its classes. The colour set is a product, like the `CMAPI` colour set used by the top API, with the addition of the relevant CMAPI object as the first component. As the `CMAPI_OBJECT` colour set contains the class identifier of the object, the first field of `CMAPI_LOW` can be used to identify which CPN module has to handle the token. In C, this information is not explicit, but lies implicitly in the CMAPI class struct by means of the function pointers.

```
colset CMAPI_CMD = union Call: CMAPI_CALL +
                       Return: CMAPI_RETURN;
colset CMAPI_LOW = product CMAPI_OBJECT *
                           APP          *
                           CMAPI_CMD;
```

Listing 5: The `CMAPI_LOW` and related colour sets

Figure 5 shows the IKE submodule, which implements multiple CMAPI classes (as listed in the guard for the transition `Receive CMAPI Call`). The transition `Receive CMAPI Call` consumes only tokens which are CMAPI commands destined to one of its classes, and places them on the place `CMAPI Calls`. Depending on the call type, either the `Create IKEv2 Object`, `Action`, or `Destroy IKEv2 Object` transition consumes the token. All three transitions produce a `CMAPI_LOW` return token, which is placed on the `CMAPI Pool`. Section 4.2 describes the details of figure 5.

4. EXAMPLES OF USAGE

This section describes how to model both CMAPI callers and CMAPI classes. The controller is used as an example of a CMAPI caller, while the IKE component is used as an example of a CMAPI class. The interaction between the controller and all components is illustrated in figure 2. The management part and the IKE API are not discussed here. Neither the controller nor IKE model are finished models, and are only used here to explain the usage of the CMAPI model. Also, the details of the IKE-protocol and thus the IKE model entities are not described.

4.1 The Controller

As mentioned earlier, the modelled part of the controller is responsible for handling the IPSec and IKE components. The controller itself is configured by a management plane. For the purpose of demonstrating how the controller communicates with the IKE component through CMAPI, the controller is modelled as a state-machine as shown in figure 6. All state-transitions are connected to the place `IKE CMAPI` with double arcs, indicating that all transitions perform calls to IKE via CMAPI. The place `Controller Objects` is a fusion place holding all CMAPI objects the controller knows about. The colour set `CMAPI_OBJECT_MAP` (listing 6) maps a variable name to a CMAPI object. In the following, selected state-transitions will be used to show how CMAPI is invoked for Objection Creation (section 4.1.1), Action (section 4.1.2), and Object Destruction (section 4.1.3).

```
colset CMAPI_OBJECT_MAP = product STRING *
                                  CMAPI_OBJECT;
```

Listing 6: The `CMAPI_OBJECT_MAP` colour set

4.1.1 Object Creation

The transition `Create IKE CSA` has been chosen to show how CMAPI objects are created. CSA is short for **C**hild **S**ecurity **A**ssociation, and is a IKE term. The transition is a substitution transition, with the `Create IKE CSA` module as submodule tag. The module is shown in figure 7 and demonstrates how CMAPI objects are created. Two objects are being created, first one of type `cmapi_class_ikev2_csad` (**C**hild **S**ecurity **A**ssociation **D**atabase), and then one of type `cmapi_class_ikev2_csae` (**C**hild **S**ecurity **A**ssociation **E**ntry).

Object creation consists of two steps. First, the "create object"-command has to be sent to CMAPI. Second, the return from CMAPI has to be retrieved. The transition `Create IKE CSAD` performs the first step. It consumes a token

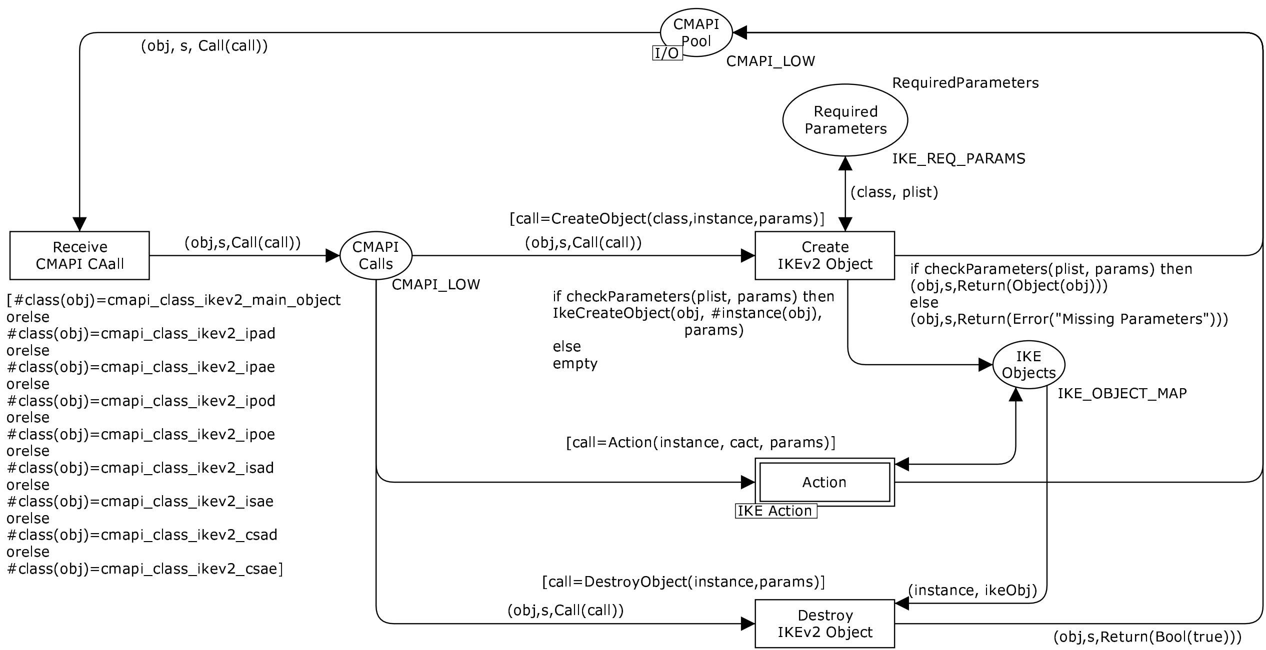

Figure 5: IKE submodule implementing multiple CMAPI classes.

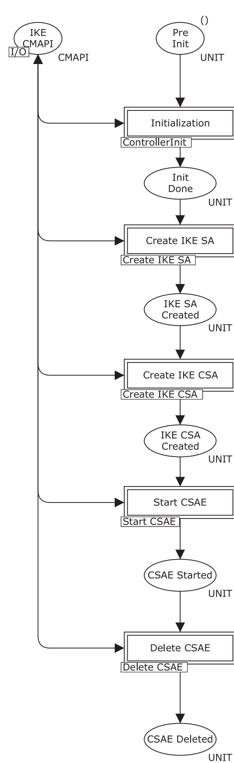

Figure 6: CPN module of the controller.

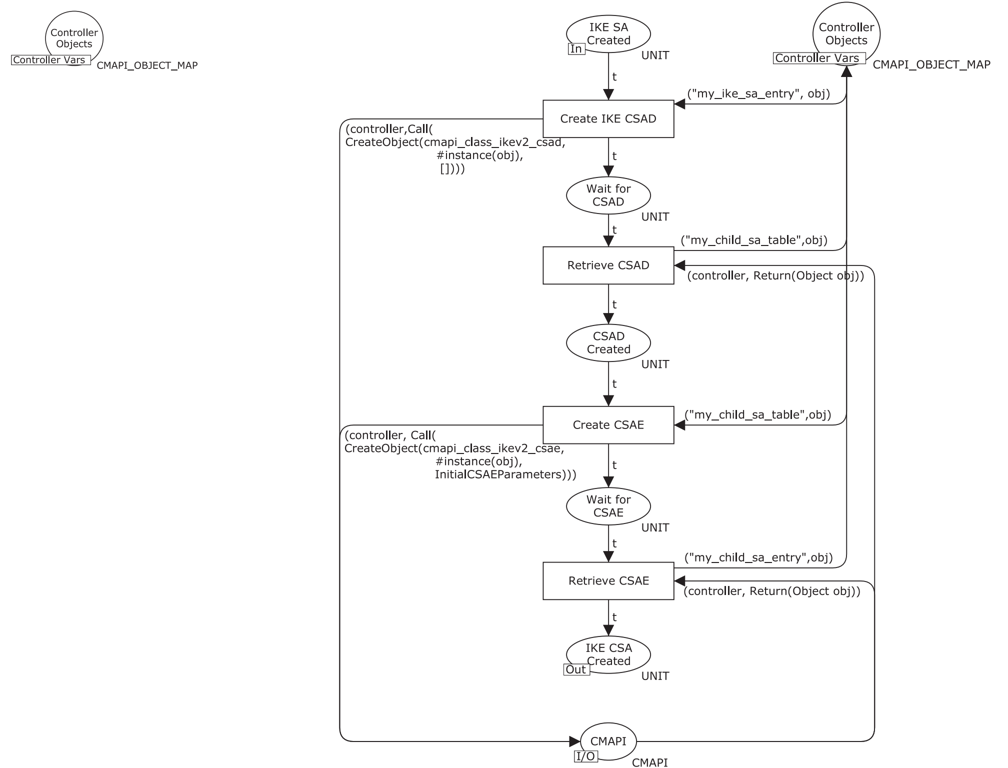

Figure 7: Create IKE CSA module.

from Controller Objects, in order to find the CMAPI object named "my_ike_sa_entry". This object is to be used as a parent for the new CMAPI object. The token returned to the place Controller Objects, as the variable "my_ike_sa_entry" should be available at a later point in time. The transition produces a token of colour CMAPI, and places this on the place CMAPI. This token represents the command sent to CMAPI. Recall, that the CMAPI colour set is a tuple, consisting of a sender application and a CMAPI command. The application here is controller, while the CMAPI command is a Call with the CreateObject operation. As mentioned in section 3.2 the create operation has a 3-tuple as parameter: The class, the instance identifier of the CMAPI parent object, and a parameter list. In this example, the class is cmapi_class_ikev2_csad. The parent object is obj, obtained from the place Controller Objects. The parameter list is empty. By producing a unit token and placing it at the place Wait for CSAD, the state machine changes state and awaits the CMAPI response.

The second step consists of retrieving the CMAPI response and is performed by the transition Retrieve CSAD. It consumes a CMAPI-token from the place CMAPI, which is destined for controller and has a CMAPI Return command with an Object value. The object is assigned to variable "my_child_sa_table" and stored in the place Controller Objects. The transition Retrieve CSAD also consumes a unit token from Wait for CSAD and produces one on CSAD Created, which indicates a state-change. The transitions Create CSAE and Retrieve CSAE model the creation of a CMAPI object of type cmapi_class_ikev2_csae. The only difference from the CSAD creation is that the constant InitialCSAEParameters is given as parameter list.

4.1.2 Action

Figure 8 shows the Start CSAE module and illustrates how CMAPI actions are invoked. In the figure it is the action ikev2_csae_start with no parameters, which is being invoked on the obj object, retrieved from the place Controller Objects. As with object creation described in section 4.1.1, the process of invoking an action consists of two steps: Requesting a CMAPI command and retrieving the result. The transition Start CSAE requests the command, while the transition Receive Response retrieves the result.

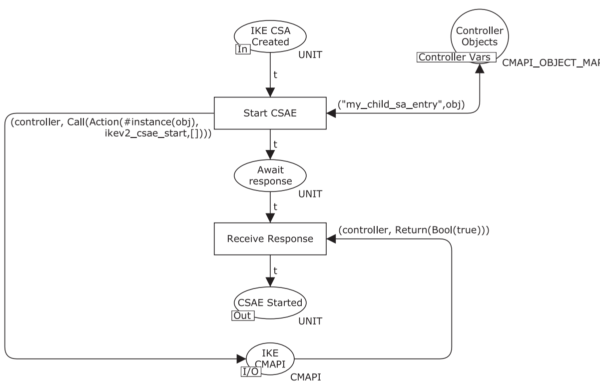

Figure 8: Invoke demonstrates CMAPI actions.

4.1.3 Object Destruction

Object destruction is very similar to action invocation described in section 4.1.2. The Delete CSAE module from the controller example is depicted in figure 9. The main difference between object deletion and action invocation, is the fact, that the variable-token is not put back into the place Controller Objects. This models the fact that there no longer is a valid mapping from the variable name to a CMAPI object.

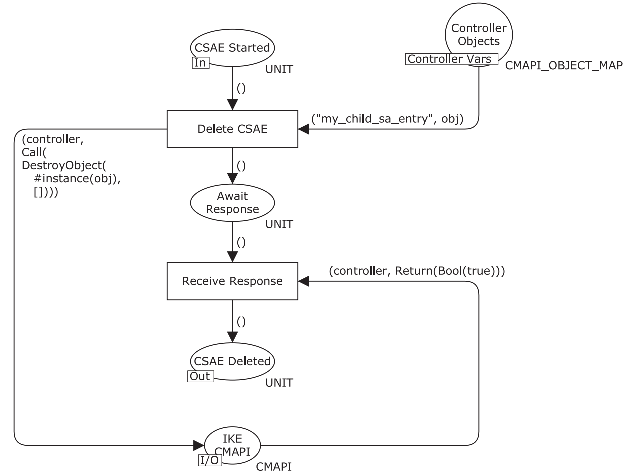

Figure 9: Delete a CMAPI Object.

4.2 The IKE Component

In order to illustrate how CMAPI classes are modelled, IKE is used as an example. Figure 5 showed the topmost page of the IKE module. As mentioned in section 3.3, this module implements multiple CMAPI classes as can be seen by the guard of the transition Receive CMAPI Call.

The general flow is as follows: CMAPI Calls for supported classes are filtered by the transition Receive CMAPI Call and placed on the place CMAPI Calls. Depending on the call type, the token is consumed by either one of the following transitions: Create IKEv2 Objects, Action, or Destroy IKEv2 Object. This represents three of the four possible CMAPI operations. Reading and writing of parameters is currently not implemented in the IKE module. The three transitions will be explained in more detail in the following.

Create IKEv2 Object has two goals: It has to verify that the required parameters were supplied, and it has to create the actual object, with all the necessary state. Recall, that CMAPI itself only keeps enough information to have an object hierarchy and dispatch calls to the correct CMAPI classes. The place Required Parameters contains tokens of the colour set IKE_REQ_PARAMS (see listing 7).

For each class implemented by the IKE module there is a single token, with a list of required parameters. An empty list means that no parameters are required. The function checkParameters takes a list of required parameters and a parameter container. It checks if all parameters in the parameters list are present in the parameter container. Actual creation of the IKE object is done by placing a new token on the place IKE Objects. The colour set of this place is IKE_OBJECT_MAP, which is shown in listing 7. It maps a CMAPI_INSTANCE to an IKE_OBJECT. The latter is a union

of all possible states needed for IKE objects. As can be seen in listing 7, there are only two objects with attributes assigned, IKE_ISAE and IKE_CSAE. The duplication of the CMAPI_INSTANCE inside the IKE_ISAE and IKE_CSAE colour sets, simplifies the IKE module somewhat. The other fields are object states which will not be described in this paper.

```
colset IKE_REQ_PARAMS = product CMAPI_CLASS *
                                list STRING;
colset IKE_ISAE = record
                 cmapi_id    : CMAPI_INSTANCE *
                 sourceIP    : STRING         *
                 interfaceId : STRING;
colset IKE_CSAE = record
                 cmapi_id    : CMAPI_INSTANCE *
                 incomingSPI : STRING         *
                 localIP     : STRING         *
                 remoteIP    : STRING         *
                 started     : BOOL;
colset IKE_OBJECT = union IkeNone +
                          IkeSae : IKE_ISAE +
                          IkeCsae: IKE_CSAE;
colset IKE_OBJECT_MAP = product
                          CMAPI_INSTANCE *
                          IKE_OBJECT;
```

Listing 7: Colour sets of the IKE module

The transition Create IKEv2 Object places a new token on the place IKE Objects if the checkParameters function returns true. In other words, objects are only created if all required parameters are specified. The IkeCreateObject function takes CMAPI_OBJECT, CMAPI_INSTANCE, and CMAPI_PARAM_CONTAINER. A new IKE_OBJECT is returned. The function will not be described in detail here, but it simply uses the class-field of the CMAPI_OBJECT variable to chose which variant of IKE_OBJECT to create. The function checkParameters is also responsible for controlling the return token generated by the transition Create IKEv2 Object. If all required parameters are present, the CMAPI object is returned, else an error value is returned.

The transition Destroy IKEv2 Object consumes (besides the CMAPI call token) the relevant object from IKE Objects, and does not put it back. This represents the fact that the object is destroyed. Other than that, it produces a return token with the boolean value true.

The most complex operation is the CMAPI action. It is modelled in the submodule IKE Action depicted on figure 10. CMAPI call tokens are consumed from the place CMAPI Calls and the object in the CMAPI call is matched with the relevant IKE object by performing a look up in the place Ike Objects. A token of colour IKE_ACTION is placed on the place IKE Actions, which contains all the necessary information needed for the actual action-part. The transition CSAE Action (which is omitted here) then performs the actual action.

5. CONCLUSION AND FUTURE WORK

A general purpose modelling of the CMAPI middleware has been presented in this paper focusing on the main operations of CMAPI: object creation, object destruction, action, reading of parameters, and writing of parameters. All four have been modelled in the CMAPI module. Object creation, destruction, and actions have been illustrated using IKE as an example of a CMAPI class. Furthermore, we have used the GAN controller to illustrate CMAPI usage. The controller and IKE modules are examples of how CMAPI connects a caller (the controller) to CMAPI classes (the IKE component).

The main documentation of the CMAPI system is its C and header files, which are documented using Doxygen. Some parts of the documentation are slightly outdated. Thus it is difficult to say how closely CMAPI has been modelled without creating more example modules using the CMAPI module, or performing a review of the model with the developers of CMAPI. Both of these activities are planned as part of future work. An area of improvement in the CMAPI module is how object creation is handled. Studying the IKE module, it can return an error rather than creating an IKE object. However, the CMAPI module does create an object no matter if the creation of the IKE object was successful or not. As the CMAPI module is supposed to be widely applicable, this is an important issue. The solution is to create a special "create error" return token, which the CMAPI module can consume and at the same time remove the relevant object from the place Objects. Another area for future work is the recursive handling of object deletion.

As the project of modelling the controller proceeds, the CMAPI model will be evaluated more thoroughly. In order to ensure that the CMAPI module models the function call properly, a state space analysis could be used to ensure that after every call-token a single return-token is placed on the place CMAPI Top. In near future, TietoEnator is going to implement the GAN controller and the plan is to use the CPN model as a basis of for the implementation. Besides the advantages of having a formal model which can be validated by means of state space analysis, the goal is to generate template code for the GAN controller directly from a CPN model of the controller. This will ease the work of the initial implementation and help ensure that the implementation is consistent with the design as specified by the CPN model.

Acknowledgements.

The authors wish to thank the anonymous reviewers for their constructive and detailed comments that have helped us to improve the paper.

6. REFERENCES

[1] 3GPP. Digital cellular telecommunications system (Phase 2+); Generic access to the A/Gb interface; Stage 2. 3GPP TS 43.318 version 6.9.0 Release 6, March 2007.

[2] 3GPP. Website of 3GPP. http://www.3gpp.org, May 2007.

[3] P. Fleischer and L. M. Kristensen. Towards Formal Specification and Validation of Secure Connection Establishment in a Generic Access Network Scenario. In *Eighth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, volume 584 of *DAIMI PB*, pages 9–28, October 2007.

[4] K. Jensen, L. M. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. *STTT (International Journal on Software Tools for Technology Transfer)*, 2007.

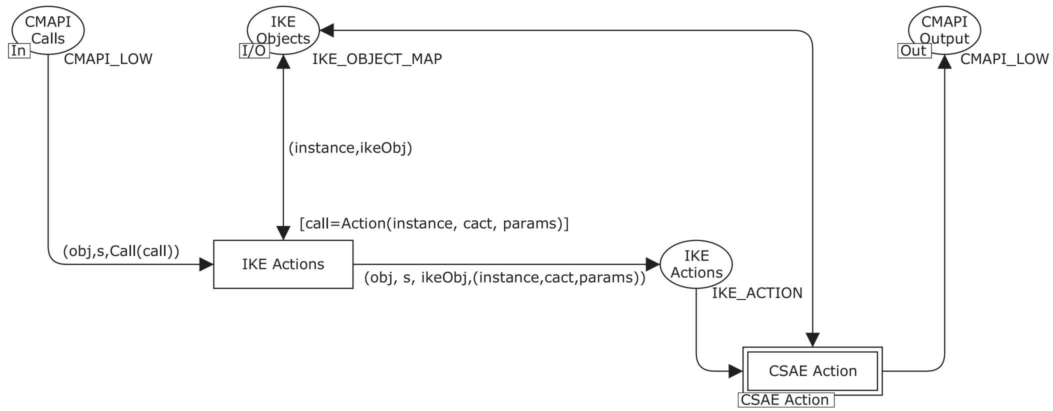

Figure 10: IKE submodule implementing actions.

[5] J. B. Jørgensen. Coloured Petri Nets in UML-Based Software Development - Designing Middleware for Pervasive Healthcare. In K. Jensen, editor, *Fourth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, volume 560 of *DAIMI PB*, pages 61–80, August 2002.

[6] C. E. Kaufman. Internet Key Exchange (IKEv2) Protocol. RFC4306, December 2005.

[7] S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC4301, December 2005.

[8] A. Singh and J. Billington. Creating an internet inter-orb protocol service specification. In *Proc. of Workshop on Formal Methods Applied to Defence Systems*, volume 12 of *Conferences in Research and Practice in Information Technology*, pages 57–66. Australian Computer Society, 2002.

[9] A. Singh and J. Billington. A formal service specification for iiop based on iso/iec 14752. In *Proc. of 5th International Conference on Formal Methods for Open Object-Based Distributed Systems*, pages 111–125, 2002.

[10] TietoEnator. Website of TietoEnator Denmark. `http://www.tietoenator.dk`, 2007.

[11] University of Aarhus. CPNTools. `http://www.daimi.au.dk/CPNTools`, 2007.