(Draft) Lecture Notes in Computer Science (LNCS). 2001, vol. 2001, p. 355-374. ISSN 0302-9743.

KRON: Knowledge engineering approach based on the integration of CPNs with objects*

J. A. Bañares, P. R. Muro-Medrano and J. L. Villarroel

Departamento de Informática e Ingeniería de Sistemas
UNIVERSIDAD DE ZARAGOZA
Maria de Luna 3, Zaragoza 50015, Spain

Abstract. This paper presents KRON (Knowledge Representation Oriented Nets), a knowledge representation schema for discrete event systems (DESs). KRON enables the representation and use of a variety of knowledge about a DES static structure, and its dynamic states and behavior. It is based on the integration of Colored Petri nets with frame based representation techniques and follows the object oriented paradigm. The main objective considered in its definition is to obtain a comprehensive and powerful representation model for data and control, and to incorporate a powerful modeling methodology. The communication model used in KRON is close to the generative communication model, which supposes an alternative to message passing. The inferences delivered from the DES behavioral knowledge are governed by a control mechanism based on a rule inference engine.

keywords: Colored Petri nets, frames, knowledge engineering, DES.

1 Introduction

This paper is devoted to illustrate the main features involved in KRON (Knowledge Representation Oriented Nets). We starting creating KRON while we were working in the development of knowledge based models for DESs. It became clear in working with DESs the need to expand the power of our knowledge engineering representation schema with the integration of an adequate formalism to deal with discrete event system features.

A lot of integrations of Petri nets with different paradigms can be found in technical literature. These may be split into three main groups: 1) Extension of Petri nets with primitives to support methodological aspects (modularity, top-down and bottom-up design, ...); 2) Integration of Petri nets with algebraic specifications and 3) Integration of Petri nets with the frame/object paradigm. Several workshops about the integration of Petri Nets and objects are held regularly as part of prestigious conferences (Int. Conf. of Application and Theory of Petri Nets, IEEE Int. Conf. on Systems Man and Cybernetics, ...), this is a proof of the growing interest in this topic.

* This work was supported by the Spanish Interministerial Comission of Science and Technology (CICYT) under project TAP95-0574.

A HLPN extension belonging to the first group is HCPN (Hierarchical Colored Petri Net). HCPNs [HJS89] provide a set of constructs to support modularity aspects. The idea behind HCPNs is to allow the construction of a large model by combining a number of small HLPNs into a larger net, and different structuring tools are proposed with this purpose. Posterior proposals extending HLPNs with structuring constructs can be found in [Feh91] and [CH94]. Other works that propose different object oriented interpretations of HCPN constructs can also be found in [Lak93] and [Eng93].

The presentation of the most representative works on the PN integration with algebraic specifications (second group) can be briefly summarized as follows: Algebraic Nets [Vau87], Many-sorted High-level Nets [Bil89] and Petri Nets with structured tokens [Rei91] are a result of the integration of HLPNs (used to describe the control structure of the system) and algebraic specifications (used to describe the data structure). These previous works have been the basis of many others, most of them also considering some object oriented focus. OBJSA Nets [BdCM88] and CO-OPN (Concurrent Object-Oriented Petri Nets) [BG91] are good examples. Its goal is to allow data abstraction and introduce net modularity.

Finally, the third group are the approaches based on a frame/object approach. From an engineering point of view, we consider them closer to human conceptual thinking than the ones based on algebraic specifications. What is required here, is a conceptual model which will enable engineers and computer scientists to describe domain concepts in a more intuitive way. Examples of this group are:

- In [DG87] high level Petri nets are integrated with the Entity-Relationship model to obtain the EER formalism. This model is revised in [DGV91] incorporating object oriented concepts to increase expressiveness in data modeling. However, this approach is not extended to the process structure in order to provide an overall modeling framework. Finally, a second revision is done in [BDLGV95]. In this last piece of work the internal behavior of each object is described by means of a Petri net (O-net). To obtain the global process structure partial nets are synchronized by another Petri net, the P-net. This P-net is not included in the object structure.
- Object Petri nets (PNO), which have been widely referred to in technical literature, were defined in [SB85] as High Level Petri Nets with Data Structures. Their objective is to incorporate the data modeling and updating into the net model by means of frame-like data structures. Starting from this seminal work, in [PR93] HOOD/PNO is proposed as a software engineering methodology that integrates PNO with the HOOD. In [SB94] two more extensions to PNO were introduced: Communicative and Cooperative nets. They enable the modeling of a system as a collection of nets that encapsulate their behavior while interacting by means of message sending and the client/server protocol.
- [BB91] presents PROTOB, an object oriented language and methodology based on PROT nets [BM86]. In this object oriented approach, objects com-

municate by message passing and a hierarchical object decomposition like HOOD is allowed. However neither inheritance nor data representation aspects are considered.

– LOOPN++ [LK94] has mainly been used to describe network protocols. LOOPN is a textual language that supports object oriented structuring into HLPNs. The language has a formal semantics which makes it possible to transform OP-nets (Object Petri nets) into the simpler HLPN formalism. However, as it has been pointed out by the author, there is not a precise relation between OP-nets and the LOOPN++ language.

As the reader can see, there are a lot of integrated models. Most of the previous approaches concentrate on providing structuring tools in compliance with software engineering principles, by enforcing constraints that may result in a loss of freedom and flexibility. Most of them also extend the formalism of HLPNs. However, there is a great scope for further work in tailoring analysis techniques to extended HLPNs.

KRON is based on the integration of Colored Petri nets (CPNs) with the frame/object oriented paradigm. The integration model presented in this paper provides a close integration of HLPNs and the object model, and it does not extend the HLPN formalism. Frames and rules have been selected as a basis to support the representation aspects due to its power for knowledge representation. Additionally, we improved programming discipline by following an object oriented methodology obtaining important methodological advantages such as: 1) it supports conceptual models closer to human conceptualization and independent from implementation, thus the models are easier to understand; 2) it facilitates reusability and model extensibility based on encapsulation and inheritance characteristics.

The rest of the paper is organized as follows. Firstly, a brief presentation of KRON constructs that support the CPN formalism is presented. In the following sections, the case study of hurried philosophers is used to illustrate KRON. Section 4 shows the definition of dynamic and no dynamic entities and their relationships. Section 5 presents inheritance as a mechanism to share code. The communication model is presented in section 6. The paper finishes with a conclusions section.

2 KRON constructs

Knowledge representation of DESs must involve the representation of information related to its dynamic behavior as well as more static information. From a conceptual point of view, the representation of a KRON model is based on semantic networks, whereas a frame implementation perspective has been adopted for its programming. In this programming context, the representation is structured around a set of conceptual entities with associated descriptions and interconnected by various kinds of associative links. However, in frame based representations, little attention has been paid to describing the coordination

(Draft) Lecture Notes in Computer Science (LNCS). 2001, vol. 2001, p. 355-374. ISSN 0302-9743.

between objects in order to achieve collective behavior [FK85]. The application of frame/object based languages to the modeling of complex dynamic systems, has certain inconveniences due to the lack of a formalism to specify its dynamic behavior (concerning both, the states of the objects and the causal relationships between states and actions).

In addition to the programming features supported by frame/object oriented languages, our knowledge representation schema includes a set of primitives implementing the CPN formalism. CPNs provide the mechanism to describe the internal behavior of the dynamic entities and the interaction between them, with no necessity for a low level communication model. Figure 1 illustrates the frame hierarchy The KRON hierarchy can be decomposed into three important groups:

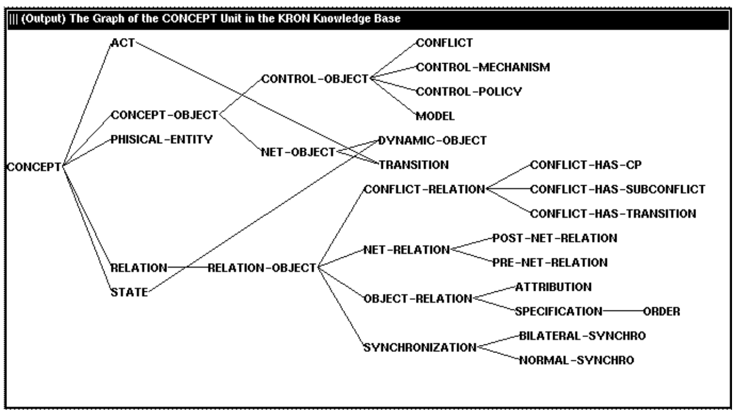


Fig. 1. KRON hierarchy.

1. **Net objects**. Dynamic entities in KRON are descendants of a specialized object called dynamic object. A dynamic object centralizes all the information related to a dynamic entity (abstract or real), and it is the repository of information about the entity states and activities. The behavioral description of a dynamic object class is represented by a CPN. The state will be mapped in a combination of CPN places and structured tokens, whereas the activities that produce state changes will be mapped in CPN transitions. The constitutive elements of the structure of a CPN are represented by individual concepts and dedicated object slots (Transitions, activity slots and state slots), which are aggregated or composed in dynamic objects to represent the behavior:
   - The state of a dynamic entity is represented by a set of state slots. To each state slot corresponds a single place of the CPN. State information in a CPN is represented by its marking, this means the places and the tokens located in the places. Tokens, which evolve by a CPN, are not mapped

onto specialized objects in KRON. Any entity evolving through state slots plays the role of a *token*. The state of a dynamic entity is defined not only by the marking relations, but also by the token attributes (slots) that are relevant for that state. Structured tokens allow KRON to benefit from some CPN advantages like the aggregation of dynamic information to obtain more concise models.

   – Activities producing state changes in a dynamic object are represented by transitions, and they are equivalent to the transitions of a CPN. Transitions that represent activities related to the same dynamic object are located in its `activity slots`. The interface of a KRON dynamic object is a subset of activity slots that hold transitions representing activities that must be carried out in cooperation with other dynamic objects (see section 6). In this way, transitions also provide information about the set of applicable services for the current state.

   Finally, CPNs of dynamic objects themselves can be aggregated to create more complex nets in a high level structure called model, which describes the collective behavior.

2. **Relations**. Relations hold the information of interdependent KRON objects. KRON allows the definition of relations as an important concept at the same level as classes or objects. Generic relations are defined as a specialization of relation-object. When a relation is defined between two classes, a slot is created in the first class with the name of the relation, and another slot is created in the second class with the name of the inverse relation. Demons attached to these slots are responsible for making automatic updating of direct-inverse relations. From the CPN point of view, relations make possible the combination of objects in more complex data structures that represent tokens.
   KRON also provides specific relations related to the description of dynamic behavior:
   – Net relations support CPN arcs and expressions labeling them, and are used to specify connections between state slots and transitions. The information about net relations is stored in transitions.
   – Synchronization relations provide a simple way to specify interconnection between dynamic entities, which is done by means of the synchronization of activities in the activity slots that constitute the interfaces of dynamic objects.
3. **Control objects**. These objects provide the mechanisms and policies used to implement the evolution rules of the underlying CPN (*token player* in Petri net terminology and *inference engine* in the knowledge representation terminology). The search for enabled transitions is carried out by an efficient matching algorithm [BMMV93].
   A KRON model can be not fully deterministic, that is, there exist points in which decisions have to be taken in order to establish the model evolution. For the selection phase, transitions are grouped into conflicts by inspecting the net structure, and each one is provided with a particular control policy.

Conflicts may also be related in order to provide them with a control policy. Conflicts enable us to establish a simple interface between the model and a decision making system.
The interpretation of a model is carried out by the control-mechanism, which applies the corresponding control-policy to each conflict located in the model.

3 Relations to CPNs

The Petri net underlying a KRON model can be considered as a subset of a CPN with a special syntax. However, there still exist restrictions that are introduced to improve the modeling and simulation capabilities of CPNs to solve practical tasks. The formal analysis of properties was not a crucial issue in the KRON development. Our approach is closer to the work presented by Cherkasova et col. in [CKR93], which combines CPNs with modeling by direct programming, than to works that extend the CPN formalism.

Following this pragmatic approach, the CPN formalism is not extended, but really, it is constrained to use simpler expressions. A KRON net differs from CPNs (as defined in [Jen92]) in the following restriction: An Arc expression may only denote a unique token, but not set of tokens.

Another important difference with CPNs is introduced by the integration of CPNs with the object model. KRON tokens are entities, and their attribute values and relationships are considered in order to describe the system behavior. The nature of these tokens introduces a property called *ubiquity* [SB85]. Ubiquity concerns the token ability to have several occurrences in a marking. Formally, a KRON net with ubiquity is not a correct CPN because it produces the loss of the transition scope. Ubiquity produces the following undesirable effects: 1) it violates the partition and encapsulation of the state in dynamic objects. Moreover, it hides the way transitions modify the state because they have unlimited writing access to all token attributes. 2) Ubiquity is a property irreducible to algebraic analysis. This problem is not exclusive of the integration of the object model and Petri Nets. The problem arises in any representation language that allows different references (object pointers) to the same object. This property, which is known as *dynamic aliasing*, makes it difficult to prove the correctness of a system representation theoretically [Mey88]. KRON allows the modeller to decide whether to avoid ubiquity in order to prove the correctness of the system representation, or to model in a more flexible way without to worrying about the ubiquity problem.

4 A KRON model for the hurried philosophers case study

In order to illustrate the representation schema, let us focus on "The hurried Philosophers" case study [SB94]. Since the proposal allowed free interpretation of philosopher behaviors and it was originally though for a message passing communication model, we state our interpretation first:

The case study is the very well known table of philosophers, with an extension: a philosopher may leave the table as he likes it, and new guests may be introduced. In the world of philosophy there are philosophers that may be thinking and eating. Moreover, philosophers must respect some rules of politeness. The philosophers interact in order to respect these rules. A philosopher who wants to be in the world of philosophy must be introduced in a common table, and must be sit in a chair, with a philosopher on his left side, and a philosopher on his right side. Philosophers share a fork with his right and one with his left neighbor. A philosopher only may start eating if he has a fork on his left and a fork on his right. Any philosopher may decide to leave the table if he is not eating. A philosopher leaves with the fork on his left side, which must be free, and he leaves his chair. A philosopher may be introduced between two philosophers if they are thinking (the fork between them is free). The guest philosopher takes a free chair and carries a fork on his left hand. Therefore, philosophers may interact to ask and give forks, and to enter and leave the table. In the following sections this case study will be completed.

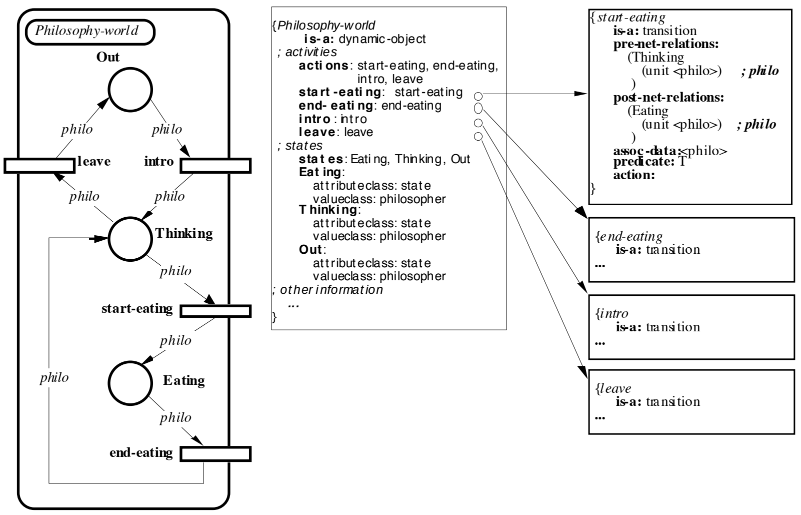
The first step in the KRON modeling methodology is the identification of the dynamic and no dynamic entities which compose the system model at the chosen abstraction level:

```
{Chair
    is-a: Phisical-Entity
  ; Relaciones
    philo:
       attributeclass: seat
       valueclass: philosopher
    left-fork:
       attributeclass: left-fork
       valueclass: fork
    right-fork:
       attributeclass: right-fork
       valueclass: fork
   ....
}

{Fork
    is-a: Phisical-Entity
  ; Relaciones
    left-chair:
       attributeclass: right-fork
       valueclass: chair
    right-chair:
       attributeclass: left-fork
       valueclass: chair
}

{Philosopher
    is-a: Phisical-Entity
  ; Relaciones
    chair:
       attributeclass: seat
       valueclass: chair
}

{ Seat
    is-a:  attribution
    domain: Chair
      slot: philo
      cardinality: 1
    range: Philosopher
      slot: chair
      cardinality: 1
   ....
}

{left-fork
    is-a:  attribution
    domain: Chair
      slot: left-fork
      cardinality: 1
    range: Fork
      slot: right-chair
      cardinality: 1
   ....
}

{ right-fork
    is-a:  attribution
    domain: Chair
      slot: right-fork
      cardinality: 1
    range: Fork
      slot: left-chair
      cardinality: 1
   ....
}
```

Fig. 2. Entities and relationships.

Token objects

In the problem description it can be identified the no dynamic entities `Chair`, `Fork` and `Philosopher`. It can also be identified the dynamic relation `Seat` representing the association between a `chair` and a `philosopher`, and the dynamic relations `left-fork`and `right-fork` representing the associations between a chair and his left and right forks. (In KRON dynamic relations are a specialization of the `attribution` relation). Figure 2 shows the frames that define these entities and relations. These entities and relations will not be considered dynamic entities from the model point of view. That means that their internal behaviors are not considered at this abstraction level, but they can complete the behavior of other entities playing the role of tokens.



```
{Philosophy-world
      is-a: dynamic-object
 ; activities
    actions: start-eating, end-eating,
             intro, leave
    start-eating:  start-eating
    end-eating: end-eating
    intro : intro
    leave: leave
 ; states
    states: Eating, Thinking, Out
    Eating:
       attributeclass: state
       valueclass: philosopher
    Thinking:
       attributeclass: state
       valueclass: philosopher
    Out:
       attributeclass: state
       valueclass: philosopher
; other information
   ...
}
```

```
{start-eating
    is-a: transition
    pre-net-relations:
       (Thinking
          (unit <philo>)     ; philo
       )
    post-net-relations:
       (Eating
          (unit <philo>)     ; philo
       )
    assoc-data:<philo>
    predicate: T
    action:
}
```

```
{end-eating
    is-a: transition
...
```

```
{intro
    is-a: transition
...
```

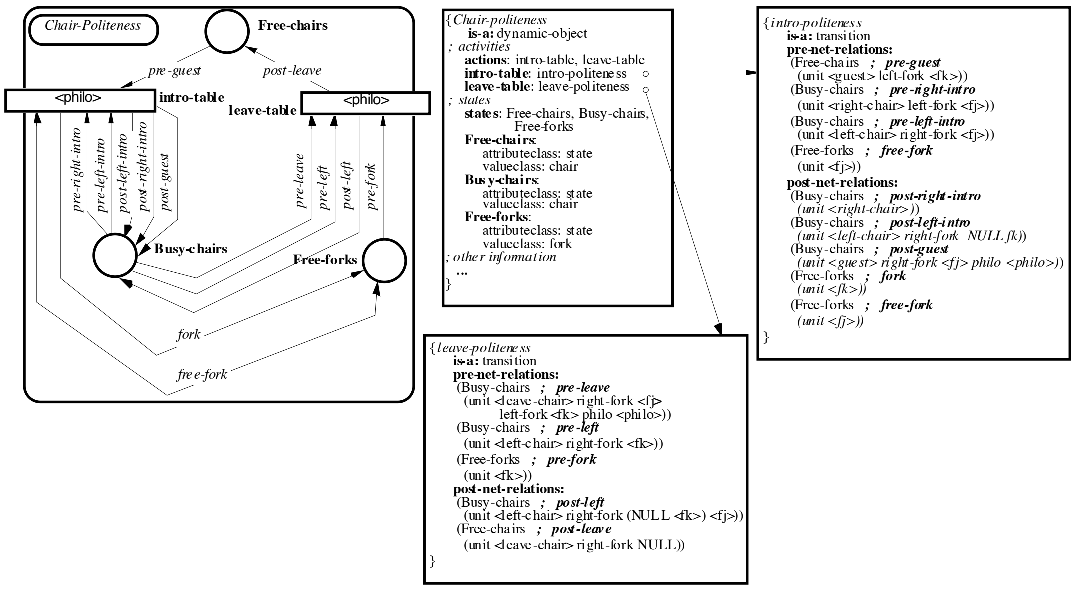
```
{leave
    is-a: transition
...
```

Fig. 3. The dynamic-object `Philosophy-world`.

Dynamic objects

It is possible to identify the following dynamic entities from the problem description. The dynamic-object `philosophy-world` represents the activities of philosophers. They may enter and leave the world of philosophy, and may be thinking and eating. Figure 3 shows the dynamic-object representing the `philosophy-world`. It has a state slot for each CPN place. Each place in a CPN has an associated set of possible tokens. In the same way, each state slot has a constraint (valueclass metaknowledge) associated to the class of objects (tokens) that it can contain. In the `philosophy-world` all state slots hold philosopher instances.

We may consider the rules of politeness incrementally. Thus, first we define the dynamic-object `Chair-Politeness`. It represents that a chair may be introduced or removed from the table considering involved forks, but it does not consider that philosophers may be eating or thinking. Figure 4 shows the `Chair-Politeness` dynamic object. The state slots `Free-chairs` and `Busy-chairs` hold `Chair` instances, and the state slot `Free-forks` hold `Fork` instances. Transitions `intro-table` and `leave-table` take into account the attributes `right-fork` and `left-fork` of `Chair` instances, and modify the relations between chairs and forks to introduce or remove a chair. The initial marking should, at least, hold



```
{Chair-politeness
    is-a: dynamic-object
; activities
   actions: intro-table, leave-table
   intro-table: intro-politeness
   leave-table: leave-politeness
; states
   states: Free-chairs, Busy-chairs,
              Free-forks
   Free-chairs:
      attributeclass: state
      valueclass: chair
   Busy-chairs:
      attributeclass: state
      valueclass: chair
   Free-forks:
      attributeclass: state
      valueclass: fork
; other infomation
  ...
}
```

```
{intro-politeness
   is-a: transition
   pre-net-relations:
    (Free-chairs  ;  pre-guest
     (unit <guest> left-fork <fk>))
    (Busy-chairs  ;  pre-right-intro
     (unit <right-chair> left-fork <fj>))
    (Busy-chairs  ;  pre-left-intro
     (unit <left-chair> right-fork <fj>))
    (Free-forks  ;  free-fork
     (unit <fj>))
   post-net-relations:
    (Busy-chairs  ;  post-right-intro
     (unit <right-chair>))
    (Busy-chairs  ;  post-left-intro
     (unit <left-chair> right-fork  NULL fk))
    (Busy-chairs  ;  post-guest
     (unit <guest> right-fork <fj> philo <philo>))
    (Free-forks  ;  fork
     (unit <fk>))
    (Free-forks  ;  free-fork
     (unit <fj>))
}
```

```
{leave-politeness
   is-a: transition
   pre-net-relations:
    (Busy-chairs  ;  pre-leave
     (unit <leave-chair> right-fork <fj>
           left-fork <fk> philo <philo>))
    (Busy-chairs  ;  pre-left
     (unit <left-chair> right-fork <fk>))
    (Free-forks  ;  pre-fork
     (unit <fk>))
   post-net-relations:
    (Busy-chairs  ;  post-left
     (unit <left-chair> right-fork (NULL <fk>) <fj>))
    (Free-chairs  ;  post-leave
     (unit <leave-chair> right-fork NULL))
}
```

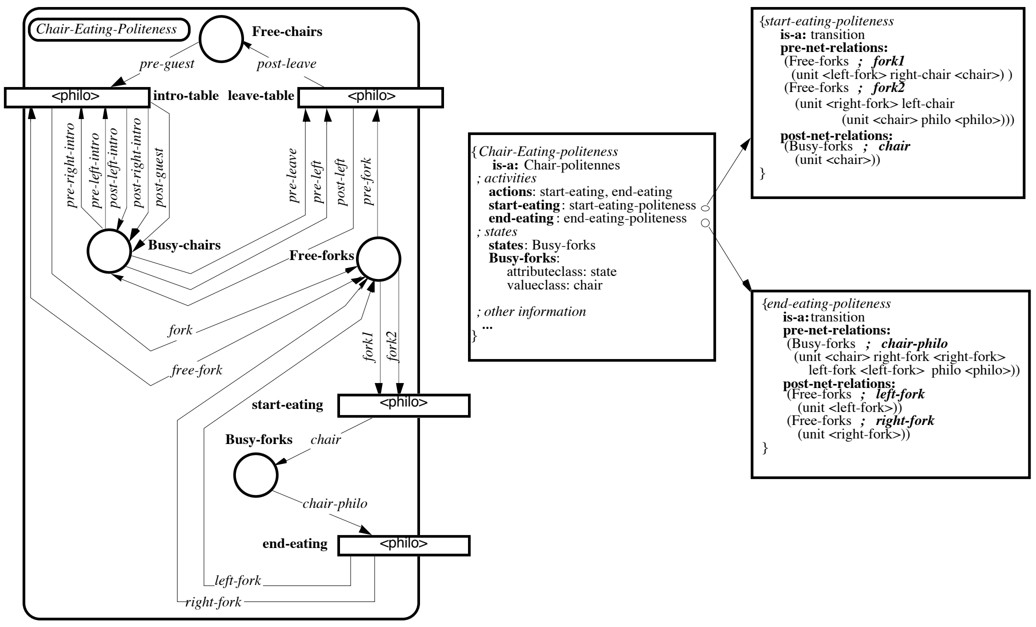
Fig. 4. The dynamic-object `Chair-Politeness`.

two busy chairs and an arbitrary number of free chairs with their corresponding left fork. Transitions are parameterized with the `<philo>` variable, which represents that the `Chair-Politeness` must be synchronized with another dynamic-object to modify relations between chairs and philosophers.

Following, the `Chair-Politeness` may be specialized to consider that philosophers may be thinking or eating. Figure 5 shows the `Chair-Eating-Politeness`, which inherits state and activity slots from `Chair-Politeness`, and adds two new activities `start-eating` and `end-eating`, and a new state slot `Busy-forks`. It represents that a philosopher only may start eating if he has a free fork on his left and another one on his right side. When a philosopher is eating the corresponding forks are removed from the place `Free-fork`. In this way, his neighbors can not start eating, and guests may not be introduced next to him.

Transitions

Let us focus in the activity descriptions of previous dynamic-objects. From a discrete event system perspective, transitions carry out the specification and semantics of CPN transitions. Petri net arc information is supported in KRON by net-relations represented by two remarkable slots of transitions: Relations from state slots to transition objects working as enabling conditions are in the pre-net-relations slot, and relations from transition objects to state slots working as causal relations are in the post-net-relations slot.



```
{Chair-Eating-politeness
   is-a: Chair-politennes
 ; activities
   actions: start-eating, end-eating
   start-eating: start-eating-politeness
   end-eating : end-eating-politeness
 ; states
   states: Busy-forks
   Busy-forks:
      attributeclass: state
      valueclass: chair

 ; other information
  ...
}
```

```
{start-eating-politeness
   is-a: transition
   pre-net-relations:
    (Free-forks  ;  fork1
     (unit <left-fork> right-chair <chair>) )
    (Free-forks  ;  fork2
      (unit <right-fork> left-chair
              (unit <chair> philo <philo>)))
   post-net-relations:
    (Busy-forks  ;  chair
      (unit <chair>))
}
```

```
{end-eating-politeness
   is-a: transition
   pre-net-relations:
    (Busy-forks  ;  chair-philo
     (unit <chair> right-fork <right-fork>
        left-fork <left-fork>  philo <philo>))
   post-net-relations:
    (Free-forks  ;  left-fork
      (unit <left-fork>))
    (Free-forks  ;  right-fork
      (unit <right-fork>))
}
```

Fig. 5. The dynamic-object `Chair-Eating-Politeness`.

From a knowledge representation perspective, information about activities can be considered as declarative knowledge in the "if/then" rule style (the similarities between CPN transitions and rules in rule based systems have been pointed out in several works [BE86], [BMMV93], [VB90]). The only difference is that in rule-based languages the enabling conditions on the left hand side of the rule (lhs), are clearly separated from the causal conditions on the right hand side (rhs). Nevertheless, the execution of a transition implies removing the enabling tokens from the input places and putting tokens in the output places according to the post-net-relations.

Expressions labeling the arcs are represented in KRON as arc expressions in pre and postconditions. An arc expression is a specification of restrictions on objects. These restrictions are represented, in a rule style, by a list of component pairs: the first component is the specification of a slot name or the string `unit` denoting an object instance; the second component, composed by one or two elements, is a partial pattern to match the slot value, it can be a variable, a specific constant value, a function or expression or another arc expression.

Following with the behavior representation of the `Philosophy-world`, the activity slots `leave`, `intro`, `start-eating` and `end-eating`, point to the corresponding transitions that represent activities producing state changes. To illustrate the internal structure of a typical transition, let us focus on a transition prototype from the `philosophy-world`, which is shown in figure 3 and called

start-eating. The value in its pre-net-relations slot is:
(Thinking (unit <philo>)). The first element identifies the state slot Thinking in the dynamic object. The second one represents the arc expression (unit <philo>) which is labeling the arc.

KRON variables are identified by angle brackets (e.g., <philo>). As it is general in rule based systems, variables play a double role:

Specify flow conditions. Arc expressions in the preconditions are interpreted as patterns that must be matched. They identify a token that must be in a place slot for a transition to be enabled. For example, the expression (unit <right-chair> left-fork <fork>) (label pre-right-intro in figure 4) defines a pattern that matches all chairs in place Busy-chairs having some value in the slot left-fork. There will be a binding between the variable <right-chair> and the matched instance, and there will be another binding between <fork> and the values in its slot left-fork. Additionally, these bindings can establish equality constraints on other arc expressions of the same **transition** with the same variable names.

Specify data flow. Values bound to variables in preconditions can be transferred to postconditions. Additionally, arc expressions in postconditions can specify modifications in the transferred data. Information of bound variables is also used to update slot values of the tokens involved in a firing.

Some particular features may be used in arc expressions to increase its expressiveness:

- An arc expression may appear as the second component of another arc expression. This is a pattern to match with the objects that are stored in the slot. For example, (see label fork2 in figure 5):
  (unit <right-fork> left-chair (unit <chair> philo <philo>))
  In this case <philo> is bound to the philosopher that is in the philo slot of the chair stored in the left-chair slot of the fork instance bound to <right-fork>.
- A function call may appear as the second component of an arc expression. A function call is represented by a list whose first element is the symbol $, the second element is the function name, and the rest are the arguments. Functions may be used in postcondition for dynamic instantiation purposes. For example, (unit ($ make-philosopher)) may down a new instance of philosopher.
- To facilitate an incremental model design, KRON allows the use of incomplete transitions whose missing variables in preconditions must be provided by transition synchronization (see section 6). These variables play the role of parameters of the activities provided by the objects. For example, all activities of Chair-Eating-Politeness constitute its interface, and they have the parameter <philo>.
- The keyword NULL may appear in postconditions. NULL deletes all values from the slot. For example, (unit <left-chair> right-fork NULL <fk>),

removes all values from slot `right-fork` before adding the new value bound to `<fk>` (label `post-left-intro` in figure 4). Additionally, a slot value can be replaced by another value using a list with `NULL` and the removed values. For example, `(unit <left-chair> right-fork (NULL <fk>) <fj>)` removes the value bound to `<fk>` from slot `right-fork` of object `<left-chair>`, then it adds the value bound to `<fj>` to this slot (label `post-left` in figure 4).

Additionally, each transition has a `predicate` associated. The predicate imposes a logical constraint on the transition enabling. It is a Boolean function, which can only contain those variables that are already in the expressions of the arcs connected to the transitions. The predicate is supposed to be `true` by default.

Sometimes it is useful to execute some action (execution of some particular subprogram). This is the purpose of a transition method called `action`. This method is called each time the transition is fired. The method receives the bindings of the transition variables as a parameter.

5 Instances, classes and inheritance in dynamic objects

The purpose of previous sections was to explain how the dynamic behavior of different kinds of entities is described in KRON. In this section we will focus on the use of inheritance as a mechanism to share code and representation. Therefore, we have considered the inheritance as a subclassing relation. Subclassing highlights redundancy within a system and the object-oriented decomposition yields smaller models through the reuse of common mechanisms, thus providing an important economy of expression. The subtyping relationship has not been considered. (Different approaches to formalize the behavior preservation between parent and descendant classes can be found in [BG91] and [BdC93]).

Object oriented modeling starts by creating a hierarchy of classes, from more generic to more specialized, whose elements will be further instantiated to build a particular system model. Frame based languages make emphasis on inheritance issues and they provide not only support for traditional slots and method inheritance, but also allow the programmers the specification of additional types of inheritance (overriding, adding, unioning, wrappering, ...).

In our working context of discrete event system domain, entities with similar state space and behavior are grouped defining a hierarchy of dynamic object classes. A dynamic object class is a template to construct a composed object, whose instantiation implies the instantiation of the CPN structure that describes its behavior. All instances of a dynamic object class inherit the same Petri net with the same initial marking. Following the same process, transitions with similar structure and behavior are classified in a hierarchy tree of transition classes. Therefore, the behavior of a child class is obtained from the inherited Petri net by adding new transitions and state slots, or providing more specific details about them. For example, inherited state slots may be specialized with additional restrictions on the tokens they can hold.

The creation of the transition hierarchy requires more attention. Thus, a child transition class may be specialized in the following different ways:

Adding enabling conditions: The inheritance type of the `pre-net-relations` and `predicate` slots is *union*. This means that their values are derived by the *and* composition of the values that are in the subclass slot and the inherited values from its superclasses. Therefore, the net enabling conditions of a transition class is restricted by defining new values in the pre-net-relation slot of a transition subclass. Additional enabling conditions may be imposed on a transition class by adding new values to the predicate slot.

Adding new actions: The inheritance type of the `post-net-relations` and `action` slots is also *union*. When the transition is fired `pre-net-relations` and `post-net-relations` imply the modification of the respective state slot values. Therefore, new actions may be defined by adding new `pre` and `post-net-relations` values to a transition subclass. A transition firing also implies the execution of the action method. The `action` method may be specialized in a child transition class by wrapping code before, after or around the inherited code, or overriding it. Moreover, the code of action methods implies the execution of dynamic object methods. Therefore, the action method can be indirectly specialized by the specialization of dynamic object methods.

A transition instance is never created directly, but only through the instantiation of its dynamic object. Transition classes in activity slots are instantiated and replaced by their instances. An important feature of KRON is that the representation of an activity that is carried out in cooperation among different entities, is collected into only one `transition` instance. In this case, the `state` slots of pre- and post-conditions may belong to different dynamic objects. For this reason a transition instance inherits all slot values from the transition class, but `pre/post-net-relations` add to each inherited net relation a reference to the dynamic object instance.

A new dynamic object class can also be created by *multiple inheritance*. In this case, the subclass inherits several separated nets from their superclasses, which can be joined to build a more complex one. The connection can be made by adding transitions and places that model the control flow interaction between inherited nets. Multiple inheritance facilitates composition of incomplete representation behavior (virtual classes) during the model development. This means that the Petri net underlying a dynamic object class may be incomplete, and therefore this class should be refined to complete the behavior representation.

Finally, it is important to note that some problems have been detected with the integration of concurrency and inheritance. In concurrent object oriented languages, it is called *synchronization code* the code that selects the set of services that a concurrent object can execute and that depends on its state; that is, the enabling conditions of transition objects in KRON terms. The reuse of the synchronization code in concurrent object oriented languages has been considered difficult due to *inheritance anomaly*: synchronization code cannot be effectively inherited without non trivial class redefinitions [MWY93]. S. Matsuoka and A. Yonezawa identify three kinds of inheritance anomaly: 1) *State partitioning anomaly* occurs when the subclass needs to make a partition of the set of states the superclass can have. 2) *History only sensitiveness of states*, appears when the methods in a parent class must be modified because the application of a method in a subclass depends on the history information, which does not manifest itself in the values of the inherited instance variables. 3) *State modification anomaly*, appears when the definition of a subclass requires the modification of inherited enabling conditions to account for a new action.

KRON mitigates some of the effects of the inheritance anomaly. A KRON model allows the appropriate separation of the synchronization code (enabling conditions) from the action (a piece of code) attached to transition objects. It makes the refinement of actions easier, allowing the inheritance mechanism to override the two parts separately. On the other hand, Petri nets have a guard based synchronization schema. Thus, the state partitioning anomaly pointed in [MWY93] does not occur because the addition of new net conditions allows the differentiation of substates.

Following with the case study, the figure 5 shows how to specialize the dynamic-object Chair-Politeness. To complete the model, it must be created an instance of Philosophy-world and an instance of Chair-Eating-Politeness. Instances of Chair, Philosopher and Fork, and the initial relations between them, will be created by the constructors of dynamic-objects that execute its initial-marking method.

6 Dynamic entity connections

The construction of system models in KRON is done incrementally by first, designing isolated entities and then imposing the interactions between them to compose a bigger subpart of the system model. From a dynamic perspective, the behavior of an entity is represented by a CPN underlying a dynamic object. The overall dynamic model will be homogeneous if the dynamic interactions between dynamic objects are described in the same terms as the internal behavior of objects. This means that if the interactions are described in terms of places, transitions and arcs, the behavioral model of the system will be a CPN and the advantages of using this formal view can be fully assumed. In KRON, the representation of interactions between dynamic objects becomes the same problem as the high level Petri net connections.

CPNs may be connected by merging transitions or places, and by means of new arcs [Bau88]. In KRON, transition merging has been selected as the main mechanism to represent the interactions between dynamic objects. The advantages of this approach will be pointed out at the end of this section. This approach provides a synchronous communication style that has been adopted by other works as modular-CPNs [CP92], CO-OPN [BG91] or OBJSA [BdC93]. In this mechanism, an interaction between two or more objects can be interpreted as the execution of a joint activity, where each object has only a partial view of

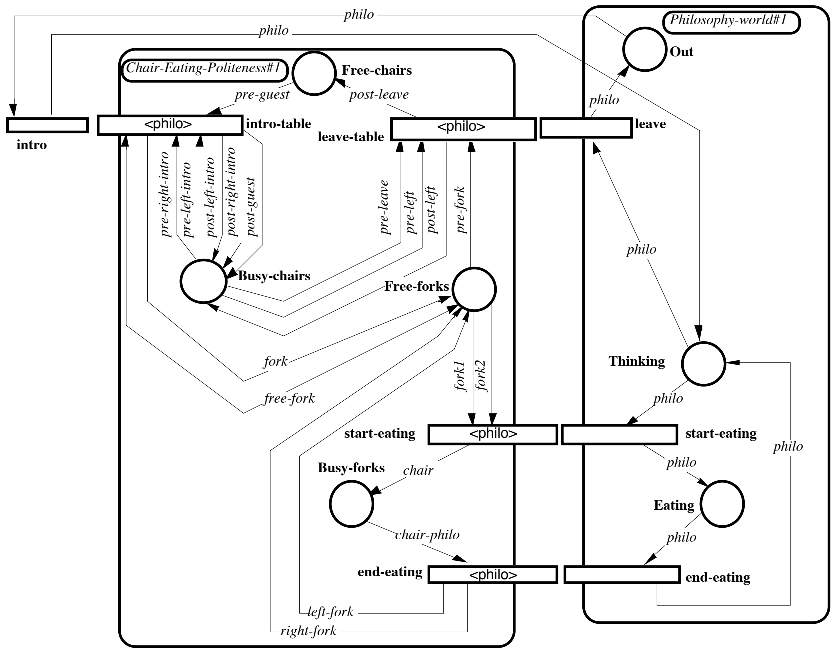


Fig. 6. Synchronization of dynamic-object instances.

the real activity and its constraints. This interaction implies the synchronization of the internal behavior of those objects.

To illustrate the possibilities for dynamic entity connections we will synchronize the instances `philosophy-world#1` and `Chair-Eating-Politeness#1`. (See figure 6). When a philosopher enters or leaves from the philosopher world, new relations between chairs, forks, and philosophers are defined. The action `intro-table` of `Chair-Eating-Politeness#1` implies that a philosopher must occupy a chair, and the action `intro` of `Philosophy-world` must respect the politeness rules. Therefore these actions must be synchronized. In fact, they could be considered as partial views of the same activity. The result is a merging transition that synchronizes the behavior of both dynamic entities. In the same way, the activity `leave` of the `Philosphy-world` must be synchronized with the activity `leave-table` of `Chair-Eating-Politeness`. On the other hand, it is necessary to have two forks to start eating. Therefore, `Philosophy-world` and `Chair-Eating-Politenes` must synchronize its respective activities `start-eating` and `end-eating`.

To support this approach, the dynamic objects interface in KRON is a set of activity slots. Thus, the transitions of a dynamic object can be internal or interface transitions. Only the interface transitions can be externally synchronized. Connections between dynamic objects are established by naming the activity slots that must be related in some manner. The synchronization mechanism generates a new merged transition by multiple inheritance of the originals (for a complete transition merging, additional mechanisms are supported in KRON to specify the relations between variables from different transitions whose names have local scope). The transition generated by the merging replaces the originals in all activity slots involved in the synchronization. It allows the different dynamic objects related by a synchronization to maintain the same view over a transition after the merging.

Two types of synchronization (by transition merging) have been designed for KRON. *Normal synchronization* substitutes synchronized transitions by a unique transition with all pre and postconditions of the original ones. However, a slightly different case arises when a single activity may be synchronized with several alternative activities, and these other activities can not be synchronized with one another. This is the place for a *bilateral synchronization*, which supposes a replication process.

Some characteristics of the proposed approach for dynamic entity synchronization are:

1. Synchronized transitions are handled as a single object that belongs to cooperative objects. In this way, communication between different entities, from a dynamic perspective, is supported by the same formalism that defines the internal dynamic of each object.
2. Synchronized transitions provide a symmetric form of cooperation by an arbitrary number of entities, and no direction of communication is intended. Transition synchronization provides a higher level mechanism to communicate objects than the classical message passing. Collective behavior of objects can be described without an implementation model of communication, and does not restrict the model to the client-server framework. Intuitively, transition objects are similar to the space tuples of generative communication [CG89]. Tokens may interact through transition objects by inserting tuples with the bindings produced by arc expressions. Communication may produce if there is pattern matching between tuples. (In the examples philosopher, chair, and fork entities interact trough transition objects in this way). A comparison of Petri nets and the generative communication can be also found in [HV96], where Petri nets are used to specify the behavior of the hurried philosophers based on objects and the generative communication. In this approach the space tuple is represented by places that hold tokens, whereas in our approach the space tuple is represented by transitions that holds the bindings produced by tokens.
3. It may be argued that synchronized transitions violate the encapsulation, because they have access to the local state of cooperative dynamic objects. However, a synchronized transition denotes a relation between dynamic objects that defines the rules of this violation. As Rumbaugh points out in [Rum87], a relation is not something to be hidden, but rather, to be specified abstractly, without imposing an implementation.

KRON also allows the definition of composite dynamic-objects. When a composite is instantiated, the different parts of the composite are instantiated and the synchronization relations between the parts are established. In this way, KRON may reuse through inheritance and aggregation.

7 Conclusions

In this paper we have presented KRON (Knowledge Representation Oriented Nets), a knowledge representation schema for discrete event systems (DESs). KRON is based on the integration of CPNs with frame based representation techniques and follows the object oriented paradigm. In addition to the features generally supported by object oriented languages, a set of primitives implementing the CPN formalism is included. CPNs provide the mechanism to describe the internal behavior of dynamic entities and the interactions between them. The Hurried Philosophers example has been adapted to highlight some relevant KRON capabilities.

Most of the approaches integrating objects and HLPNs, extend the HLPN formalism. The approach adopted here does not extend the CPN formalism. The frame-based representation of KRON supports the data and methodological aspects with no need to extend the CPN formalism. So, all the advantages of the use of this formalism can be profited from working with KRON.

KRON may reuse models through inheritance and aggregation. On the one hand, KRON uses inheritance as a mechanism to share code and representation. On the other hand, aggregation is supported by CPN composition. CPNs may be connected by merging transitions or places, and by new arcs. In KRON, transition merging has been selected as the main mechanism to represent the interactions between dynamic objects. This approach provides a synchronous communication style with all its advantages.

The semantics of the behavioral rules is supported in KRON by a so called control mechanism or interpreter. The control mechanism interprets the model to make the net evolve. The implementation of an efficient interpreter of KRON models may be found in [BMMV93]. In order to interpret the model, transitions are grouped into conflicts. The interpretation of the model is orthogonal to the model itself. An only interpreter may execute the model, or it may be attached an interpreter to each dynamic entity.

A prototype of a simulation tool with graphical display and animation facilities has been implemented on top of a known knowledge engineering environment called KEE [Int89] from Intellicorp.

References

[Bau88] B. Baumgarten. *Advances in Petri Nets 1988*, chapter On internal and external characterization of PT-net building block behavior. Number 340 in Lecture Notes in Computer Science. Springer Verlag, 1988.

[BB91] M. Baldassari and G. Bruno. PROTOB: An object oriented methodology for developing discrete event dynamic systems. *Computer Languages*, 16(1):39–63, 1991.

[BdC93] E. Battiston and F. de Cindio. Class orientation and inheritance in modular algebraic nets. In *Proc. of IEEE International Conference on Systems, man and Cybernetics, Le Touquet-France*, pages 717–723, 1993.

[BdCM88] E. Battiston, F. de Cindio, and G. Mauri. *Advances in Petri Nets 1988*, chapter OBJSA Nets: a class of high-level Petri nets having objects as domains, pages 20–43. Number 340 in Lecture Notes in Computer Science. Springer Verlag, 1988.

[BDLGV95] G. Berio, A. Di Leva, P. Giolitto, and F. Vernadat. The m*-object methodology for information system design in cim environments. *IEEE Tran. on Systems, Man, and Cybernetics*, 25(1):68–85, January 1995.

[BE86] G. Bruno and A. Elia. Operational specification of process control systems: Execution of prot nets using OPS5. In *Proc. of IFIC'86, Dublin*, 1986.

[BG91] D. Buchs and N. Guelfi. CO-OPN: a concurrent object oriented petri net approach. In *Proc. of the 12th International Conference on Application and Theory of Petri Nets*, pages 432–454, Gjern (Denmark), June 1991.

[Bil89] J. Billington. Many-sorted high-level nets. In *Proc. of Third International Workshop on Petri Nets and Performance Models, Kyoto*, pages 166–179, 1989.

[BM86] G. Bruno and G. Marchetto. Process-translatable Petri nets for the rapid prototyping of process control systems. *IEEE transaction on Sosftware Engineering*, 12(2):346–357, 1986.

[BMMV93] J.A. Bañares, P.R. Muro-Medrano, and J.L. Villarroel. *Application and Theory of Petri Nets 1993*, chapter Taking Advantages of Temporal Redundancy in High Level Petri Nets Implementations, pages 32–48. Number 691 in Lecture Notes in Computer Science. Springer Verlag, 1993.

[CG89] N. Carriero and D. Gerlenter. Linda in context. *Communications of the ACM*, 32(4), April 1989.

[CH94] S. Christense and N.D. Hansen. *Application and Theory of Petri Nets 1994*, chapter Coloured Petri Nets Extended with Channels for Synchronous Communication, pages 159–178. Number 815 in Lecture Notes in Computer Science. Springer Verlag, 1994.

[CKR93] L. Cherkasova, V. Kotov, and T. Rokicki. *Applications and Theory of Petri Nets 1993*, chapter Modeling of Industrial Size Concurrent Systems, pages 552–561. Number 691 in Lecture Notes in Computer Science. Springer Verlag, 1993.

[CP92] S. Christensen and L. Petrucci. *Applications and Theory of Petri Nets 1992*, chapter Towards a Modular Analysis of Coloured Petri Nets, pages 113–133. Number 616 in Lecture Notes in Computer Science. Springer Verlag, 1992.

[DG87] A. Dileva and P. Giolito. High-level petri nets for production system modelling. In *Proc. of the 8th European Workshop on Application and Theory of Petri Nets*, pages 381–396, Zaragoza (Spain), June 1987.

[DGV91] A. Dileva, P. Giolito, and F. Vernadat. Executable models for the representation of production systems. In *Proc. of the IMACS-IFAC Symposium on Modelling and Control of Technological Systems, IMACS MCTS 91*, pages 561–566, Lille (France), June 1991.

[Eng93] S. English. *Coloured Petri Nets for object-oriented modelling.* PhD thesis, University of Brighton, 1993.

[Feh91] R. Fehling. A concept for hierarchical petri nets with buiding blocks. In *Proc. of the 12th International Conference on Application and Theory of Petri Nets*, pages 370–389, Aarhus, 1991.

[FK85] R. Fikes and T. Kehler. The role of frame-based representation in reasoning. *Communications of the ACM*, 28(9):904–920, September 1985.

[HJS89] P. Huber, K. Jensen, and M. Shapiro. Hierarchies in coloured petri nets. In *Proc. of the 10th European Workshop on Application and Theory of Petri Nets*, pages 192–209, Bonn, June 1989.

[HV96] T. Holvoet and P. Verbaeten. Using petri nets for specifying active objects and generative communication. In *Object-Oriented Programming and Models of Concurrency. A workshop within the 17th International Conference on Application and Theory of Petri Nets*, 1996.

[Int89] Intellicorp. *KEE User Guide.* Intellicorp, 1989.

[Jen92] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use.* EATCS Monographs on theoretical Computer Science, Springer-Verlag. Edited by W. Brauer, G. Rozenberg and A. Salomaa, Berlin Heidelberg, 1992.

[Lak93] C.A. Lakos. The role of substitution places in hierarchical coloured Petri nets, thecnical report tr93-7. Technical report, Computer Science Department, University of Tasmania, August 1993.

[LK94] C.A. Lakos and C.D. Keen. LOOPN++: A new language for object-oriented petri nets, thecnical report tr94-4. Technical report, Computer Science Department, University of Tasmania, 1994.

[Mey88] B. Meyer. *Object-Oriented Software Construction.* Computer Science. Prentice Hall, Englewood Cliffs, N.J., 1988.

[MWY93] S. Matsuoka, K. Wakita, and A. Yonezawa. *Research Directions in Object-Based Concurrency*, chapter Inheritance anomaly in object-oriented concurrent programming languages. MIT Press, 1993.

[PR93] M. Paludetto and S. Raymond. A methodology based on objects and petri nets for development of real-time software. In *Proc. of IEEE International Conference on Systems, man and Cybernetics, Le Touquet-France*, pages 717–723, 1993.

[Rei91] W. Reisig. *Theoretical Computer Science 80*, chapter Petri Nets and Algebraic Specifications, pages 1–34. Elsevier Science Publishers B.V., 1991.

[Rum87] J. Rumbaugh. Relations as semantic contructs in an object-orientated language. In *Proc. of the ACM Object-Oriented Programming Systems, Languages and Applications, OOPSLA'87*, pages 466–481, October 1987.

[SB85] C. Sibertin-Blanc. High-level petri nets with data structures. In *Proc. of Workshop on Applications and Theory of Petri Nets. Finland*, June 1985.

[SB94] C. Sibertin-Blanc. *Advances in Petri Nets 1994*, chapter Cooperative Nets, pages 377–396. Number 815 in Lecture Notes in Computer Science. Springer Verlag, 1994.

[Vau87] J. Vautherin. *Advances in Petri Nets 1987*, chapter Parallel Systems Specifications with Coloured Petri Nets and Algebraic Specifications., pages 293–308. Number 266 in Lecture Notes in Computer Science. Springer Verlag, 1987.

[VB90] R. Valette and B. Bako. Software implementation of petri nets and compilation of rule-based systems. In *11th International Conference on Application and Theory of Petri Nets*, Paris, 1990.