Knowledge Representation-Oriented Nets for Discrete Event System Applications

Pedro R. Muro-Medrano, José A. Bañares, and José Luis Villarroel

Abstract— **This paper presents knowledge representation-oriented nets (KRON), a knowledge representation schema for discrete event systems (DES's). 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 high-level 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 of DES's. The use of the DES behavioral knowledge is governed by a control mechanism stored in a separate inference engine. KRON provides an efficient execution mechanism to make the models evolve. This is an adaptation of the RETE matching algorithm in order to deal with the features provided by high-level Petri nets and it takes advantage of its integration with a frame/object-oriented representation schema. Moreover, KRON facilitates dealing with decision points in the execution of nondeterministic models. A prototype of a simulation tool with graphical display and animation facilities has been implemented for KRON and it has been used in several case studies in the manufacturing systems domain.**

I. Introduction

THIS paper deals with a knowledge representation schema for *discrete event system* (DES) models. DES's are dynamic systems that change their state at the occurrence of discrete events. In general, the state of *DES models* has logical or symbolic, rather than numerical values, and the events may also be described in nonnumerical terms [1]. Complex DES's are composed of elements that evolve concurrently and asynchronously. The elements of a DES interact with one other by means of synchronization and information passing mechanisms.

There are several formalisms used to build models of complex DES including those based on *Petri nets*, *calculus of communicating systems*, and *communicating sequential processes*. Petri nets have been recognized as a suitable means to describe such complex DES. They allow formal analysis, graphic representation, and the execution of the system models [2]. Additionally, it is possible to fill the gap between the system modeling and implementation phases by means of automatic code generation techniques [3]. However, the use of ordinary Petri nets in the modeling of large complex DES's can lead to models of unmanageable size. This drawback is reduced by using high-level Petri nets (HLPN's) [4] (e.g., colored Petri nets or predicate/transition nets) which provide more compact and manageable descriptions.

Manuscript received January 19, 1996; revised November 10, 1996 and May 25, 1997. This work was supported in part by grants from the Spanish Interministerial Comission of Science and Technology (CICYT) under Project TAP95-0574 and the University of Zaragoza under Project 284-63.

The authors are with the Departamento de Informática e Ingeniería de Sistemas, Universidad de Zaragoza, 50015 Zaragoza, Spain.

Publisher Item Identifier S 1083-4427(98)01581-1.

Several researchers have established that the high-level Petri net formalism is still lacking in two main aspects.

1) *Methodological aspects*: Several methodologies can be adopted during the system model design process using high-level Petri nets. However, this formalism lacks appropriate features to explicitly support concepts from modern systems engineering such as modularity, encapsulation, top-down and bottom-up designs, data abstraction, specialization, and inheritance.
2) *Data representation aspects*: In the majority of practical applications, it is necessary to describe aspects that are not related to the system dynamics. Moreover, HLPN's are not powerful enough to describe complex systems (such as information or manufacturing systems), which are characterized by having many different attributes and services and complex relations between elements. To conclude, Petri nets are good process-oriented formalisms, but they are not data-oriented.

To deal with these deficiencies, the main approach adopted has been the integration of the high-level Petri net formalism with other data representation paradigms such as abstract data types, object-oriented concepts, or entity relationship models. There are many integrated models in technical literature, which are analyzed in Section II of this paper. It can be established that there has been a global evolution in integration strategies. This evolution leads to a fully integration of Petri nets and objects. The proposed integration model follows this last approach.

This paper is devoted to illustrate the main features involved in knowledge representation-oriented nets (KRON). Two main objectives were considered in its definition: 1) to obtain an overall and powerful representation model for data and control and 2) to incorporate appropriate features in order to facilitate methodological aspects.

KRON is based on the integration of high-level Petri nets with the frame/object-oriented paradigm. KRON increases the features generally supported by object and frame-oriented languages in the following ways.

- It provides a set of semantic constructs implementing the high-level Petri net formalism. HLPN's provide the mechanism to describe the internal behavior of the dynamic entities and the interaction between them.
- It allows the description of the collective behavior of objects with no necessity for a low-level communication model.

- KRON effectively represents this behavior in a declarative rather than procedural form.
- It provides an interpreter of the underlying HLPN, which is based on an efficient matching tool that makes effective use of the underlying data structures.

The paper is organized as follows. First, a brief survey of similar approaches is presented in Section II. Section III presents the KRON constructs that support the HLPN formalism. Section IV introduces the mechanism to support the connection of dynamic entities. The execution model, which is supported by the HLPN underlying the model, is presented in Section V (KRON could be seen as a language that allows the implementation of the HLPN formalism). Section VI illustrates some modeling restrictions to be considered in order to make a correct implementation. Section VII shows the KRON development environment used to create KRON models facilitated by graphical and animation capabilities. Finally, conclusions and future works are presented.

II. Petri Net Integrated Models

A Petri net is a graph with two kinds of nodes, *places* and *transitions*, which represent respectively conditions and actions. Tokens evolving through the places complete the state representation. On the other hand, the execution of a particular activity generally requires the satisfaction of some preconditions in the system state; whereas the activity execution implies additional postconditions in the system state. In a Petri net, preconditions are specified by those arcs going from places to transitions whereas postconditions are specified by arcs going from transitions to places. Transitions whose preconditions are satisfied are enabled and may be fired. A transition firing implies removing the enabling tokens from their previous places, and putting tokens in posterior places.

A high-level Petri net is a Petri net whose tokens carry information that may be represented by data structures. In a HLPN, arcs representing preconditions are labeled by expressions which identify states defined by the value of tokens; arcs representing postconditions are labeled by expressions which define state changes by means of the modification of the value of tokens. In this way, HLPN's provide a more concise behavior representation than ordinary Petri nets.

Many 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;
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 international conferences; this is proof of the growing interest in this topic.

One HLPN extension belonging to the first group is called hierarchical colored Petri net (HCPN). HCPN's [5] provide a set of constructs to support modularity aspects. The idea behind HCPN's is to allow the construction of a large model by combining a number of small HLPN's into a larger net, and different structuring tools are proposed with this purpose. Posterior proposals extending HLPN's with structuring constructs can be found in [6] and [7].

The presentation of the most representative works on integration with algebraic specifications (second group) can be briefly summarized as follows.

- Algebraic nets [8], many-sorted high-level nets [9], and Petri nets with structured tokens [10] are a result of the integration of HLPN's (used to describe the control structure of the system) and algebraic specifications (used to describe the data structure). The main objective is to expand the net formalism with an explicit abstraction mechanism and a description formalism used for the data structures. Algebraic specifications provide a suitable formalism for data representation which is independent of a concrete programming language. However, there is no much emphasis on the methodological aspects.
- OBJSA nets [11] are also based on algebraic specifications and introduce modularity in Petri nets. The objective is to allow data abstraction and introduce net modularity. The algebraic Petri net is constructed over the OBJ2 abstract data type language. The formalism is increased by means of a building methodology based on transition synchronization. However, the modeling power is affected by the restriction of net model on *superposed automata nets*. This model has been revised in [12] in order to introduce object orientation and inheritance. In the same way, PrE-nets with algebraic specifications [13] (SEmiGrAphical Specification language, or SEGRAS), which were defined before algebraic nets, were formalized introducing modularity into Petri nets. However, the modeling power is also affected by the restriction on PrE-nets [14], which only allows single individual objects.
- These previous works have been the basis of many others, most of them considering object-oriented aspects. Concurrent object-oriented Petri nets (CO-OPN) defined in [15] are an example of that work. The CO-OPN formalism extends algebraic nets with modular features, which are carried out by introducing parameterized transitions (in order to cope with data transmission) and synchronization mechanisms. A system is composed of objects, whose behavior is defined by an algebraic net, that communicate by transition synchronization. In CO-OPN, a method is an externally visible transition in opposition to the internal transition describing internal behavior.

Previous approaches are based on algebraic specifications. We consider that approaches based on a frame/object approach can be closer to human conceptual thinking than the ones based on algebraic specifications. What is required is a conceptual model which will enable engineers and computer scientists to describe domain concepts in a more intuitive way, and which may also be understandable by users.

- In [16], the authors emphasize that the current Petri net formalism does not support a real data-oriented view of the system. Based on that, a new augmented model is proposed: object-oriented net (OONET) which combines an object-oriented data model, called L2, with high-level

Petri nets. The object-oriented paradigm is considered in data descriptions, but not in modeling the control structure. Thus, rule enabling depends on the presence of tokens (and their time-stamps), but not on the value of the tokens. Objects are considered to be passive without message passing. With the same goal of increasing the data modeling power, in [17] high-level Petri nets are integrated with the entity–relationship model to obtain the EER formalism. This model is revised in [18] 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 [19]. 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.

- Hierarchical object-oriented design (HOOD) nets [20], [21] integrate HOOD with Petri nets. The result is a model which is control-oriented. The object-oriented methodology is not fully extended to data modeling.
- Object Petri nets (PNO), which have been widely referred to in technical literature, were defined in [22] 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. In the first version, few methodological aspects were considered in the overall system modeling. Starting from this seminal work, in [23] HOOD/PNO is proposed as a software engineering methodology that integrates PNO with the HOOD methodology. In [24] 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. Although a primary motivation for Sibertin's introduction PNO was to integrate the information model with Petri nets (where data have an effect on behavior), most of the subsequent work on PNO has focused on the behavioral aspects rather than on structural representation issues.
- Reference [25] presents PROTOB, an object-oriented language and methodology based on PROT nets [26]. In this object-oriented approach, objects communicate by message passing and a hierarchical object decomposition like HOOD is allowed. However neither inheritance nor data representation aspects are considered.
- LOOPN++ [27] has mainly been used to describe network protocols. LOOPN is a textual language that supports object-oriented structuring into HLPN's. The language has a formal semantics which makes it possible to transform object Petri nets (OP-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 many integrated models. It can be established as a conclusion that all these integration works have evolved in the same direction to obtain a full integration of Petri nets and objects. As has been pointed out in [28], there are two approaches to the integration of Petri nets and object-oriented concepts: objects inside Petri nets versus Petri nets inside objects. However, the approach based on the integration of objects inside Petri nets (e.g., object Petri nets) has evolved to incorporate also the second one. 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 HLPN's. However, there is a great scope for further work in tailoring analysis techniques to extended HLPN's. The integration model presented in this paper provides a full integration of HLPN's and the object model, and it does not extend the HLPN formalism.

III. KRON Constructs

Knowledge representation of DES's 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 between objects in order to achieve collective behavior. 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 a high-level Petri net based formalism. Fig. 1 illustrates the frame hierarchy that provides the conceptual entities that support the construction of KRON models. The KRON hierarchy can be decomposed into three important groups.

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 HLPN. The state will be mapped in a combination of HLPN places and structured tokens, whereas the activities that produce state changes will be mapped in HLPN transitions. The constitutive elements of the structure of a HLPN 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 following 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 HLPN. State information in a HLPN is represented by its marking, this means the places and the tokens located in the places. Tokens which evolve by a HLPN are not mapped onto specialized objects in KRON. Any entity evolving through state slots plays the role of a *token*.

- Activities producing state changes in a dynamic object are represented by transitions, and they are equivalent to the transitions of a HLPN. 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 IV). In this way, transitions also provide information about the set of applicable services for the current state.

Finally, HLPN's 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 updatings of direct-inverse relations. From the HLPN point of view, relations make possible the combination of objects in more complex data structures which represent tokens. KRON also provides specific relations related to the description of the following dynamic behavior:

- net relations support HLPN 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 HLPN (*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.

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.

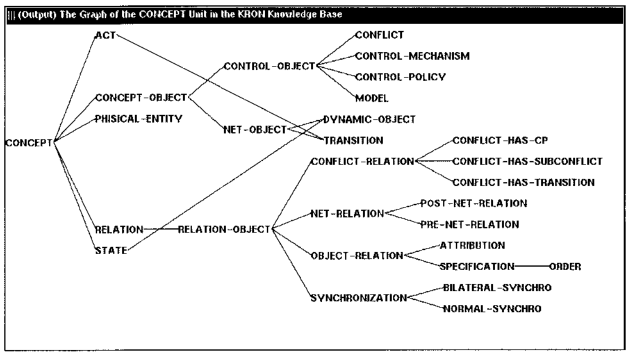


Fig. 1. KRON hierarchy.

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.

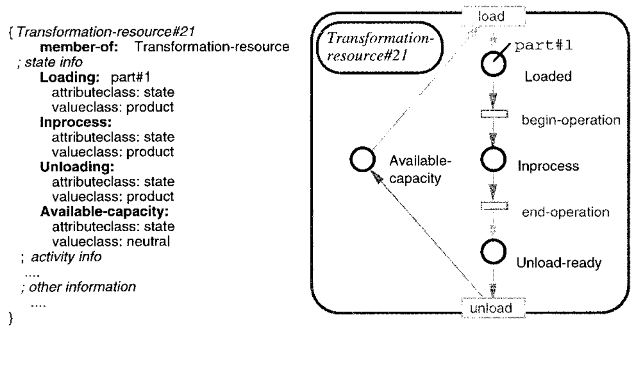
The rest of this section is devoted to provide more details about state and activity representations. Some issues about dynamic object classes, instances, and inheritance will be described in the following sections.

A. State Representation

In order to illustrate the representation schema used to design dynamic objects, let us focus on a representative entity from a manufacturing system application which is called `transformation-resource#21`. This object holds the prototypical knowledge about a generic transformation resource which is instantiated to collect its particular features. It has several state slots representing its overall state (see Fig. 2). `Available-capacity` represents the capacity to process different parts at the same time, whereas `loaded, inprocess,` and `unload-ready` identify different substates when part or its overall capacity is being used in processing products.

Objects that evolve through the underlying net of a dynamic object have a dynamic relation with state slots called marking relation, which defines the entity state (for example, as can be deduced from Fig. 2, product `part#1` has a marking relation with state slot `loaded`).

Each place in a HLPN 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 example, `available-capacity` may hold `neutral` tokens (they can also be seen as simple integer values, because their identity and slots are not considered). However, `Loaded, inprocess,` and `unload-ready` may hold `products`, and the identity of instances in these places is considered in the state representation. A `product` instance in `loading` denotes that it is loaded and ready to be processed, in `inprocess` denotes that



```
{ Transformation-resource#21
      member-of:   Transformation-resource
  ; state info
      Loading:  part#1
         attributeclass: state
         valueclass: product
      Inprocess:
         attributeclass: state
         valueclass: product
      Unloading:
         attributeclass: state
         valueclass: product
      Available-capacity:
         attributeclass: state
         valueclass: neutral
  ; activity info
   ....
  ; other information
   ....
}
```

Fig. 2. `Transformation-Resource`: (a) textual state description and (b) graphical state specification.

the product is being processed, and in `unloading` that the `product` has been processed and ready to be unloaded.

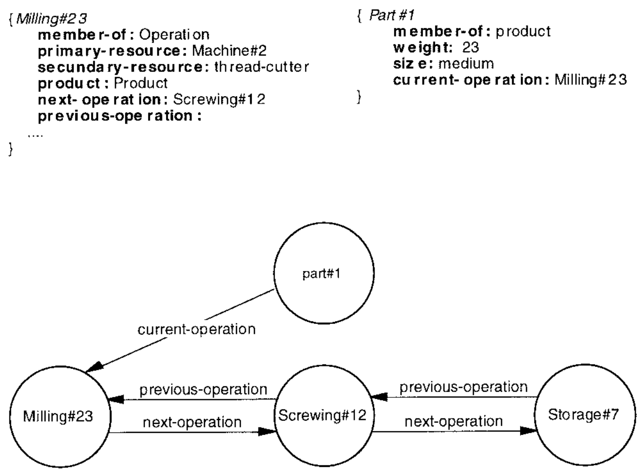
In KRON, token slots are taken into account in order to complete the state representation. This means that the system state is defined not only by the marking relations, but also by the token slots that are relevant for that state. Structured tokens allow KRON to benefit from some HLPN advantages like the aggregation of dynamic information to obtain more concise models, maintaining at the same time the power of using relations and its integration with the rest of the system code.

To illustrate the representation power supported by this relational construction of tokens, let us increase the complexity of the example. The processing of product `part#1` is not only performed in machine `transformation-resource#21` but that its complete processing is performed by means of a sequence of operations in different machines. A good designer would include, for example (see Fig. 3), a new object class `operation` with information about several operation features and related operations (`next-operation` and `previous-operation`). Fig. 3 illustrates graphically how `part#1` (and all products of type `product`) follows a sequence of three operations (`milling#23`, `screwing#12`, and `storage#7`). This relational information is managed by the application from various possible points of view, other than its token perspective.

B. Activity Representation

From the data structure point of view, a transition holds descriptions such as specialization relation, membership of its dynamic object, procedural information attached to the activity, pre and postconditions, actions produced by activity execution, etc.

From a discrete event system perspective, transitions carry out the specification and semantics of HLPN 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.



```
{ Milling#23
      member-of: Operation
      primary-resource: Machine#2
      secondary-resource: thread-cutter
      product: Product
      next-operation: Screwing#12
      previous-operation:
   ....
}
```

```
{ Part#1
      member-of: product
      weight: 23
      size: medium
      current-operation: Milling#23
}
```

Fig. 3. Token and relation representations.

On the other hand, from a knowledge representation perspective, information about activities can be considered as declarative knowledge in the "if/then" rule style (the similarities between HLPN transitions and rules in rule based systems have been pointed out in several works [29]). 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. In this sense, pre-net-relations play the role of the *lhs* as enabling conditions, and both pre-net-relations and post-net-relations play the role of the *rhs* as causal relations. The integration of rule-like knowledge in a frame based representation schema is common to most knowledge engineering environments, see for example KEE [30], LOOPS [31], and KnowledgeCraft [32]. Some advantages of such integration are described in [33] and [34]. The rest of this section is devoted to explain how several transition features are represented and used in KRON.

Expressions labeling the arcs are represented in KRON as arc expressions in pre- and post-conditions. An arc expression is a specification of restrictions on objects. These restrictions are represented by a list of component pairs: the first component is the specification of a slot name or the string `unit.name` denoting an object name; 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 `Transformation-resource#21` object, the activity slots `start-process`, `end-process`, `begin-operation`, and `end-operation`, 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 instance from the `Transformation-resource#21` object, which is shown in Figs. 4 and 5 and called `begin-operation#21`. The value in its `pre-net-relations`

```
{ Transformation-resource#21
     member-of:Transformation-resource
  ;state info
   ....
  ; activity info
     load: load#21
     unload: unload#21
     begin-operation: begin-operation#21
     end-operation: end-operation#21
  ;other information
    ....
}

Where

n:     (unit.name 'n)
prod: (unit.name <prod>)
pop : (unit.name <prod> current-operation
              (next -operation <next-op>))
next: (unit.name <prod>
          current-operation NULL <next-op>)
```

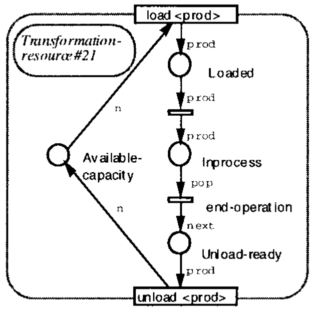


Fig. 4. `Description of Transformation-resource#21`.

```
{ begin-operation#1
     member-of:   begin-operation
     pre-net-relations:
       (Transformation-resource #21 Load
         (unit.name <prod>))
     post-net-relations:
       (Transformation-resource #21 Inprocess
         (unit.name <prod>))
     assoc-data:  <prod>
     predicate:   T
     action:
}

{ end-operation#1
     member-of:    end-operation
     pre-net-relations:
       (Transformation-resource #21 Inprocess
         (unit.name <prod>  current-operation
                       (next -operation <next-op>)))
     post-net-relations:
       (Transformation-resource #21 Unload-ready
        (unit.name <prod>
                  current-operation NULL <next-op>))
     assoc-data:  <prod> <next-op>
     predicate:  T
     action:
}
```

```
{ load#21
     member-of:   load
     pre-net-relations:
       (Transformation-resource #21  Available-capacity
         (unit.name 'n))
     post-net-relations:
       (Transformation-resource #21 Loaded
         (unit.name <prod>))
     assoc-data:  <prod>
     predicate:   T
     action:
     parameters:   <prod>
}

{ unload#21
     member-of:   unload
     pre-net-relations:
       (Transformation-resource #21  Unload-ready
         (unit.name <prod>))
     post-net-relations:
       (Transformation-resource #21 Available-capacity
         (unit.name 'n))
     assoc-data:  <prod>
     predicate:   T
     action:
     parameters:   <prod>
}
```

Fig. 5. Transitions of `Transformation-resource#21`.

slot is: `(Transformation-resource#21 Loaded (unit.name ⟨prod⟩))`.

The first two elements identify the state slot `Loaded` in the dynamic object `Transformation-resource#21`. The third element represents the arc expression `(unit.name ⟨prod⟩)` which is labeling the arc.

KRON variables are identified by angle brackets (e.g., `⟨prod⟩`). 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.name ⟨prod⟩ operation ⟨op⟩)` defines a pattern that matches all tokens having some value in the slot `operation`. There will be a binding between the variable `⟨prod⟩` and the matched value with slot name name, and there will be another binding between `⟨op⟩` and the values in its slot `operation`. 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 which are stored in the slot. For example: `(unit.name ⟨prod⟩ current-operation (next-operation ⟨next-op⟩))`

  In this case `⟨next-op⟩` is bound to the object that is in the `next-operation` slot of the object stored in the `current-operation` slot of the object bound to `⟨prod⟩`.
- 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 IV). These variables play the role of parameters of the services provided by the objects. For example, the activities `load` and `unload` of `Transformation-resource#21` constitute its interface, and they have the parameter `⟨prod⟩`.
- The keyword `NULL` may appear in postconditions. `NULL` deletes all values from the slot. For example, `(unit.name ⟨prod⟩ current-operation NULL ⟨next-op⟩)`, removes all values from slot `current-operation` before adding the new value bound to `⟨next-op⟩`. Additionally, a slot value can be replaced by another value using a list with `NULL` and the removed value. For example, `(unit.name ⟨prod⟩ operation (NULL ⟨op⟩) ⟨next-op⟩)` removes the value bound to `⟨op⟩` from slot `operation` of object `⟨prod⟩`, then it adds the value bound to `⟨next-op⟩` to this slot.

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.

C. Instances, Classes and Inheritance in Dynamic Objects

The purpose of previous subsections was to explain how the dynamic behavior of particular instances is described in KRON. However, as it is typical in object-oriented systems, the prototypical knowledge of the `transformation-resource#21` behavior is inherited from its ancestors; an instance is only the last link on the inheritance tree. In this section we will focus on the use of inheritance as a mechanism to share code and representation.

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, and wrappering).

The construction of the class hierarchy is based on a classification process. 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 HLPN 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/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.

In this design context, the construction of the `Transformation-resource`, whose instance `Transformation-resource#21` has been shown in previous examples, does not start from scratch, but as a specialization of another class: `Manufacturing-resource`. The `Manufacturing-resource` class represents the behavior of a generic manufacturing resource. The responsibility of this abstract entity is to perform some process on some product (modification, transport, storage, etc.). The underlying Petri net of `Manufacturing-resource` specifies that a generic manufacturing resource has a limited capacity that must be available to perform its process and that the process starts with a load action and finishes with an unload action.

The `Transformation-resource` class is built from `Manufacturing-resource`. As it is shown in Figs. 6 and 7, the `Transformation-resource` class adds two actions, `begin-operation` and `end-operation`, which describe respectively the beginning and end of the transformation performed on the product. It also defines substates of products in `Used-capacity`: `Loaded, Inprocess,` and `Unload-ready`.

Figs. 6, 7, and 8 provide a better understanding of the transition inheritance in KRON and its use by dynamic objects. The `load` action of a `Manufacturing-resource` is specialized in `transformation-resource` by overriding the `load` slot value with `load-tr`, which is a child of `load`

```
{Manufacturing-resource
   is-a:   dynamic-object
 ;state info
   Available-capacy:
      attributeclass: state
      valueclass: neutral
 ; activity info
   load: load
   unload:  unload
 ; other information
   ....
}
```

```
{Transformation-resource
   is-a:  Manufacturing resource
 ; state info
   Loadig:
      attributeclass: state
      valueclass: product
   Inprocess:
      attributeclass: state
      valueclass: product
   Unloading:
      attributeclass: state
      valueclass: product
 ; activity info
   load:  load-tr
   unload:  unload-tr
   begin-operation:   begin-operation
   end-operation:   end-operation
 ; other information
   ....
}
```

Fig. 6. Template modification of dynamic object classes.

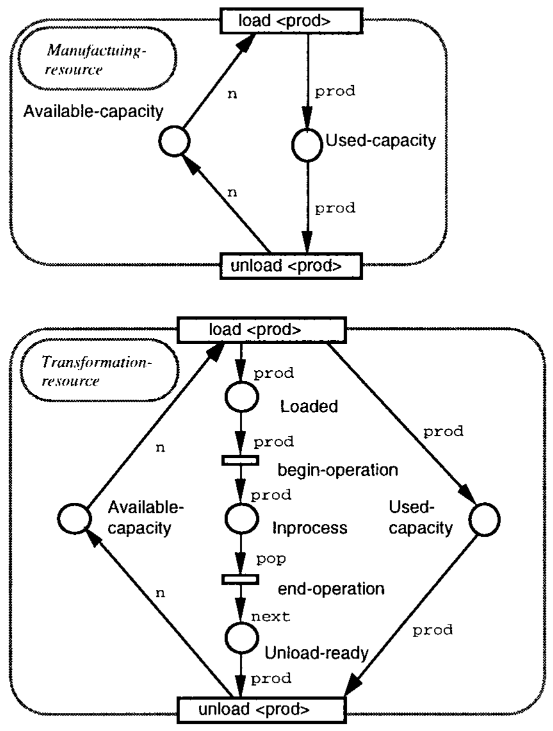


Fig. 7. Behavior modification of dynamic object classes.

```
{load                                                  {unload
    is-a: transition                                       is-a: transition
    pre-net-relations:                                     pre-net-relations:
      (Available-capacity (unit.name 'n))                    (Unload-ready (unit.name <prod>))
    post-net-relations:                                    post-net-relations:
      (Used-capacity (unit.name <prod>))                     (Available-capacity (unit.name 'n))
    assoc-data:  <prod>                                    assoc-data:  <prod>
    parameters:   <prod>                                   parameters:   <prod>
    predicate:  (unit.name  <prod> is-a 'product)       }
 }

{load-tr                                               {unload-tr
    is-a: load                                             is-a: load
    post-net-relations:                                    pre-net-relations:
      (Loaded  (unit.name <prod>))                           (Unload-ready  (unit.name <prod>))
}                                                      }
```

Fig. 8. Template modification of transition classes.

transition. `load-tr` specializes the load action specifying that loaded products will be placed in `loaded` state slot. In the same way, the `unload`transition is specialized by overriding the `unload` slot with its `unload-tr` child.

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 nontrivial class redefinitions [35]. Matsuoka and 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 out in [35] does not occur because the addition of new net conditions allows the differentiation of substates.

IV. 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 HLPN 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 HLPN 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 HLPN connections.

HLPN's may be connected by merging transitions or places, and by means of new arcs [36]. 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 in Petri net integration as CO-OPN [15] or OBJSA [12]. 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 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 some objects designed above with other auxiliary entities creating a more complex model. Consider, as an example, a simple manufacturing work cell, composed of two machines (`M#1` and `M#2`) with input and output buffers, a random-access input store (`IS#1`), a random-access output store (`OS#1`) and a robot (`R#1`) for palleting. Fig. 9(a) shows the dynamic object instances needed before their connections. (Underlying Petri nets have been simplified and the arc expressions and marking relation are not illustrated for simplicity.)

Let us focus on the connections between machine `M#1` and its input buffer `IBM#1`. Transition `unload` from `IBM#1`

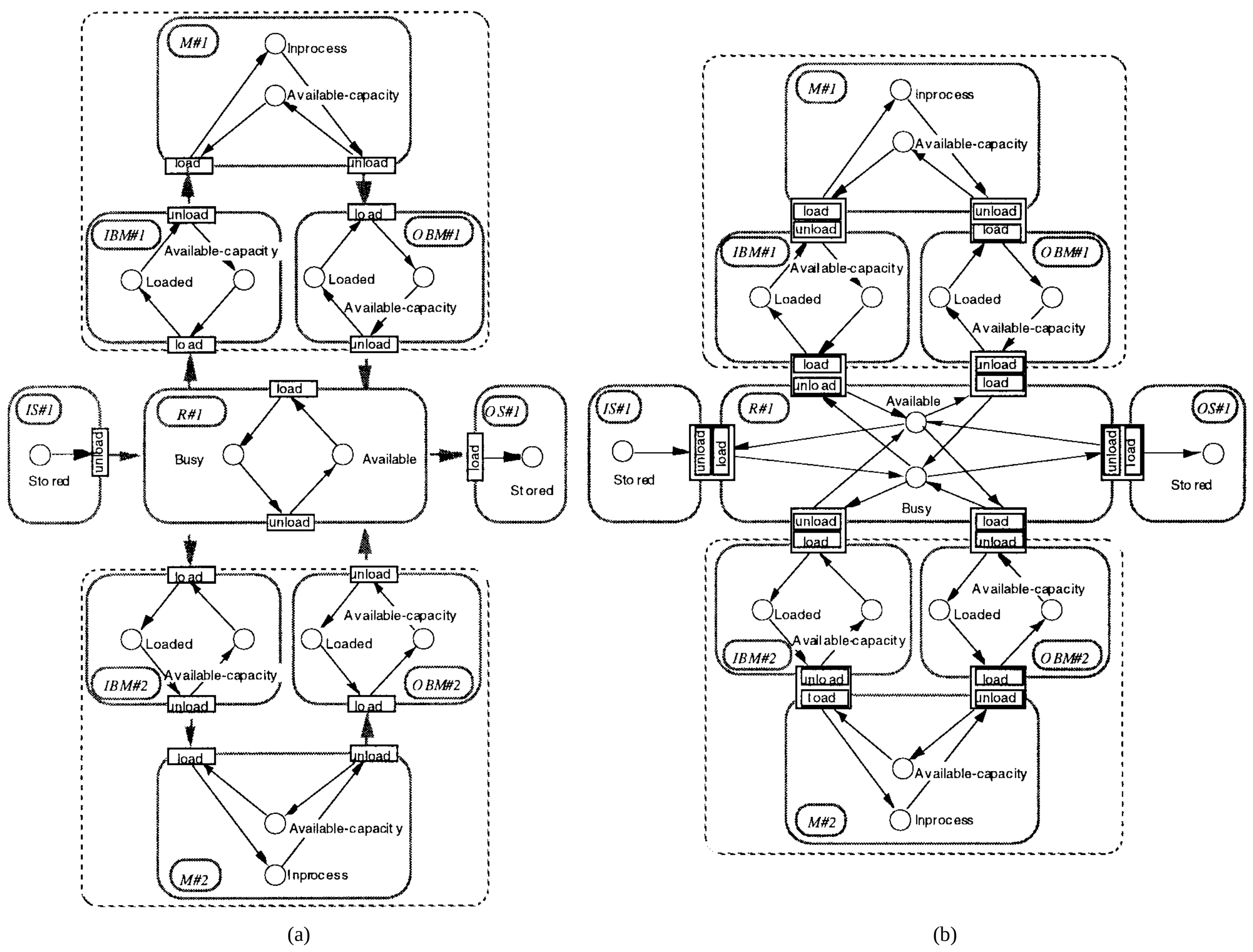


Fig. 9. Example of normal and bilateral synchronization.

specifies an activity to unload a part stored in the buffer; to do that, the part must be in the buffer. On the other hand, transition load from M#1 specifies an activity to load a part in machine M#1; to do that, it is necessary the availability of M#1 (specified by its preconditions). In the example, we want to feed M#1 with the parts from IBM#1, this means that there must be some interaction between both activities (in fact, they could be considered as partial views of the same activity). This interaction can be represented as the merging of transition load from M#1 and transition unload from IBM#1. The result is a transition that models the transfer of a part from the buffer to the machine synchronizing the behavior of both.

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 (this is the case of the interaction between M#1 and IBM#1, which was shown above and which is illustrated graphically in Fig. 9). 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. This is the case of the robot R#1 in the example. R#1 can be used to transport the parts between buffers and stores. The same load transition is used to take a part anywhere, but a different instance must be created for any buffer or store, which must not be influenced by the others. The model of the robot must be built as a reusable component independent of the necessary number of inputs and outputs. Thus, the robot is modeled with only one input and one output that will be replied as necessary when the robot is connected

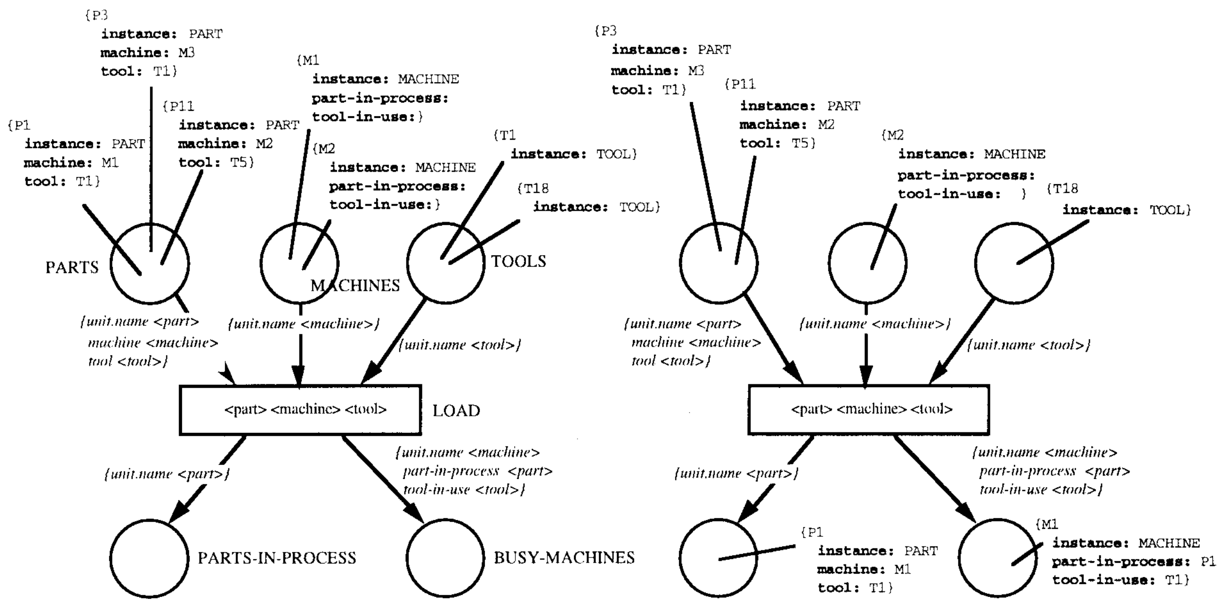

{P3 instance: PART machine: M3 tool: T1}
{P11 instance: PART machine: M2 tool: T5}
{P1 instance: PART machine: M1 tool: T1}
{M1 instance: MACHINE part-in-process: tool-in-use:}
{M2 instance: MACHINE part-in-process: tool-in-use:}
{T1 instance: TOOL}
{T18 instance: TOOL}
PARTS
MACHINES
TOOLS
{unit.name <part> machine <machine> tool <tool>}
{unit.name <machine>}
{unit.name <tool>}
<part> <machine> <tool>
LOAD
{unit.name <part>}
{unit.name <machine> part-in-process <part> tool-in-use <tool>}
PARTS-IN-PROCESS
BUSY-MACHINES
{M1 instance: MACHINE part-in-process: P1 tool-in-use: T1}

Fig. 10. A simple KRON net example.

with other resources of the manufacturing plant. The robot model before synchronization can be seen in Fig. 9(a) whereas Fig. 9(b) shows the robot model after its synchronization.

Some characteristics of the proposed approach for dynamic entity synchronization are as follows.

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) Transition merging has the advantage of easing the system analysis, because many properties are preserved when nets are composed in this way ([37] and [38] among many others).
3) 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.
4) 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 [39], a relation is not something to be hidden, but rather, to be specified abstractly, without imposing an implementation.

V. The Execution Model

In the previous sections, we explained the different primitives and mechanisms to create models of discrete event systems in KRON, but we still have to consider how these models are really executed. The underlying HLPN provides the behavioral rules for the model evolution which is established by transition *firing*. Firstly, the execution model will be illustrated through an example and then, an efficient implementation of the execution mechanism will be presented. This is the software created to simulate the firing of transitions following the theoretical rules imposed by the HLPN semantics.

Fig. 10 shows an example used to illustrate the execution model. The net, composed of five places and only one transition, represents the loading activity for a set of machines. The expression (unit.name ⟨part⟩ machine ⟨machine⟩ tool ⟨tool⟩) in the arc from place PARTS to transition LOAD, allows three different bindings for variables (⟨part⟩ ⟨machine⟩ ⟨tool⟩) regarding the actual marking:
(⟨part⟩ = P1 ⟨machine⟩ = M1 ⟨tool⟩ = T1)
(⟨part⟩ = P3 ⟨machine⟩ = M3 ⟨tool⟩ = T1) and
(⟨part⟩ = P11 ⟨machine⟩ = M2 ⟨tool⟩ = T5).

A transition is enabled if there exists a *consistent binding* for its associated variables. This means that all transition variables are bound, the bindings from all input arc expressions are the same and the variable values verify the restrictions imposed by the predicate associated to the transition. Each of these consistent bindings defines a different *firing mode*. In the example shown in Fig. 10, there exists only one firing mode that is defined by the following consistent binding: (⟨part⟩ = P1 ⟨machine⟩ = M1 ⟨tool⟩ = T1).

A transition firing may occur if the transition is enabled by some firing mode. The state change happens when the firing mode occurs.

It is important to note that HLPN based models can not be completely deterministic. There exist points in which decisions must be taken. From the Petri net point of view, this decision point corresponds to a *conflict*. A conflict can be created by just one or several transitions with firing modes that compete for the same tokens. A conflict must be solved as a whole. That means that if two transitions are in conflict with a third transition they are also in conflict. A conflict groups transitions that are in a coupled conflict relation in the net, and associates a control policy to decide the firing modes that will be fired. In view of the Petri net structure it is possible to make a partition of transitions in conflicts automatically. According to what we have established so far, the execution of a net follows a conflict-oriented approach. Control policies receive the firing modes that enable transitions and decide the firing or firing modes to be executed.

The behavioral rules are supported in KRON by a so-called **control mechanism** or **interpreter**. The control mechanism interprets the model to make the net evolve. The implementation is based on the similarities between the inference engine of a rule based system and the interpretation mechanism of a HLPN. The proposed technique makes use of an adaptation of the RETE matching algorithm [40], which is used in rule based language, such as OPS5 [41], to provide an efficient inference engine. As in RETE, the main idea is to exploit temporal data redundancies (coming from the marking that is not changed during transition firing).

The similarities between Petri nets and rule based systems have been pointed out in several papers [42]–[45]. A HLPN can be interpreted as a rule based system in which transitions have the role of rules and tokens are the working memory elements. We are interested here in those aspects of this equivalence that are important for the interpretation mechanism.

Rule based systems can be decomposed into three parts: rule memory, working memory and inference engine. The rule memory contains declarative information based in precondition/consequence sentences. The working memory (data memory) contains dynamic data that are compared with the precondition part of the rules. The individual elements of the working memory are referred to as the working memory elements. The rule interpretation mechanism is materialized by the inference engine. The inference engine executes what is called a recognize-act cycle [41], which is itself composed of three main activities.

1) *Match*: Performs the comparison of the dynamically changing working memory elements with the precondition part of the rules. If a precondition is satisfied, the rule is included in the conflict set (the set of executable rules for the present working memory state). Currently, the match phase problem is often solved by the RETE match algorithm [40].
2) *Select*: Selects one rule from the conflict set.
3) *Act*: Executes the selected rule according to its consequence part.

The inference engine cycles until no rules are satisfied or the system is explicitly halted.

Differences in the representation appear because a rule does not remove the data that enable it in the execution. It forces the explicit representation of these updatings in the consequent part of rules.

Differences in the inference engine appear because HLPN's provide some features that make its implementation more specific than the one in general rule-based systems. The control mechanism of KRON implements a strategy for specializing the RETE match algorithm so that it is more suitable for HLPN interpretations; furthermore, the KRON execution model allows executing different firing modes in the same cycle to preserve the concurrent semantics of HLPN's.

A. The RETE Algorithm

The RETE match algorithm [40] avoids the brute force approach taking advantage of *temporal redundancy* (persistence of information in the working memory across the recognize-act cycle is called temporal redundancy). This is accomplished by matching only the changed data elements against the rules rather than repeatedly matching the rules against all the data. With this purpose, the information about previous matchings is recorded in a graph structure called *network*.

The network is composed of a global *test tree*, common for all rules, and a *join tree* specific for each rule. The test tree is composed of *one-input* nodes, each of them representing a test over some attribute. A path between the root node and a leaf node represents the sequence of tests required by a rule precondition. The information related to binding consistency tests is represented by the join tree associated to the rule (join tree nodes are called *two-input* nodes or *join* nodes).

When an element is added to the working memory, a pointer to the element is introduced in the root of the test tree and propagated if the test is successful. The working memory element pointers coming out of the test tree leaves are introduced in the join tree. Then, the pointers are combined in tuples and stored in each two-input node. Each tuple collects the working memory elements allowing a consistent binding at the corresponding level. A tuple in a final two-input node points to the working memory elements with which the associated rule can be applied.

On the other hand, when a working memory element is removed, the corresponding pointer must be removed from the nodes of the test tree. The tuples having this pointer must also be removed from the two-input nodes. When many working memory elements match the same condition element, removing a working memory element is expensive. It takes a linear search to find the particular element to be removed in the list of pointers for a condition. The RETE test tree and join tree associated to the transition `LOAD`, seen as a rule, is shown in Fig. 11.

B. Taking Advantage of HLPN Structure

The main differences between HLPN's and rule based models in the match phase are related to the *working memory*. A rule based system has just one global working memory for all rules, whereas a HLPN has its working memory split into places.

The preconditions of a transition must match only against the tokens of its input places. This fact makes a place to be seen as a working memory partition. The main effects on a RETE network produced by this partition can be established as follows.

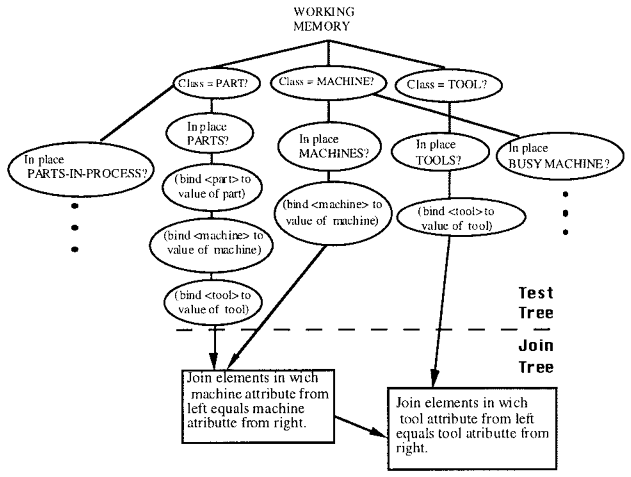

WORKING MEMORY
Class = PART?
Class = MACHINE?
Class = TOOL?
In place PARTS-IN-PROCESS?
In place PARTS?
In place MACHINES?
In place TOOLS?
In place BUSY MACHINE?
(bind <part> to value of part)
(bind <machine> to value of machine)
(bind <tool> to value of tool)
(bind <machine> to value of machine)
(bind <tool> to value of tool)
Test Tree
Join Tree
Join elements in wich machine attribute from left equals machine atributte from right.
Join elements in wich tool attribute from left equals tool atributte from right.

Fig. 11. RETE network generated from transition LOAD.

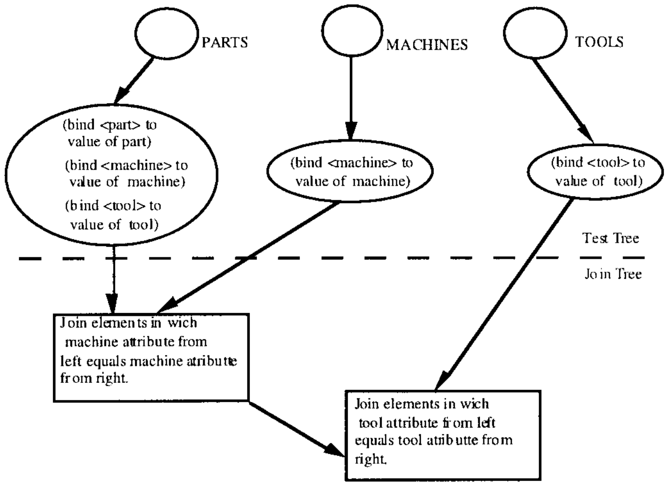

PARTS
MACHINES
TOOLS
(bind <part> to value of part)
(bind <machine> to value of machine)
(bind <tool> to value of tool)
(bind <machine> to value of machine)
(bind <tool> to value of tool)
Test Tree
Join Tree
Join elements in wich machine attribute from left equals machine atributte from right.
Join elements in wich tool attribute from left equals tool atributte from right.

Fig. 12. Network generated from LOAD transition profiting from the HLPN structure.

- The root node is split into several nodes, one for each input place, each one defining itself a local working memory.
- The test tree of the network is reduced because no class or place tests are needed. Consider the RETE network for the transition LOAD. Each branch has a test over the token class and a test over the place where the token is. The place tests can be avoided when it makes use of the HLPN structure because each condition element has an input arc implicitly associated. The class test (PART, MACHINE, or TOOL) can also be avoided because each place has a set/class of possible tokens associated.
- It can not make use of the *structural similarity* [41]. The structural similarity allows the sharing of partial branches of the test tree by different, but similar, condition elements. Except in places that are shared by several transitions, the branches of the test tree come out from separate root nodes. This fact makes it impossible for nodes to be shared.

If the structural similarity is not used, the test tree obtained is a set of separate chains coming out from root nodes. Thus, each chain can be reduced to only one node representing all tests and bindings of the corresponding condition element. These nodes will be called *entry*nodes. This is graphically illustrated by the network shown in Fig. 12. It can also be compared to the RETE network in Fig. 11.

C. KRON Inference Cycle

The recognition phase is executed when a token is added or removed from a place. We will illustrate how the pattern matching phase takes advantage of the partition of working memory. The phases of selection and execution should take into account the concurrency represented by the model. It implies that more than one enabled transition can be fired.

1) Matching Phase: A similar data structure to the RETE network is generated for each KRON transition. As in a RETE network, the intermediate results of the matching process are stored and recalculation can be avoided while no changes in the allocation of the tokens of the places involved are made. However, there exists another important aspect to be considered in the interpretation of HLPN's. Removing a datum from the working memory in the RETE match algorithm implies finding all references to the datum through the RETE network. In order to find the nodes that have references to a removed datum, the original RETE algorithm repeats the process of pattern matching. This source of inefficiency is not dramatic in rule base systems because this is not an action that occurs systematically. But it would be unacceptable in HLPN interpretations. For this reason, in KRON, each set of bindings is linked with the set of bindings that generates it by means of consistency tests (their predecessors), and with their successors. The set of bindings located in entry nodes do not have predecessors. In this case, a link is set between the set of bindings and the token that generated it. This highly related data structure allows a fast tree updating because it avoids search.

2) Selection Phase: The selection of one rule from the conflict set is accomplished in rule-based systems applying concepts such as recency, refraction or specificity (see strategies LEX and MEA from OPS5 language [41]). However, there are other important issues to be considered in implementing HLPN's.

a) A HLPN can model concurrence. In each execution cycle, more than one enabled transition can be fired.
b) The strategies to solve conflicts depend on application, and the resolution objectives can be different in each conflict.

In the KRON interpreter, transitions are grouped by conflicts, each one having its resolution strategy called control policy. During the selection, all enabled transitions from a conflict are considered as a whole, the conflict control policy is responsible for providing a solution (transitions and firing modes must be chosen). The KRON interpreter offers a control policy object library but the user can design new control policies according to the application domain.

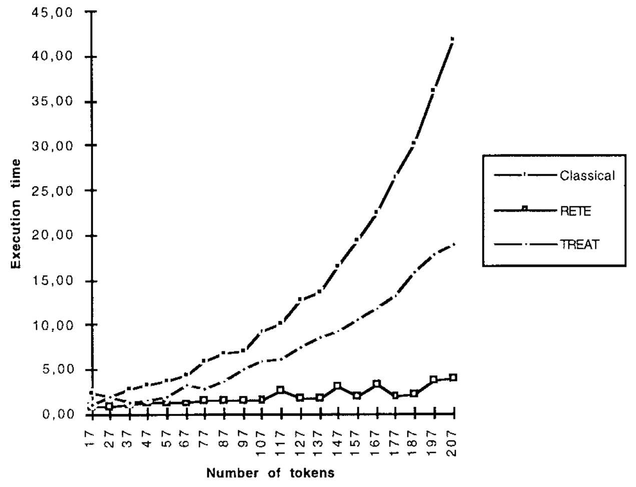


Fig. 13. Experimental comparison of Classical, RETE and TREAT approach. Time in seconds corresponds to 300 cycles of a KRON model for the world blocks.

3) Firing Phase: A transition firing with respect to a firing mode (pattern of the n level) carries out the following steps.

a) The calculation tree is traversed backward from the consistent binding (firing mode) to the tokens used to generate it. Each of these tokens is then withdrawn from the input places and a tree updating is performed in all the transitions sharing these input places.
b) The piece of code associated to the firing mode is executed.
c) The tokens specified by the output arcs of the transition are added to the output places and a tree updating is performed in all the transitions having these places as inputs.

4) Efficiency Issues: Classical implementations of HLPN's interpreters have emphasized techniques for fast selection of transitions that may be enabled. In situations with medium and high marking levels, an approach based on RETE normally seems more efficient because it avoids recalculations and it considers also transitions that are completely enabled. However, McDermott, Newell, and Moore suggested in [46] that the RETE match algorithm may be inappropriate when a completely recalculation of firing modes implies smaller number of operations than the number of operations involved in keeping the information about the tests performed in each cycle. The cost of removing a datum from the working memory and the size of the saved information in the join tree become the main drawbacks of the original RETE algorithm. Miranker developed the TREAT algorithm to avoid those kinds of drawbacks [47]. The TREAT algorithm is similar to RETE but does not save the result of partial matching between cycles, this means that it does not use the join tree. The KRON approach avoids the cost of removing a datum in a different way from TREAT, because it is done by linking the patterns to their ancestors and successors.

Fig. 13 illustrates an experimental comparison of three different interpreters of a KRON model based on

1) classical interpretations of HLPN's;
2) the RETE;
3) the TREAT approach.

The example model was made using a model of the typical world block problem [48]. Fig. 13 shows that the KRON approach based on the RETE algorithm does not suffer the combinatorial explosion that an increment of the size in the number of tokens entails. (KRON interpreter has been implemented in KEE, on top of Common Lisp, running on a SUN workstation).

VI. Relations to HLPN's

The Petri net underlying a KRON model can be considered as a subset of a HLPN with a special syntax. However, there still exist restrictions that are introduced to improve the modeling and simulation capabilities of HLPN's 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 al.* [49], which combines HLPN's with modeling by direct programming, than to works that extend the HLPN formalism. Cherkasova points out that a straightforward modeling of large systems by means of nets can result in practically nonexecutable models due to the intensive testing of firing modes for a larger number of transitions and the huge number of possible bindings. For this reason, they proposed using nets only when necessary, representing the system concurrency, and directly programming other procedures of searching, matching, sorting, etc.

Following this pragmatic approach, the HLPN formalism is not extended, but really, it is constrained to use simpler expressions. A KRON net differs from HLPN's (as defined in [50]) in the following restriction: the number of tokens added or removed to or from a given place is always the same for different occurrences, and it is constrained to one-per-arc condition. It is not an important restriction because more than one firing mode can be fired in each interpretation cycle. That means that conflicts can decide to remove a different number of tokens according to the defined control-policy.

Another important difference with HLPN's is introduced by the integration of HLPN's 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* [22]. Ubiquity concerns the token ability to have several occurrences in a marking. Formally, a KRON net with ubiquity is not a correct HLPN 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

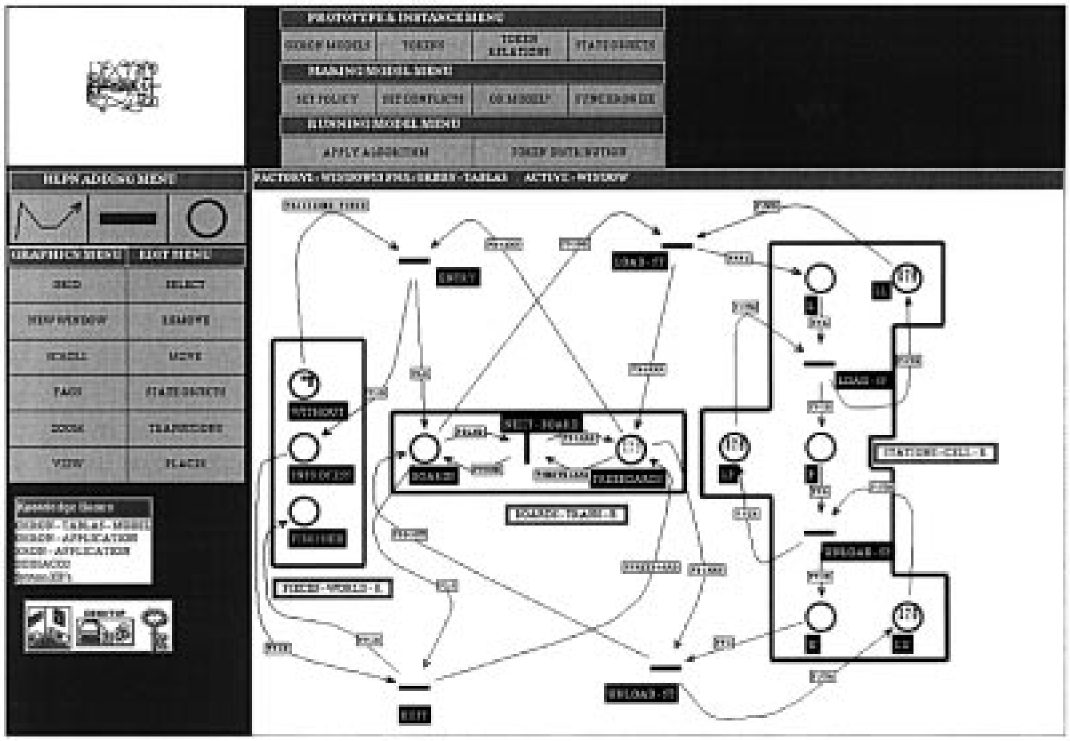


Fig. 14. Appearance of the KRON editor/animator.

the correctness of a system representation theoretically [51]. KRON allows the modeler 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.

VII. Development Environment Prototype

The software development of complex SED's requires tools to facilitate the modeling task and to manage a huge volume of different knowledge. A prototype of a simulation tool with graphical display and animation facilities has been implemented in KEE [30] running on Sun workstation. KRON is used as the kernel in the environment and provides a generic knowledge representation schema.

The user friendly interaction is carried out by a graphic interface based on menus and windows accessed to through the mouse. KRON interfaces are implemented upon KEE facilities to develop graphical interface. KEE provides interfaces to display a knowledge base organized in an inheritance network, and allows fast access to all information. Any frame may be picked out from such a graph, its contents examined, and if other frames are referred to as slot values, their contents can be examined as well.

In addition, the graphic module offers a KRON net editor/animator. It allows a graphical representation of the behavior based on the HLPN formalism. Dynamic objects are represented as bordered Petri nets. As an editor, KRON constructs may be built, modified, and connected. As an animator, it can animate the model during its execution. Animation may be automatic when defined control-policies solve decision points, or manual when the user interacts with the animator to solve decision points. Fig. 14 shows the appearance of the KRON editor/animator.

As an example of application of KRON to a particular kind of SED's, a manufacturing-oriented tool has been built using KRON. Manufacturing intended KRON (MIKRON) provides specialized concepts for the modeling of manufacturing systems, such as resources and operations, and it covers tasks like coordination, scheduling, planning, simulation, etc. [52]. A graphical interface specializing in manufacturing systems has also been developed.

VIII. Conclusion

In this paper we have presented KRON, a knowledge representation schema for DES's. 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 high-level Petri nets 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 high-level Petri net formalism is included. HLPN's provide the mechanism to describe the internal behavior of dynamic entities and the interactions between them.

In the representation of discrete event systems, HLPN's lack methodological and data representation aspects. To deal with these deficiencies, the main approach adopted by researchers has been to increase HLPN's with other paradigms such as abstract data types, object-oriented concepts or entity relationship models. From a historical point of view, the work on integration has evolved to the integration of objects and Petri nets in two different ways: the use of objects as tokens inside a Petri net and the use of Petri nets to represent the internal behavior of objects. However, the first approach has evolved to incorporate also the second one. It can be established, as a conclusion, that the work on integration has evolved to obtain a fully integration of Petri nets and objects; KRON follows this approach.

On the other hand, most of the integration approaches extend the HLPN formalism. The approach adopted does not extend the HLPN formalism. The frame-based representation of KRON supports the data and methodological aspects with no need to extend the HLPN formalism. So, all the advantages of the use of this formalism can be profited from working with KRON.

KRON uses inheritance as a mechanism to share code and representation. The subtyping relationship has not been considered because there is not a clear choice between the different notions of subtyping that we have found in the approaches which integrate Petri nets and objects. KRON mitigates some of the effects of the inheritance anomaly allowing the appropriate separation of the synchronization code (enabling conditions) from the action (a piece of code) attached to transition objects. On the other hand, Petri nets have a guard based synchronization schema that eliminates the state partitioning anomaly.

HLPN's 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 is based on the similarities between the inference engine of a rule based system and the interpretation mechanism of a HLPN. The proposed technique makes use of an adaptation of the RETE matching algorithm, which is used in OPS5 rule based language to provide an efficient inference engine. As in RETE, the main idea is to exploit temporal data redundancies (coming from the marking that is not changed during transition firing). Our experimental studies have shown how good this strategy is, which makes the computational performance remain quite regular even with high net markings.

Finally, a prototype of a simulation tool with graphical display and animation facilities has been illustrated. The prototype has been implemented on top of a known knowledge engineering environment called KEE from Intellicorp.

References

[1] P. Varaiya and A. Kurzhanski, *Discrete Event Systems: Models and Applications, No. 103 in Lecture Notes in Computer and Information Sciences*. Berlin, Germany: Springer-Verlag, 1988.

[2] T. Murata, “Petri nets: Properties, analysis and applications,” *Proc. IEEE*, vol. 16, pp. 39–50, Jan. 1990.

[3] J. Colom, M. Silva, and J. Villarroel, “On software implementation of Petri nets and colored Petri nets using high-level concurrent languages,” in *Proc. 7th Europ. Workshop Application Theory Petri Nets*, Oxford, U.K., July 1986, pp. 207–241.

[4] *High-Level Petri Nets*, K. Jensen and G. Rozenberg, Eds. Berlin, Germany: Springer-Verlag, 1991.

[5] P. Huber, K. Jensen, and M. Shapiro, “Hierarchies in colored Petri nets,” in *Proc. 10th Europ. Workshop Application Theory Petri Nets*, Bonn, Germany, June 1989, pp. 192–209.

[6] R. Fehling, “A concept for hierarchical Petri nets with building blocks,” in *Proc. 12th Int. Conf. Application Theory Petri Nets*, Aarhus, Denmark, 1991, pp. 370–389.

[7] S. Christense and N. Hansen, “Colored Petri nets extended with channels for synchronous communication,” in *Application and Theory of Petri Nets, No. 815, Lecture Notes in Computer Science*. Berlin, Germany: Springer-Verlag, 1994, pp. 159–178.

[8] J. Vautherin, “Parallel systems specifications with colored Petri nets and algebraic specifications,” in *Advances in Petri Nets, No. 266, Lecture Notes in Computer Science*. Berlin, Germany: Springer-Verlag, 1987, pp. 293–308.

[9] J. Billington, “Many-sorted high-level nets,” in *Proc. 3rd Int. Workshop Petri Nets Performance Models*, Kyoto, Japan, 1989, pp. 166–179.

[10] W. Reisig, “Petri nets and algebraic specifications,” in *Theoretical Computer Science 80*. Amsterdam, The Netherlands: Elsevier, 1991, pp. 1–34.

[11] E. Battiston, F. de Cindio, and G. Mauri, “OBJSA nets: A class of high-level Petri nets having objects as domains,” in *Advances in Petri Nets, No. 340, Lecture Notes in Computer Science*. Berlin, Germany: Springer-Verlag, 1988, pp. 20–43.

[12] E. Battiston and F. de Cindio, “Class orientation and inheritance in modular algebraic nets,” in *Proc. IEEE Int. Conf. Systems, Man, Cybernetics*, Le Touquet, France, 1993, pp. 717–723.

[13] B. Kramer and H. Schmidt, ”Type and modules for net specifications,” in *Advances in Petri Nets*, K. Voss, H. J. Genrich, and G. Rozenber, Eds. Berlin, Germany: Springer-Verlag, 1987, pp. 269–286.

[14] W. Reisig, *Petri Nets: An Introduction*. Berlin, Germany: Springer-Verlag, 1985.

[15] D. Buchs and N. Guelfi, “CO-OPN: A concurrent object oriented Petri net approach,” in *Proc. 12th Int. Conf. Application Theory Petri Nets*, Gjern, Denmark, June 1991, pp. 432–454.

[16] K. Hee and P. Verkoulen, “Integration of a data model and high-level Petri nets,” in *Proc. 12th Int. Conf. Application Theory Petri Nets*, Paris, France, 1991, pp. 410–431.

[17] A. Dileva and P. Giolito, “High-level Petri nets for production system modeling,” in *Proc. 8th Europ. Workshop Application Theory Petri Nets*, Zaragoza, Spain, June 1987, pp. 381–396.

[18] A. Dileva, P. Giolito, and F. Vernadat, “Executable models for the representation of production systems,” in *Proc. IMACS-IFAC Symp. Modeling Control Technological Systems, IMACS MCTS 91*, Lille, France, June 1991, pp. 561–566.

[19] G. Berio, A. Di Leva, P. Giolitto, and F. Vernadat, “The m*-object methodology for information system design in CIM environments,” *IEEE Trans. Syst., Man, Cybern.*, vol. 25, pp. 68–85, Jan. 1995.

[20] R. Di Giovanni, “Petri nets and software engineering: HOOD nets,” in *11th Int. Conf. Application Theory Petri Nets*, 1990, pp. 123–138,

[21] W. G. Hood, “HOOD user manual, issue 3.0,” Tech. Rep., Europ. Space Agency, Noordwijk, The Netherlands, 1989.

[22] C. Sibertin-Blanc, “High-level Petri nets with data structures,” in *Proc. Workshop Applications Theory Petri Nets*, Finland, June 1985.

[23] M. Paludetto and S. Raymond, “A methodology based on objects and Petri nets for development of real-time software,” in *Proc. of IEEE Int. Conf. Systems, Man, Cybernetics*, Le Touquet, France, 1993, pp. 717–723.

[24] C. Sibertin-Blanc, ”Cooperative nets,” in *Advances in Petri Nets, No. 815, Lecture Notes in Computer Science*. Berlin, Germany: Springer-Verlag, 1994, pp. 377–396.

[25] M. Baldassari and G. Bruno, “PROTOB: An object oriented methodology for developing discrete event dynamic systems,” *Comput. Languages*, vol. 16, no. 1, pp. 39–63, 1991.

[26] G. Bruno and G. Maretto, “Process-translatable Petri nets for the rapid prototyping of process control systems,” *IEEE Trans. Software Eng.*, vol. 12, no. 2, pp. 346–357, 1986.

[27] C. Lakos and C. Keen, “LOOPN++: A new language for object-oriented Petri nets,” Tech. Rep., Comput. Sci. Dept., Univ. Tasmania, 1994.

[28] R. Bastide, “Approaches in unifiying Petri nets and the object-oriented approach,” in *Object-Oriented Programming Models Concurrency: Workshop 16th Int. Conf. Application Theory Petri Nets*, 1995.

[29] J. Bañares, P. Muro-Medrano, and J. Villarroel, ”Taking advantages of temporal redundancy in high level Petri net implementations,” in *Application and Theory of Petri Nets, No. 691, Lecture Notes Computer Science*. Berlin, Germany: Springer-Verlag, 1993, pp. 32–48.

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

[31] D. Bobrow and M. Stefik, *The LOOPS Manual: A Data and Object Oriented Programming System for InterLisp*, Xerox PARC, Palo Alto, CA, 1983.

[32] *Knowledge Craft Reference Manual*, Tech. Rep., Carnegie Group Inc, Pittsburgh, PA, 1986.

[33] R. Fikes and T. Kehler, “The role of frame-based representation in reasoning,” *Commun. ACM*, vol. 28, pp. 904–920, Sept. 1985.

[34] C. L. Dym and R. Levitt, ”Semantic nets, frames, and object-oriented programming,” in *Knowledge-Based Systems in Engineering*. New York: McGraw-Hill, 1991, pp. 129–161.

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

[36] B. Baumgarten, ”On internal and external characterization of PT-net building block behavior,” in *Advances in Petri Nets, No. 340, Lecture Notes in Computer Science*. Berlin, Germany: Springer-Verlag, 1988.

[37] Y. Soussi and G. Memmi, “Composition of nets via a communication medium,” in *Proc. 10th Europ. Workshop Applications Theory Petri Nets*, Bonn, Germany, June 1990, pp. 292–311.

[38] S. Christensen and L. Petrucci, ”Toward a modular analysis of colored Petri nets,” in *Applications and Theory of Petri Nets, No. 616, Lecture Notes in Computer Science*. Berlin, Germany: Springer-Verlag, 1992, pp. 113–133.

[39] J. Rumbaugh, “Relations as semantic contructs in an object-orientated language,” in *Proc. ACM Object-Oriented Programming Systems, Languages, Applications, OOPSLA’87*, Oct. 1987, pp. 466–481.

[40] C. Forgy, “A fast algorithm for many pattern/many object pattern match problem,” *Artif. Intell.*, vol. 19, pp. 17–37, 1982.

[41] L. Browston, R. Farrell, E. Kant, and N. Martin, *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming*. Reading, MA: Addison-Wesley, 1985.

[42] G. Bruno and A. Elia, “Operational specification of process control systems: Execution of prot nets using OPS5,” in *Proc. IFIC’86*, Dublin, Ireland, 1986.

[43] J. Duggan and J. Browne, “ESPNET: Expert-system-based simulator of Petri nets,” *Proc. IEEE*, vol. 135, pp. 239–247, July 1988.

[44] R. Valette and B. Bako, “Software implementation of Petri nets and compilation of rule-based systems,” in *11th Int. Conf. Application Theory Petri Nets*, Paris, France, 1990.

[45] G. Harhalakis, C. Lin, L. Mark, and P. Muro-Medrano, “Information systems for integrated manufacturing (INSIM)—A design methodology,” *Int. J. Comput. Integr. Manuf.*, vol. 4, no. 6, 1991.

[46] J. McDermott, A. Newell, and J. Moore, “The efficiency of certain production system implementations,” in *Pattern-Directed Inference Systems*. New York: Academic, 1978.

[47] A. Miranker, “TREAT: A new and efficient match algorithm for AI production systems,” Ph.D. dissertation, Dept. Comput. Sci., Columbia Univ., New York, 1986.

[48] N. Nilsson, *Principles of Artificial Intelligence*. Berlin, Germany: Springer-Verlag, 1982.

[49] L. Cherkasova, V. Kotov, and T. Rokicki, “Modeling of industrial size concurrent systems,” in *Applications and Theory of Petri Nets, No. 691, Lecture Notes in Computer Science*. Berlin, Germany: Springer-Verlag, 1993, pp. 552–561.

[50] K. Jensen, *Colored Petri Nets: Basic Concepts, Analysis Methods, and Practical Use, EATCS Monographs on Theoretical Computer Science*, W. Brauer, G. Rozenberg, and A. Salomaa, Eds. Berlin, Germany: Springer-Verlag, 1992.

[51] B. Meyer, *Object-Oriented Software Construction*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

[52] P. Muro, J. Villarroel, J. Martínez, and M. Silva, “A knowledge representation tool for manufacturing control systems design and prototyping,” in *Proc. 6th IFAC/IFIC/IFORS/IMACS Symp. Information Control Problems Manufacturing Technology, INCOM’89*, Madrid, Spain, Sept. 1989, pp. 585–590.

Pedro R. Muro-Medrano received the M.S. and Ph.D. degrees in electrical engineering from the University of Zaragoza, Spain, in 1985 and 1990 respectively.

He was a Graduate Research Assistant in the Department of Electrical Engineering and Computer Science, University of Zaragoza, from 1985 to 1987. He was a Visiting Scholar at the Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, and a Visiting Research Associate at the Systems Research Center, University of Maryland, College Park. He joined the faculty of the University of Zaragoza as an Assistant Professor in 1989, and since 1992 he has been an Associate Professor. His research interests include object-oriented software engineering and application of object-oriented methods to distributed information systems and geographic information systems.

José A. Bañares was born in Zaragoza, Spain, in 1966. He received the M.S. degree in industrial–electrical engineering and the Ph.D. degree in computer science from the University of Zaragoza, in 1991 and 1996, respectively.

Since 1994, he has been an Assistant Professor in the Computer Science and Systems Engineering Department, University of Zaragoza. His research interests include Petri nets, object-oriented modeling, and artificial intelligence. His recent work has been focused on the integration of high-level Petri nets with the object model, and algorithms for the efficient interpretation of high-level Petri nets.

José Luis Villarroel was born in Huesca, Spain, in 1961. He received the Ph.D. degree in industrial–electrical engineering in 1990 from the University of Zaragoza, Spain.

He is currently Associate Professor in the Department of Computer Science and Systems Engineering, University of Zaragoza, where he is in charge of courses on control and real-time systems. His research interests include Petri nets, object-oriented modeling, robotics and real-time systems. In particular, he is currently working on the application of time Petri net techniques to the development of real-time systems.

IEEE TRANSACTIONS ON

SYSTEMS, MAN, AND CYBERNETICS

PART A: SYSTEMS AND HUMANS

A PUBLICATION OF THE IEEE SYSTEMS, MAN, AND CYBERNETICS SOCIETY

MARCH 1998 VOLUME 28 NUMBER 2 ITSHFX (ISSN 1083–4427)