Data Encapsulation and Data Abstraction with Petri Nets – a graphical Visualization of Modules*

Olaf Fricke

Technical University Berlin
Institute for Communication and Software Technology
Computation and Information Structures (CIS)
Einsteinufer 17, D-10587 Berlin
email: ofricke@cs.tu-berlin.de

Abstract

In this paper, the concepts of *data encapsulation* and *data abstraction* are applied to Petri nets. Encapsulation is realized by enclosing a data object inside a high level Petri net. Access to the data object is permitted only via exported operations.

Abstraction is realized by separating a definition net and an implementation net. This separation is a well known concept of modules, used to handle the complexity problem of large systems and to increase the reusability and maintainability. But this concept has not yet been applied to a modelling technique with Petri nets.

Two new hierarchy concepts are introduced to establish the connection between the implementation net and the definition net of one module and between different modules, respectively.

Together with the new methodical concepts, a graphical notation of the Petri net model is introduced, which results in an intuitive visualization of Petri net modules.

Keywords: High level Petri nets, data abstraction, encapsulation, modules, modelling, hierarchy concepts, visualization.

1 Introduction

A well known concept in software engineering are modules. They are used to make the complexity of large systems more manageable. Therefore, the system is split into a number of subsystems, allowing the independent development of parts of the system as well as the reuse of the same module in different systems. Modules can be used to cluster data structures and procedures belonging to the same logical task.

One very important aspect of modules is the possibility to hide implementation details. Local values or operations cannot be seen outside the module, even their existence is unknown. This information hiding is the main structuring aspect of modules, since it allows abstraction from details. To realize the abstraction, each module is normally divided into at least two parts:

1. The *definition* part is used to name the exported data types, variables or operations (export interface) and to define the import from other modules (import interface). This part of a module is called *interface specification* in the context of algebraic specifications (see [7]). Only the definition part is made visible to the public.

*This work is part of the common research project "DFG-Forschergruppe PETRINETZ-TECHNOLOGIE" between H. Weber (Coordinator), H. Ehrig (both from the Technical University Berlin) and W. Reisig (Humboldt University Berlin), supported by the Deutsche Forschungsgemeinschaft (DFG).

2. The *implementation* part contains the actual implementation details of the module (body part). Here all the exported items have to be realized. It is possible to change or even to extend the implementation part as long as it fulfils the definition part.

In addition to this separation, different kinds of modules are distinguished according to their contents. In this paper, we will treat the so called *abstract data type*. It exports the name of a data type together with operations to create, manipulate and destroy concrete data objects of this type. All implementation details are hidden, and access is permitted only via exported operations.

This kind of module realizes *data abstraction*, since it abstracts from implementation details of data types. Only the use of a data type or of concrete data objects is of interest. Data abstraction must be seen in contrast to the conventional *functional abstraction*, which is realized by functional decomposition or refinement, eg. transition substitution in the context of coloured Petri Nets. We will apply the concepts of abstract data types to develop a new kind of modelling technique for Petri nets.

There are some papers treating other modular approaches for Petri nets. But the focus of these papers is a compositional view on Petri nets with the aim of a modular approach to analysis (see for example the *modular CP-nets* in [5] or a revised definition in [14]). But these formalisms only examine encapsulation of behaviour.

A number of approaches to integrate data abstractions and Petri nets already exist in literature. Most of these approaches were developed in the context of high level Petri nets in combination with algebraic specifications. Some examples are *OBJSA nets* (see [1, 2]), *CO-OPN* (see [4]), *AHL-nets* (see [8]) or *algebraic system nets* (see [12]). The data abstraction is realized in a textual form with an inscription language, defining the data types that can be used in the nets. In all of these formalisms the concrete data objects are represented by high level tokens, containing all the information. The availability of a token on a certain place triggers an operation on the data object. Therefore, no real protection against unauthorized access is possible, because every transition can access the internal representation of the tokens.

Another problem is the concept of connecting objects and their operations. In nearly all Petri net approaches operations are represented by transitions. A call of an operation for one object is modelled by sending the high level token representing the object to the operation. This is an inverse view compared to concepts of object orientation. There an object sends a message to an operation and not vice versa.

Other approaches combine concepts of object orientation and Petri nets to achieve some kind of data encapsulation. These net formalisms suffer either from a rather complex formalism like the *Object Petri Nets* (see [13, 15, 14]), or they hide too much information (like the call of an operation) in an object oriented inscription language (see [9]).

Most of the object oriented approaches use an (implicit) scheduler to direct operation requests to the right modules. Although these approaches may realize some kind of encapsulation, they hide the communication between the modules, implying that the import relations between the modules are not visible. Therefore they are not very suitable for a modular Petri net approach. Some examples are *G-Nets* (see [6]), the *class diagrams based on Coloured Petri Nets* (see [3]) or the *G-CPN* formalism (see [17]).

In this paper, a new approach concerning data abstraction in Petri nets is presented. The main idea is to hide the implementation details of an abstract data type. The concrete data object resides *inside* a high level Petri net. Access to the data object is permitted only via exported operations. These operations are shown in a definition net, representing an abstract view of the module. Explicit message passing is used to make the cooperation of modules visible.

Coloured Petri nets (CPN) have been chosen as a formalism for the net (see [11] for details). The presented Petri nets were drawn and simulated with the tool Design/CPN (cf. [16, 18]).

The paper is a shortened and revised version of [10]. It is organized as follows: In section 2, first concepts of data encapsulation are derived with the help of an example. This section treats the definition part of a module. The result will be a Petri net with one special interface place. This place *surrounds* all net elements of the module, forming a visual border for the module. Additionally, the concept of *unique identifiers* is introduced to allow the dynamic creation and deletion of data objects as well as the management of various numbers of data objects. Section 3 gives some hints how to use the definition net in other modules or in an application. The implementation part of a module is described in section 4. It is realized with a hierarchical Petri nets. In section 5, the used hierachical concepts are discussed and it is shown how these concepts can be modelled by conventional place fusion and transition substitution. The paper finishes with a short conclusion.

2 Definition part of an abstract data type

The well known stack is used as a general example to illustrate and motivate the main ideas behind the modelling of data abstraction with Petri nets. Look at the small code fragment shown in figure 1. It represents the definition part of a stack, written in the imperative programming language Modula-2. In this part of the module all exported operations for the stack are defined, whereas all implementation details are hidden.

```
DEFINITION MODULE ADTStack;
TYPE Stack; (* for INTEGER *)

PROCEDURE NewS(): Stack;
PROCEDURE DelS(VAR s: Stack);
PROCEDURE Push(VAR s: Stack, dat: INTEGER);
PROCEDURE Pop(VAR s: Stack, VAR dat: INTEGER);
PROCEDURE IsEmpty(s: Stack): BOOLEAN;
PROCEDURE Clear(VAR s: Stack);
```

Figure 1: Code fragment of an abstract data type

The intended functionality should be obvious. The operation `NewS()` creates a new stack while the operation `DelS(s)` destroys a stack `s`. The Operation `Push(s,x)` writes the value `x` onto the stack `s` while the operation `Pop(s,y)` assigns the top value of the stack `s` to the variable `y`. The operation `IsEmpty(s)` is used to check if the stack is empty, and the operation `Clear(s)` empties the stack no matter of its current state. It is important to see that the number of stack objects can dynamically vary in time and that all stacks must be distinguishable.

To define the semantics of the stack the following two equations can be used (let σ be a function describing the current state of a stack):

$$\{\sigma(\mathtt{s}) = \sigma_1\}\mathtt{Push}(\mathtt{s},\mathtt{x});\mathtt{Pop}(\mathtt{s},\mathtt{y})\{\sigma(\mathtt{s}) = \sigma_2\} \quad \Rightarrow \quad \mathtt{y} = \mathtt{x} \wedge \sigma_1 = \sigma_2 \tag{1}$$

$$\mathtt{Clear}(\mathtt{s});\mathtt{Pop}(\mathtt{s},\mathtt{y}) \quad \Rightarrow \quad \mathsf{failure} \tag{2}$$

Equation 1 says that doing first a `Push(s,x)` on a stack `s` (with state σ_1) for some value `x` followed by a `Pop(s,y)` for some variable `y` implies that `y` is assigned the value of `x` and that the state σ_2 of the stack after doing both operations is equal to its former state σ_1.

Equation 2 expresses a general restriction while using the stack: it is a failure to do a `Pop` on an empty stack.

When building a net model for the definition part of a module, all the exported operations should be visible together with their parameters. Therefore, exactly one named transition exists for each operation. Each transition is connected with exactly two arcs to an interface place, one for each direction. The interface place is used to communicate with the module. In addition, some more places are used to denote other imported modules and to represent causal relations between the operations.

One important aspect of the modelling concerns the representation of the interface place itself. It is drawn *around* the places and transitions of the module. This is a very important concept for visualizing the data abstraction, since it shows directly the border of the module. But it does not extend the Petri net formalism because it is a representation concept only (imagine to graphically shrink the interface place and to put it between the transitions; then everything is as usual).

To distinguish different operation calls, the used tokens must be typed. For each operation two types are defined: the first is used to trigger the corresponding operation, whereas the second signalizes the end of an operation. Arc inscriptions are used to realize the selection of the right token. In conjunction with the flow of control (modelled by the existence of tokens), values can be passed to and from the operations. They serve as input and output parameters.

In addition, all created objects must be distinguishable. The concept of *unique identifiers* is used for this purpose: an identifier will be attached to each data object. To realize the intended data abstraction and data encapsulation, the actual data items belonging to concrete data objects reside in the interior of the module, only the identifiers of concrete data objects are known outside the module. These identifiers have to be used as further parameters for all operations except the creation operation.

Due to the use of identifiers, different concrete data objects can not influence each other. This quality allows the concurrent execution of operations for different concrete data objects, known as *inter object concurrency* in contrast to *intra object concurrency*. Both kinds of concurrencies are supported by the presented modelling technique. This does not apply for most of the object oriented programming languages.

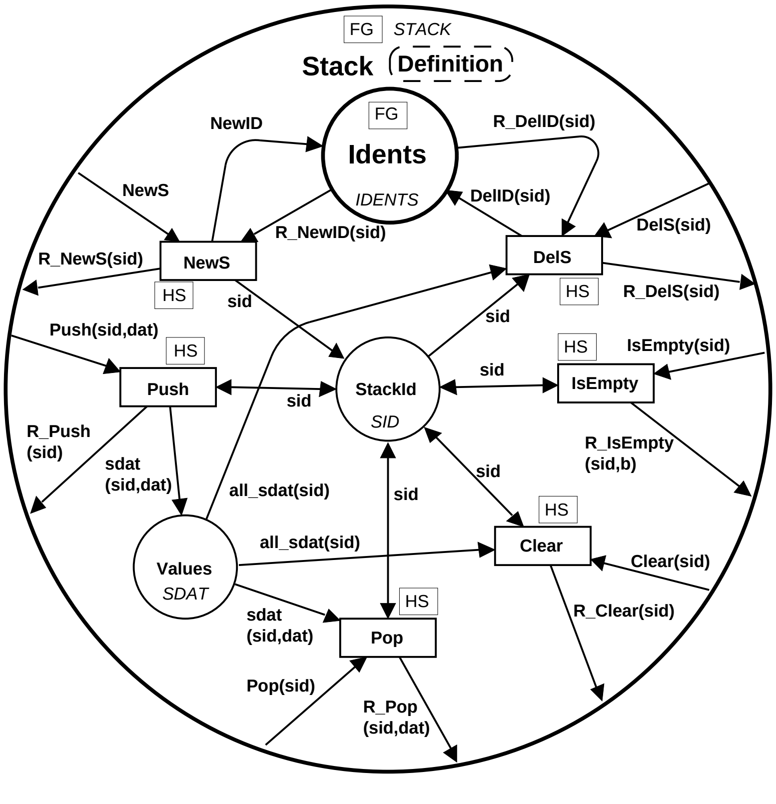

Figure 2: Definition net of an abstract data type

In figure 2 you can see the net representing the definition part of the abstract data type Stack, using a high level Petri net. The six operations are represented by six named transitions, the interface place is drawn *surrounding* the transitions. The names of the transitions are repeated in the inscriptions of arcs directed from the interface place to a transition, since they represent the triggering of an operation. The inscriptions of arcs directed from a transition to the interface place are prefixed by R_ to denote the return value.

The inscriptions of the arcs connected to the interface place use the syntax of CPN-ML for the type constructor union (which corresponds to the conventional variant record). They can be regarded as *selection* or *naming functions*. Only tokens with a named type (or call it *name*) matching with an arc inscription are accessible to that arc. It is important to realize that no calculating function is called when evaluating the arc inscriptions, even if parameters are given. These parameters are used in or are calculated by the operations of the data module, represented by the transitions.

In the definition part of a module only the existence of operations and their parameters should be shown. The implementation details should be treated in a later step. Therefore, all exported operations are represented by transitions, which must be refined by the implementation net. This refinement is indicated by the annotation "HS", which stands for *hierarchy substitution*.

To create a new stack, a token named NewS has to be put on the interface place of the stack. The result of the operation will be a token named R_NewS, which contains the unique identifier. The definition net highlights that the creation will be done with the help of the imported operation NewID of the module Idents. A very similar solution manages the destruction of a stack. It is not necessary to know how these imported operations are implemented in detail.

The restriction while using the data module is shown directly in the net: the transition Pop cannot be activated until the transition Push has occurred at least once, putting the value on the place Values. This direct modelling is a great advantage compared to the code fragment shown in figure 1.

Additionally, another restriction prevents concurrent execution of operations. Therefore, all transitions need to access a token on the common place StackId. This restriction exists implicitly in the code fragment, too. It is hidden in the sequential execution assumption. The restriction is shown explicitly in the net because of the potential concurrency in Petri nets. It should be noted, that each stack has its own identifier on the place StackId, permitting the concurrent execution of operations for different stacks.

3 The use of an abstract data type

To use the net representing an abstract data type it must be imported in some other net. This import has to make available the interface place of the module. An operation of the module is triggered by putting the appropriate input token on the interface place. Afterwards, the net using the operation has to wait until the operation is finished, i. e. until the corresponding output token is contained on the interface place. When using an abstract data type, it is necessary to create a new data object first. Thereafter, the identifier of the new stack is known outside the encapsulated module. This identifier is used to trigger any access operation and to synchronize the application with the response.

An example for the use of an abstract data type is shown in figure 3. The purpose of the application is to create a stack and to push an arbitrary number of integer values onto it. After the pushing is finished, the stack is checked for emptiness. As long as it is not empty, its values are popped and queued in a list. This queueing is done with the ::-constructor of CPN-ML which stands for the conventional cons of functional languages. It is possible to manage more than one data object in one application, since the data objects are distinguishable. Looking at the example this means that no problems arise, if two (or more) tokens are lying on the place Start[1].

For both tokens the transition Create is activated independently. When firing the transition, a token with name NewS is put on the place Stack together with a black token on the place Creating. The former token triggers the operation NewS of the stack, while the latter token is used to make the enabling of the transition Wait1 possible. This is the way to remember that the operation NewS was called and that the application is waiting for the response. Simultaneously, the place Creating is used to inhibit the transition Wait1 from consuming a response token belonging to some other call.

The place for the black token mentioned in the previous paragraph and the places for the identifiers represent the progress of execution in the application for each stack. To highlight this flow of control, bold arrows are used in the figure.

[1]Note that a lacking type inscription denotes the trivial type of black tokens. If no arc expression is given, one black token has to be used. The standard inscriptions in CPN-ML are hidden (unit colourset and () token)

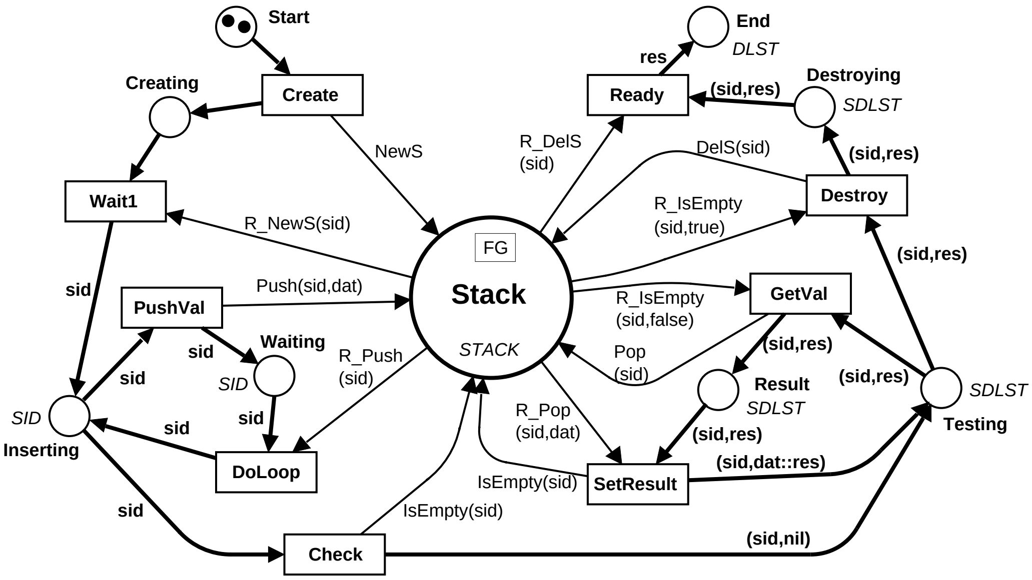

Figure 3: The use of an abstract data type

4 Implementation part of an abstract data type

When implementing an abstract data type, the main task is to implement all the operations which are defined in the definition part of the module. In general the implementation part consists of a hierarchical Petri net. In the top level Petri net of the implementation part (the *implementation net*), all places and transitions of the definition net are repeated. This redundancy is intended, since the implementation net should show more details of the whole module. To be precise, it may introduce some more places and may refine some arc inscriptions, but must not set up new transitions. The new places are used to store internal data.

To carry out the implementation of the operations, the transitions must be extended with functionality. This extension can be as simple as the introduction of a guard or the refinement of arc inscriptions. Or the extension can be done by refining the transition through a subnet, using the usual substitution. This refinement results in a hierarchical Petri net for the implementation part.

The implementation net of the stack example is shown in figure 4. When comparing it to the definition net shown in figure 2, the repetition of all places and transitions is striking. The only new place is used to store the value of the current stack pointer. This pointer is initialized with a value of zero during creation time. Every data item which is pushed onto the stack is tagged with the value of the stack pointer. In this way, all the items are ordered. The stack pointer is updated with every push or pop operation and it is used to test if a stack is empty or not. Be aware that all tokens used to implement the abstract data type are marked with an identifier to make the distinction of different data objects possible.

All the other places in the implementation net are repetitions of places already shown in the definition net. Speaking in terms of coloured Petri nets, these places are *port places*, serving as interfaces in hierarchical Petri nets. They are annotated with a "P". A fusion between the places shown in the definition net and the port places of the implementation net realizes the connection between both nets.

Like the places, all the transitions of the definition net are shown in the implementation net, too. Some of the operations represented by the transitions can be implemented completely in the implementation net. This is true for the less complex operations Push, Pop and IsEmpty. It is sufficient to refine some arc inscriptions (to make explicit the subcomponents of a variable which is compounded) and to introduce some more arcs. The implementation of these operations can be derived from the

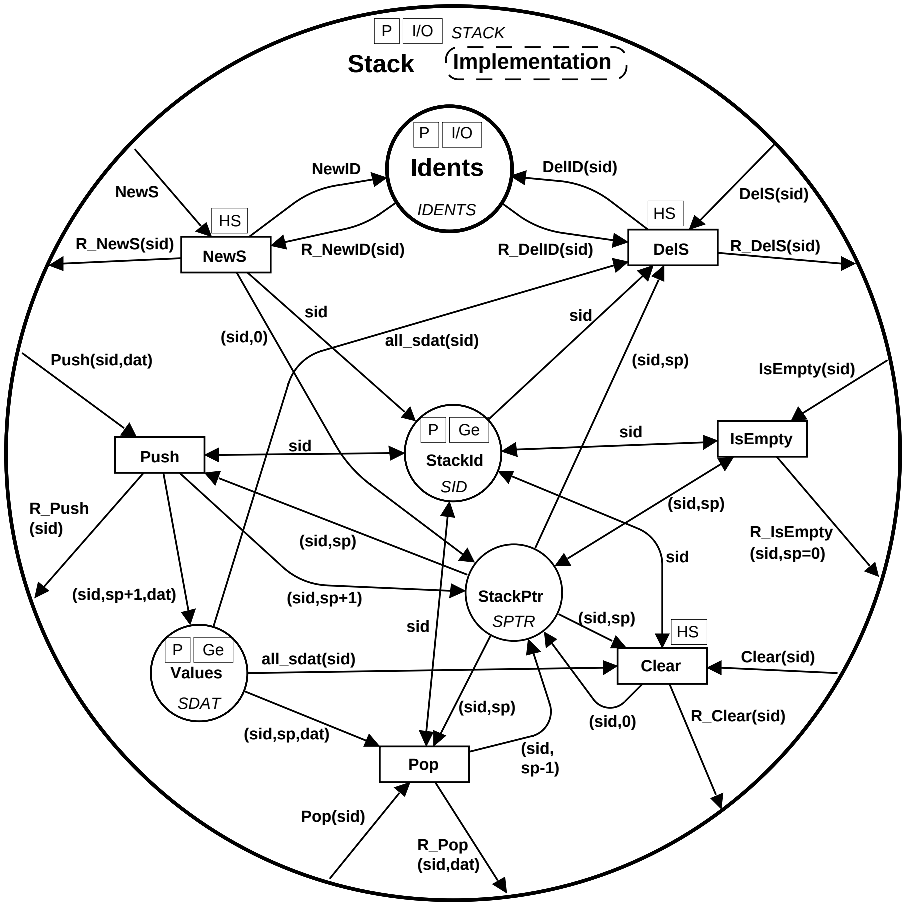

Figure 4: Implementation net of an abstract data type

Petri net in figure 4.

All other operations need at least two transitions for their implementation, because they either need to communicate with other modules or they need to solve a complex task. These operations are implemented by additional nets, which serve as refinements for the corresponding transitions of the implementation net.

The net representing the creation of a new stack is shown in figure 5. The creation of a new identifier is delegated to the module Idents. This module is accessed via a port place, implying that the module has to be imported in the upper nets. After it has been created, the new identifier is stored in the place StackId which is used by all other operations to prevent concurrent execution on *one* stack. The stack pointer of the new stack is created with a value of zero and is put on the place StackPtr. Together with these two tokens the response of the operation is put back on the place Stack.

The Petri net implementing the destruction of a stack is a little bit more complex. It is presented in figure 6. The net contains three transitions, two concrete ones and one abstract one, which is refined by yet another net. Some interesting modelling qualities of the net should be mentioned here.

The main idea behind the implementation of the operation DelS is to use the operation Clear. One obvious solution would have been to trigger this operation by writing the corresponding token on the interface place Stack and to wait for the response. But this solution would imply that another operation for the same stack object can be triggered, because the identifier has to reside on the place StackId, violating the causalities shown in the definition part of the stack (see figure 2).

For that reason another solution is needed. The implementation subnet of the operation Clear (see figure 7) is embedded in a different environment compared to the implementation net in figure 4. To do this, the local transition Clear is refined by the same implementation net. The environment

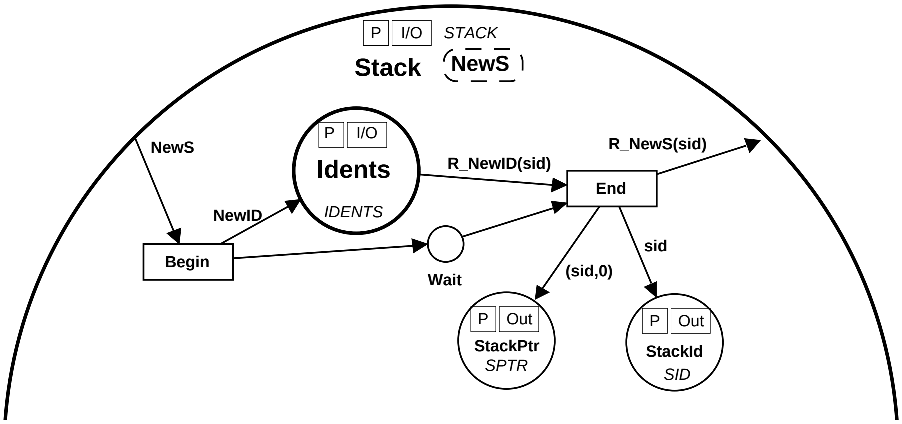

Figure 5: Implementation subnet of the operation NewS

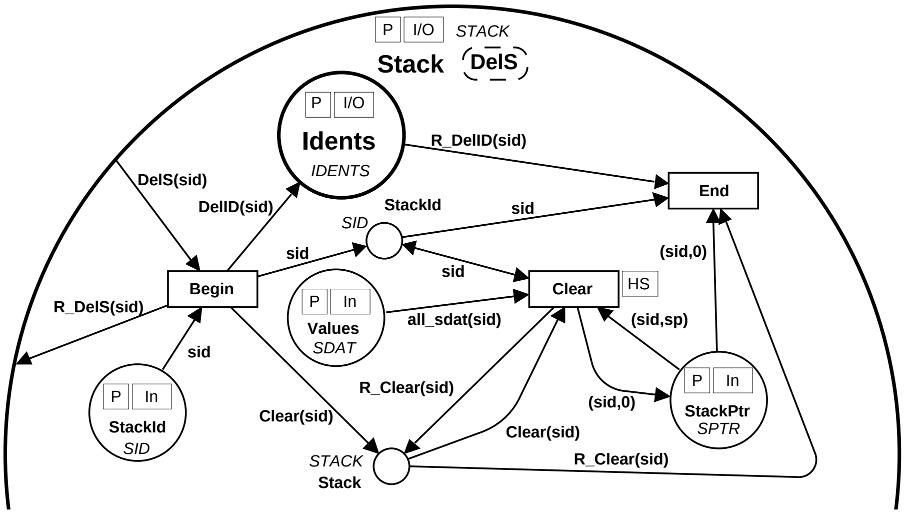

Figure 6: Implementation subnet of the operation DelS

of this transition consists of a local place holding the identifier and a local place which serves as communication place. The other two places connected with the new transition are port places which are fused with the corresponding places in the upper net.

The transition Begin represents the first step of the destruction of a stack. It moves the identifier from the port place StackId to the mentioned local place (also called StackId). By this, no other operation can reach the identifier any more, meaning that no other operation for the current stack object can be triggered. Therefore it is possible to create the response to the operation DelS immediately, although the operation itself is not yet finished. At the same time, the transition calls the operation Clear of the stack module and the operation DelID of the identifier module by putting the corresponding tokens on the appropriate interface places.

The transition End waits for the responses of the two called operations. If received, the local copy of the identifier is deleted together with the stack pointer. Provided that the operation Clear is implemented correctly, all tokens related to the actual stack object are deleted up to this point.

The last net belonging to the stack module is the implementation net of the operation Clear. It is shown in figure 7. As in the net before, the identifier is moved to a local place to prevent the triggering of other operations. Then the stored values are deleted one after the other, while decreasing the stack pointer. Note that the stack pointer cannot reach a value less than zero, because the number of tokens on the place Values, which are tagged with the current identifier, is equal to the value of the

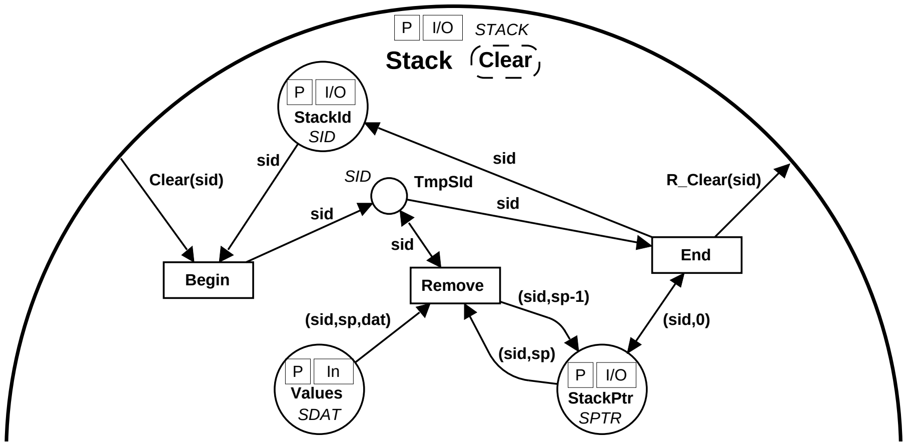

Figure 7: Implementation subnet of the operation Clear

stack pointer. After all tokens have been deleted the stack pointer has reached the value zero and the transition End can fire, putting back the identifier and creating the response of the operation.

To make the example complete, the declaration of the used colours and the variables is presented in figure 8. The used language is `CPN-ML`. I have refused to show the Petri nets of the module Idents due to space limits. They can be found in [10]. The only important quality of the module is that unique identifiers are produced. Due to the modularity it is really unimportant, how the indentifiers are created or look like. The example of the abstract data type Stack has been syntax checked and simulated with the tool Design/CPN.

```
(* General purpose colours: *)
color TOK  = unit with Tok;
color INT  = int;
color BOOL = bool;

(* Data type for stack items: *)
color DATA = int;

(* Colours for the ADT IDENTS: *)
color ID = int;
color IDENTS = union
  NewID     + R_NewID: ID +
  DelID: ID + R_DelID: ID;

(* Colours for the ADT STACK: *)
color SID = ID;
color SDATA = product SID * DATA;
color SBOOL = product SID * BOOL;
color STACK = union
  NewS            + R_NewS:    SID +
  DelS:    SID    + R_DelS:    SID +
  Push:    SDATA  + R_Push:    SID +
  Pop:     SID    + R_Pop:     SDATA +
  IsEmpty: SID    + R_IsEmpty: SBOOL +
  Clear:   SID    + R_Clear:   SID;
color IDX = int;
color SPTR  = product SID * IDX;
color SDAT  = product SID * IDX * DATA;

(* Colours for the example application: *)
color DLST = list DATA;
color SDLST = product SID * DLST;

(* Variables: *)
var id: ID; var sid: SID; var dat: DATA; var b: BOOL; var sp: IDX;
var res: DLST;

(* Some dummy functions (for abstract inscriptions only): *)
fun sdat(sid:SID,dat:DATA) = (0,0,0):SDAT;
fun all_sdat(sid:SID) = 1'(0,0,0):SDAT ms;
```

Figure 8: Declaration node of the CPN example

5 Discussion of the used hierarchy constructs

As mentioned earlier the various modules representing abstract data types must be related in some way to build up a hierarchical Petri net. That applies to the different nets belonging to one module, too. The relations between the Petri nets belonging to the stack example are shown in figure 9. Different hierarchy constructs are used for the different relations, represented by different kinds of arcs.

As discussed in the example the *import* of an abstract data type is realized with the hierarchy construct *place fusion*. This place fusion is used to connect the modules via communication places. Unlike the usual place fusion of coloured Petri nets (as defined in [11]), only restricted access to such a communication place is allowed:

- The module itself may only *read* tokens which are used to trigger an operation and it has to *write* tokens which represent the response of a triggered operation from or into the communication place.

- A module or application using another module may only *write* triggering tokens and has to *read* corresponding response tokens into or from the communication place.

It would be worthwhile to formally define such a restricted access to fusion places. It is called ***Uses-Relation***, since this hierarchy construct expresses the *use* of other modules.

Another kind of hierarchy construct is used to connect the definition net and the implementation net of a module. It is called ***group-substitution***, since a group of transitions is refined by *one* net. It is important to see the difference between this kind of substitution and the conventional transition substitution used in coloured Petri nets. When refining two transitions by the same net using the conventional substitution, two distinct instances of this net are created. This is not true in the case of group substitution, where only *one* instance is created.

A formal definition of the group substitution should be very similar to the formal definition of the conventional substitution. One has to consider a set of transitions instead of only one transition. The connection between the related nets must be defined according to the places surrounding the whole group and the port places of the subnet.[2]

The last utilized hierarchy construct is the conventional substitution of transition. It is used for the refinement of complex operations, connecting an abstract transition of the implementation net with the corresponding implementation subnet.

In figure 9, the hierarchical arrangement of the modules building up the example is shown. The hierarchy consists of three main parts: one application and two modules. The application is called Example. It imports the module Stack. The latter module imports another module, called Idents, which is used as server for unique identifiers. The three parts are connected by two uses-relations, represented by dotted arcs. The arc inscriptions name the used operations.

Both of the two modules consist of two subparts: the definition net and the implementation net. In each module the definition net and the implementation net are connected by group-substitution, respectively. These connections are drawn as dashed arcs, annotated with an arc inscription, which names all substituted transitions.

The implementation net of the module Stack contains three subnets for the realization of the more complex operations, while the module Idents can be implemented without substituting transitions. The conventional substitutions are represented as solid arcs in the figure.

---

[2]By the way, to simulate the presented example with the tool `Design/CPN`, the group substitution had to be expressed in terms of the normal substitution. This was done by a trick, not visible in the net representation: each port place of the implementation net is duplicated without connecting them to any transition. Only one transition of the definition net is refined with respect to the visible port places. All the other transitions utilize the invisible port places. The result is, that only in one instance of the implementation net the tokens on port places can be used by transitions.

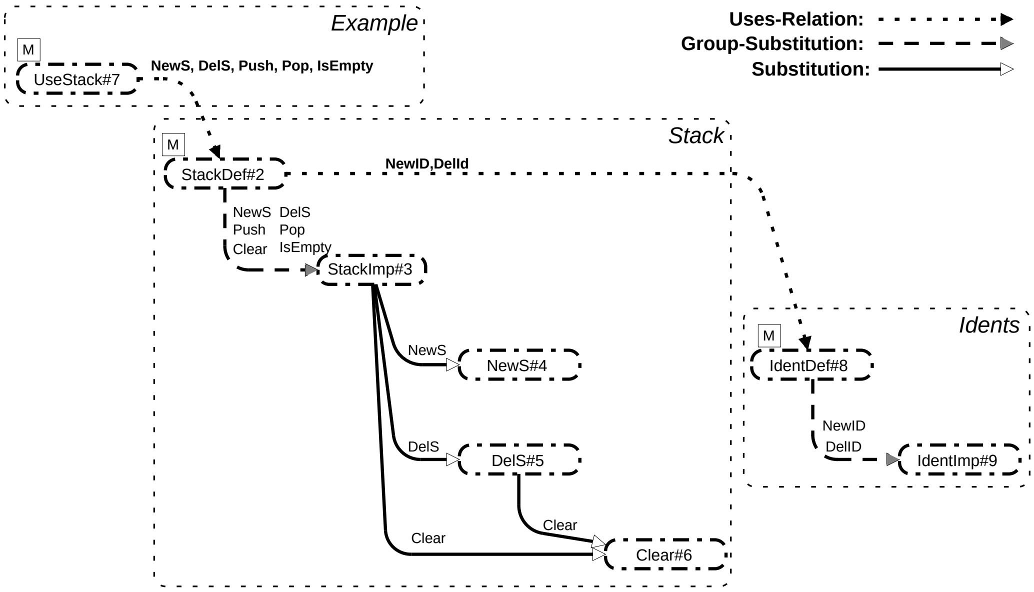

Figure 9: Hierarchical arrangement of the modules

6 Conclusion

This paper presented a new approach for modular Petri nets. The main idea was to apply the concepts of data encapsulation and data abstraction to Petri nets. The introduced modelling technique *encloses* data objects inside a high level net structure and makes available identifiers for reaching and operations for manipulating the data objects. The graphical notation supports the visualization of modules, because the net structure of a module is *surrounded* by a single interface place, building up a visual border of a module.

The concept of identifiers allows to distinguish different data objects, which reside inside one module. This supports the dynamical creation and deletion of data objects, while all of the data objects use the same Petri nets for their functionality. The dynamical behaviour of a module makes strong use of high level Petri nets. When mapping to low level Petri nets, a dynamical modification of Petri nets would be necessary, because the dynamical creation of a new data object would imply the dynamical extension of the low level Petri net.

Another topic was to increase reuse and maintainability of Petri net modules. Therefore the idea of separating definition and implementation part of modules was adapted to Petri nets, too. The result was the introduction of a *definition net*, determining the export and import interface of a module, and an *implementation net*, realizing the functionality of the module. This separation supports abstraction, since only the interface must be known to use a module. It makes it possible to change the implementation part as long as the definition part is unchanged. Additionally, the reuse of Petri nets is increased, because modules can establish a library.

The connection between definition and implementation net was realized by a new hierarchy construct, called *group substitution*. The idea was to refine a set of transitions by one instance of one subnet. This was done to illustrate the abstract view of the definition net. Another hierarchy construct was introduced, called *uses relation*, making explicit the use of one module by another.

References

[1] E. Battiston, F. de Cindio, and G. Mauri. OBJSA nets: A class of high-level nets having objects as domains. In G. Rozenberg, editor, "Application and Theory of Petri Nets, 8th International Conference, Saragoza, Spain", number 340 in LNCS, pages 20–43. Springer-Verlag (1988).

[2] E. Battiston, F. de Cindio, G. Mauri, and L. Rapanotti. Morphisms and minimal models for OBJSA nets. In “Application and Theory of Petri Nets, 12th International Conference, Gjern, Denmark”, pages 455–476. University of Aarhus, IBM Deutschland (June 1991).

[3] Ulrich Becker and Daniel Moldt. Object-Oriented Concepts for Coloured Petri Nets. In IEEE, editor, “Conference Proceedings, IEEE International Conference on Systems, Man and Cybernetics”, volume 3, pages 279–286, Le Touquet, Frankreich (17.–20. October 1993). IEEE.

[4] Didier Buchs and Nicolas Guelfi. CO-OPN: A Concurrent Object Oriented Petri Net Approach. In “Application and Theory of Petri Nets, 12th International Conference, Gjern, Denmark”, pages 432–454. University of Aarhus, IBM Deutschland (June 1991).

[5] Søren Cristensen and Laure Petrucci. Towards a Modular Analysis of Coloured Petri Nets. In K. Jensen, editor, “Application and Theory of Petri Nets, 13th International Conference, Sheffield, UK”, number 616 in LNCS, pages 113–133, Berlin Heidelberg (June 22–26 1992). Springer Verlag.

[6] Yi Deng, S. K. Chang, Jorge C. A. de Figueired, and Angelo Perkusich. Integrating Software Engineering Methods and Petri Nets for the Specification and Prototyping of Complex Information Systems. In Marco Ajmone Marsan, editor, “Application and Theory of Petri Nets, 14th International Conference, Chicago, Illinois, USA”, number 691 in LNCS, pages 206–223, Berlin Heidelberg (June 21–25 1993). Springer Verlag.

[7] H. Ehrig and B. Mahr. “Fundamentals of Algebraic Specifications 2: Module Specifications and Constraints”, volume 21 of “EATCS Monographs on Theoretical Computer Science”. Springer-Verlag, Berlin, Heidelberg (1990).

[8] H. Ehrig, J. Padberg, and L. Ribeiro. Algebraic High-Level Nets: Petri Nets Revisited. In “Recent Trends in Data Type Specification”, number 785 in LNCS, pages 188–206. Springer-Verlag (1994).

[9] Sarah English. “Coloured Petri Nets for object-oriented modelling”. PhD thesis, University of Brighton, England, Brighton, Sussex (June 1993).

[10] Olaf Fricke. Data Encapsulation and Data Abstraction with Petri Nets – a graphical Visualization of Modules. In H. Ehrig, W. Reisig, and H. Weber, editors, “Move-On-Workshop der DFG-Forschergruppe Petrinetz-Technologie, Berlin, 3. und 4. April 1997”, Forschungsbericht des Fachbereichs Informatik, Franklinstr. 28/29, D-10587 Berlin (planned for September 1997). Technical University Berlin.

[11] Kurt Jensen. “Coloured Petri Nets: Volume 1; Basic Concepts, Analysis Methods and Practical Use”. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin Heidelberg New York (1992).

[12] Ekkart Kindler and Wolfgang Reisig. Algebraic System Nets for Modelling Distributed Algorithms. In Hartmut Ehrig, Wolfgang Reisig, and Herbert Weber, editors, “Kick-Off-Workshop der DFG-Forschergruppe Petrinetz-Technologie, Berlin, 15. July 1996”, number 73 in Informatik-Bericht, pages 56–71. Humboldt-Universität zu Berlin (November 1996).

[13] Charles Lakos. Object Petri nets – definition and relationship to coloured Petri nets. Technical Report R94-3, Department of Computer Science, University of Tasmania, Dep. Comp. Sc., GPO Box 252C Hobart Tasmania 7001 (April 1994).

[14] Charles Lakos. From coloured petri nets to object petri nets. In Jonathan Billington and Michel Diaz, editors, “Workshop on Petri Nets applied to Protocols within the 16th ICATPN”, Turin, Italy (June 28 1995). University of Torino.

[15] Charles Lakos. The object orientation of object petri nets. In Gul Agha and Fiorella De Cindio, editors, “1st Workshop on Object-Oriented Programming and Models of Concurrency within the 16th ICATPN”, Turin, Italy (June 27 1995). University of Torino.

[16] Meta Software Corporation, Cambridge, MA, USA. “Design / CPN Handbook Version 2.0” (1993).

[17] Angelo Perkusich and Jorge C. A. de Figueiredo. Modeling a Multi-Agent Cooperative Graphical Editor with a Class of Object Oriented Petri Net. In Gul Agha and Fiorella De Cindio, editors, “The Second Workshop on Object-Oriented Programming and Models of Concurrency during the 17th ICATPN”, Osaka, Japan (June 24 1996). University of Torino.

[18] University of Aarhus, DK-8000 Aarhus C, Denmark. “Design / CPN Online Manuals Version 3.0.1” (1996). http://www.daimi.aau.dk/designCPN/man/.