Morphic Nets: Model Based Design Diagrams

P. C. P. Bhatt* Wolfgang Mueller†

Abstract

Morphic nets (MN) are net-based diagrams that offer a model based design framework. MNs combine the inherent advantages of a net structure with concepts from programming languages to model morphism (or change in shape) of data as it happens. We introduce MN extensions with some design rules and briefly compare and contrast MNs with UML and Coats-Mellon operational specifications formalisms.

Introduction and Rationale

The raison d'être for model based design is the designer’s preference to employ known structures and use them as models. One of the most popular modelling frameworks is Petri-Nets (PN) [1]. It can faithfully capture a partial order of events (and states) manifested in a system. PNs are capable of abstracting concurrent control flow and can explicitly depict synchronisation amongst arbitrary subsystems very well [2]. However, PNs lack ability to depict semantics of the data being processed. Nevertheless, there are some PN extensions which partly cover data processing capabilities [4, 5].

Morphic Nets (MN) were formally presented in a co-design conference [6]. Concepts similar to MNs were used in enriching the token structure of Pr-T nets [3]. In appearance, MNs are like PNs but they differ from it substantially. MNs are data-flow centered. The basic premise is: when we process data, we may output data in a form (or shape) different from the input, i.e., data processing is data morphism. MNs simply depicts the morphism on data as it happens.

In the next section we introduce the basic MN definitions and notations with few examples. In section 3 we introduce new extensions for distributed applications with message-based operations. Section 4 compares MNs with other approaches, i.e., UML and Coats-Mellon.

2 Morphic Net: Infrastructure

A major asset of PNs is visualisation through its graphical representation [1]. As depicted in fig. 1, MNs too have a similar representation. Places are shown as circles. However, we may have two kinds of transitions which are distinguished by using a straight or a wiggled line segment (fig. 1-b). Directed arcs capture the relationships. In contrast with PNs, places in MNs have a role of individual “data-holders” and the transitions have a role of “data-transformers” (or function evaluators). Tokens have a class (or a data type) and a value, i.e., tokens are objects with a state and some properties. Such an interpretation

* Kochi University of Technology, Japan. Email: bhatt@info.kochi-tech.ac.jp
† Paderborn University, Germany. Email: wolfgang@acm.org
‡ Bhatt first presented some loosely structured concepts during seminars held in 1992 at Univ. Paderborn, Univ. Dortmund and McGill University.
has two advantages. Firstly, it seamlessly integrates data flow and control flow. Secondly, the modelling paradigm now spans enough breadth. It brings MNs closer to a proposed implementation without in any way compromising on its abstraction capabilities. On to definitions now.

**Universe of values, classes and tokens:** We basically envisage a set which is closed under computations [10], i.e., we limit to data-morphism that stipulates input from a set within the universe \( V \) and produce values which are also within \( V \). A class \( C \) defines a subset within \( V \). A token \( k \in K \) in MN is specified by a class value pair.

**Example:** Let \( V \) cover the sets of integer \( I \) and real values \( R \). Then tokens may have forms like \( I : 5 \) and \( R : 7.2 \).

**Morphic Net MN [8]:** A Morphic Net MN is defined as a graph which has places or transitions for nodes, and arcs to define relations between places and transitions. The transitions may be polymorphic or monomorphic transitions.

Graphically we may show \( n \) circles (one for each place), one straight line segment for each polymorphic transition in \( T \), one wiggled line segment for each monomorphic transition in \( S \). Arcs in \( A \) connect circles with transitions to capture the desired relation. Not all MNs may be meaningful. A well-formed MN (defined next) may be meaningful in the sense of a valid token marking.

**Well Formed Morphic Net MN:** An MN with following constraints is an MN.

1. **Exclusivity constraints**: A node can either be a place or a transition.
2. **Connectivity constraint**: If a well-formed MN has a monomorphic transition in it then there are at least as many places as there are monomorphic transitions.
3. **Input output constraints**: As is obvious, arcs may be incident to (from) places from (to) transitions. In a well-formed MN connectivity can be expressed in terms of input and output functions. We shall use \( I \) for input mapping and \( O \) for output mapping and subscripts \( s \) for monomorphic transitions and \( t \) for polymorphic transitions.

   (a) **Constraints on \( s \):** Each monomorphic transition has a distinct input place and a different, distinct output place. Two monomorphic transitions do not share an input place or an output place. However, an output place of one monomorphic transition may be an input place for another monomorphic transition.

   (b) **Constraints for \( t \):** We ensure that input (or output) places connect to polymorphic transitions with at most one arc. The motivation for this restriction was to do away with the need to handle bag theory and define the extent of non-determinism.

Polymorphic and monomorphic transitions cannot share a place either for input or for output. However, an output place from a polymorphic (monomorphic) transition can be an input place for monomorphic (polymorphic) transition. We can also define a dual set of input and output.
functions that can identify input and output transitions w.r.t. places. Hereafter we consider only well formed MNs.

Net Marking: A marking defines the distribution of tokens in places of an MN. We now examine transitions and their firing in some detail.

An Enabled Polymporphic Transition: A transition $t \in T$ is enabled when input and output places incident to and from $t$ meet all the conditions given below.

1. Ensure that a data from a subclass qualifies within a place to provide an input to a transition and ensure qualifying token’s uniqueness.
2. Ensure preservation of token class uniqueness with regard to the new tokens produced in output places. If input places are distinct from output places then this uniqueness property ensures that only those transitions fire (evaluate a function) which can produce a token (generate a data item) of a class not already present in output places. When an output is fed back the place first loses its token on firing and gets it after firing.
3. Ensure that if there are arcs emanating from an output place, then the class associated with these arcs cover the class of tokens that an output place may receive, i.e., these places receive tokens which can be consumed.

MNs as defined above, are class-safe. In a finite MN, each place has no more than one token of a distinct class. With a closed universe of values, and a finite number of classes the MNs are bounded and therefore, MNs are class-safe. Of course, we assume that we are dealing with values that have finite representations only.

Firing Function of a Polymporphic Transition: When a polymorphic transition in $\mathcal{M}$N is enabled it may fire. Upon firing, this transition evaluates its user defined firing function for a new marking of $I$ and $O$.

In fig. 2 we show a well formed MN. This is an illustration of the kind of abstractions one may use without giving any implementation details.

Firing of Polymporphic Transitions: When an enabled polymorphic transition fires, new marking occurs. Some places lose their tokens and some acquire new tokens. The firing rules require identification of input tokens followed by removal of these tokens and evaluation of firing function associated with the transition and consequent insertion of new tokens. The rules are similar to PNs except that tokens have data interpretations.

The Role of Monomorphic Transitions: The monomorphic transition was primarily conceived to permit type casting. Quite often we need to think of a data form either as a specialisation or as a generalisation of yet another data form. Basically, a monomorphic transition serves that need. It can receive data in one form and present it in a form which is either a generalisation or a specialisation. With the input and output on monomorphic transitions, only one object undergoes a transformation of form.

In case of generalisation, upon firing of $s$, the place $p_e$ will always receive a token of class $c_{eq}$. However, in case of specialisation, a token is produced only
if \( v_i \in c_{o_s} \). In other words, if the corresponding partition to which \( v_i \in g_s \notin c_{o_s} \), i.e., \( v_i \notin c_{o_s} \), then transition \( s \) consumes the input token but no token is generated in the output place.

**Consumable Tokens**: In a net marking of \( MN \) a place \( p \) has a consumable token \( k \in K \) of class \( c_{ph} \) if it has the property that there \( \exists t \in (S \setminus T) \) with \( p \in I(t) \) and the arc \( a \) from \( p \) to \( t \) has a class marking \( c_a \) such that \( c_{ph} \subseteq c_a \).

**Valid Marking**: A marking of \( MN \) is valid if (i) at least one transition in \( MN \) is enabled and (ii) places have only potentially consumable tokens.

For a well formed \( MN \), with a valid initial marking, there is at least one firing possible. With the properties associated with firing functions, we expect subsequent markings to generate only potentially consumable tokens. The process (or firing sequence) can continue till either we choose to terminate it or there is no enabled transition left.

A short example demonstrates use of monomorphic transitions for type casting as well as how we can use type casting mechanism to realise conditionals. In fig. 3, the place \( P1 \) receives a boolean value which is duplicated in \( P2 \) and \( P3 \). These are then specialised for being true or false. Token propagation continues through either place \( P4 \) or \( P5 \). This gives the MNs a capability of conditional selection. With a feedback and monomorphic transitions MNs can model iterations. The boxed MNs in fig. 3 are compact abstractions of large MNs. In general, monomorphic transitions may be used to effect computations for each partition in a class.

### 3 Conventions, Syntactic Sugar and Extensions

In adapting conventions or using minor extensions we retain the basic flavour of the original MN. In fact, the extensions do not add to empowerment of the net but do help in realising more comprehensible nets. Most of these extensions cater to modelling message-based distributed applications and also to abstract large monolithic MNs in to compacted forms.

**Figure 4: Morphic Net Abstractions and Conventions**

- **4a**: TO GET MULTIPLE PARTITIONS OR TO GET IF THEN ELSE OR CASE OR WHILE LOOPS
- **4b**: \( d \) DENOTES THE PERIOD BY WHICH THE TRANSITION’S FIRING IS DELAYED
- **4c**: ABSTRACTED TO THE PLACE THAT STORES A CONSTANT CAN BE MODELED WITH AN EMANATING DASHED WITH NO FEEDBACK
Abstractions for large monolithic MNs: There are two levels in abstractions. One, that compacts monomorphic transitions. The other, that compacts smaller MNs, each corresponding to a subtype case as shown in fig. 4a.

Interconnection Protocol: We recommend that a uniform interconnection convention be adhered to by design teams. Suppose MN A connects to MN B, then we recommend that an output transition of A connects to an input place in B. This convention has the advantage that it mimics the notion that an event in A is being recognised as a change in the environment of B. This recognition leads to a new state in B. Our experience in translating requirements to check out consistency in specifications suggests that having a uniform interconnection protocol helps immensely in translating to an implementable form of MNs. Thus MN can, and should serve, as an intermediate form between a system’s specification and implementation.

Time in MNs: True time specification is a difficult issue. Timed PNs and their extensions using temporal logic and unit time delay have been described in the literature abundantly [11, 12]. However, for systems that are designed using discrete event methodology often, if not always, a simple time model suffices. For example, we assume that firing is instantaneous unless stated otherwise. So the output is immediate upon firing. The duration of firing and delay of token delivery is immediate, i.e., basically zero. However, in order to achieve a consistent time model, we assume it to be infinitesimally small. In case the firing has a duration or the output delivery is delayed then the time is explicitly stated by associating a delay parameter with a transition. The interpretation is that after removing input the firing lasts the assigned period of time, say \( d \) time units as shown in fig. 4b.

Constants in MNs: Often designers need to define constants. Constants should be read only. As the mechanism to read leads to consumption of a token. Any syntactic extension that permits a non-destructive read out of token is simply a syntactic sugar. We simply make the read-out arc a dashed line as shown in fig. 4c.

\[ \text{Figure 5: Model of a Battery-Driven Vehicle} \]

AGV Example: Suppose we need to assign a battery driven automated guided vehicle (AGV) on a job. Clearly, the battery should have sufficient charge. We depict the situation in fig. 5. Place _StateInfo holds all the state information including current charge level. The token in place _StateInfo\(^2\) shall be a set of data items that describe AGV’s state in all its facets. The non-destructive read-out gets this information. The MN in part II calls out that information which is needed to check if the battery charge is low. Only when the charge is determined to be adequate, the AGV can be declared to be ready. This requires that we delay the operations that are associated with destructive readouts. In our case we delay firing any of the transitions in part I till check on battery charge level is finished in part II. The arrangement in fig. 5 ensures that there has been a readout for the battery charge well before the place _StateInfo loses its token, or is considered to be ready (with a token in the output place _VehicleReady). The example shows how a race between the two arcs from place _StateInfo has

\(^2\)This basically compares to the concept of delta delays in the IEEE Std hardware description language VHDL.

\(^3\)From now on we shall assume that the token types are defined by the context of use.
been preempted in favour of a non-destructive readout. Fig. 5 also shows an indication of battery level to the “environment”.

A Design Rule: Let us go over this example once more. For instance, why delay d should be incorporated? Suppose part I and part II in fig. 5 are being designed in isolation for a later integration. The specification now needed is: whenever the StateInfo is updated, the second part should determine if the charge on battery level is adequate within a specified time. This time can be then used as the required delay d in part I. The absence of d could result in an undesirable race condition. We recommend use of such a practice whenever a token information is required for a ready out or use in branches that fork now and merge later.

![Diagram of message processing](image)

**Figure 6: Message Processing MNs**

**Support for Distributed Applications:** An MN being from PN family, is normally limited to communication using places and transitions connected by arcs. This means that every broadcast or every possible multicast need be shown explicitly in the connected graph structure. This clearly is a very contrived form of representation. More so it is far removed from reality as most often there is an identified port for communication. To have a meaningful representation of the scenario, communicating MNs (CMNs) **identify a message as a data** and the **process of sending a message as an event**. CMNs use a special kind of place to receive messages and a special kind of transition to send messages as shown in fig. 6. The forked lines on communicating transitions and places are to capture the notion of antennas to establish communication amongst sending transition(s) and receiver place(s). In fig. 6a we show receiver end of message processing. Message tokens may encompass sender identification, and a number of other token data segments. In general, a message type may have subtypes to identify special characteristics of the message. Note the figure also uses a filled rectangle to show abstraction of a transition when we can easily identify all its input and output relationship or the associated complex computation. In fig. 6b we show the sender end of message processing.

![Diagram of basic template for modelling communication](image)

**Figure 7: Basic Template for Modelling Communication with Environment**

**Communication with environment (A design convention):** When
many devices (or entities) interact, devices may receive signals or messages from other devices. Thus, for a device a message coming from another device corresponds to sensing changes in the environment. Having a convention, like the one in figures 4, is most useful in conceptualising interactions. It ensures

1. that the environment and the device are shown interfacing each other across a dashed vertical line. This helps to instantly recognise which arcs come from the environment. The device, or entity, occupies the left side of the figure and the communication from the environment is received always from top right and leaves at bottom right part of the diagram.

2. that the messages adhere to the representations discussed above, i.e., they are sent from transitions that are specifically identified to communicate a signal or a message to be received at a designated place.

**Figure 8: Interconnecting Major States**

**Identifying Top Level Operations (A Design Rule for Top Down Design):** Before we employ the concepts introduced for communication in the design of a distributed systems we shall briefly discuss a simple design rule. It is a good design practice to identify major operations within a system. Let us again consider the AGV in fig. 5. The AGV operations broadly cover the activities like getting its battery charged, moving to a location to take a load etc. For instance, we may consider the AGV in motion. We may identify major state (defined later) “moving” with a moving AGV. Similarly, when its fork-lift is loading an object we may say it is in a major state “loading”.

The following points are noteworthy in this regard. Once the goal is achieved like the destination is reached in case of “moving” or the AGV is loaded in case of “loading”, we may have to switch to another major state. Once we have identified all the major states it is easy to build an operational cycle as a state transition diagram with interconnections amongst major states. However, a major state may have several intermediate states within it. For instance, while AGV is moving, the position is changing. So while the major state moving holds over a duration of time many minor states may manifest themselves. It is a good design practice to group a set of related dynamics within a system with a major state.

**Evolving a Design in a Top Down Manner (A Design Rule):** Suppose we consider the major state “moving” for AGV. First, we may assume an auxiliary MN to capture the dynamics within this major state. Now we can associate a transition to and a transition from this major state as shown in fig.8 along with the connections to our auxiliary MN that captures all the minor states. In fig. 8 we assume that the MN to the left shall become available when those details are worked out. The long transition bars ensure transition to, and transition from, the major state given by the place on the right. Once the major states have been identified it is easy to identify a state transition diagram that will describe an operational cycle. For instance, for AGVs we have an operational cycle which covers major states like: IDLE , MOVING, CHARGING the
battery, LOADING / UNLOADING a work piece. We can connect these using the mechanism just described.

AN ACTIVITY AND A MAJOR STATE: In an observed behaviour of a device, it may be possible to find a pattern within a set of related events. Further, if this event pattern manifests itself only under certain specific conditions within a device then this set of events is regarded as “an activity” and is given a name. Thus the name of an activity identifies it with a set of events within a device and the specific set of conditions is abstracted as a major state. We shall illustrate a full example for activities and major states.

TOP DOWN DESIGN (AN EXAMPLE): We shall use the conventions and design rules as a conceptual aid to evolve our design in a top down manner. In an automated factory, AGVs receive requests from manufacturing stations (MS) to deliver feed for them. To collect feed for the first MS in the process, the AGVs move to an input station. The operations of an input station and its interaction with an AGV are defined as follows.

- Input station receives a request from a AGV to collect an input.
- Input station acknowledges the request and sends its state. We assume it is free to receive an AGV for input, i.e., the state is not\(\not\) engaged.
- Input station verifies the AGV’s id. (We assume correct id for now.)
- AGV docks with the input station to collect some input.
- AGV is loaded with the feed.
- Input station sends a signal to the AGV when loading is finished.
- AGV starts to undock.
- Input station senses undock and declares itself not\(\not\) engaged.

The notions such as loading or docking can well be regarded as major states of the input station. As we explained earlier, each such major state may have more detailed minor states within it. For instance, there would be a sequence of steps within docking starting with sensing the approach of AGV and consequent movement of a stepper motor to engage the AGV. All the states of stepper motor are to be within the major state docking for the input station. Clearly, by our convention a major state identifies with an activity. The activity may have several events within it. However, from the point of view of the main device these events take place only when the device is in a well identified major state. The identification of a set of major states of a device helps in developing a system design in top down manner. For instance, based on the specifications stated above we can easily set up a MN that captures all the transitions from and to major states. The details of each activity can be filled out later. Such a step in a design exercise is extremely helpful in rapid prototyping.

Fig. 9 shows the behavioural model of an input station. We can arrive at fig. 9 as follows:

1. Draw a vertical dotted line to separate environment from local behaviour.
2. From the specifications identify the local behaviour of a device. From the stated behaviour of an input station, we know that it may receive a message, communicate its state, await its docking and subsequent loading of feed. Following that it sends a signal to undock, gets disengaged and is ready to receive another AGV.
3. Associate a major state with every one of the activities. Note that all the major states, except the initial one, lie between two long transition bars.
4. Associate a smaller morphic net with each of the identified major state. The smaller nested MNs are organised such that exiting the corresponding set of events results in exiting the major state as well. In fig.9, these nets correspond to the references made to \(L0\) through \(L5\).
5. Identify starting states within each major state. For each of the sub-MNs (in the last step), identify likely starting states as well as, if possible, the other states of interest. In fig. 9 we have identified states like \texttt{Inp\_wait}, \texttt{Load\_begin}, \texttt{Sense\_un\_dock} etc.

6. Now fill in the details by designing each of the sub-MNs. The example illustrates use of rapid prototyping methodology for system design. Recently [7] a major system design description was specified using MNs. Dynamic MNs: One of the interesting facets of permitting data typing is to think in terms of meta-morphic options. Can we generate nets that are an outcome of a generic formulation? In the simplest case we may be able to have a standard network with a parameter to generate a specialisation. This is clearly feasible within the polymorphism that is achievable in MNs. At one stage during our studies [7, 14] we wondered if we could generate entities like AGVs on demand, i.e., create a data object when needed. Clearly, as each AGV has its own MN description, this would lead to a variable net structure. Distributed systems often have such a requirement.

While the Guide [8] discusses many other issues, we can only mention them here. These issues include signatures for MNs, message structures and protocols, data updates and management using a MN manager. In system design it is important to have a right sequence of operation and communication. There is a need, therefore, to associate some form of a scheduling policy emanating from the given specifications. The MN manager module should be able to select firing of the transitions in accordance with such a schedule.

4 MN and other modelling paradigms

By now we have seen all the current extensions available in MNs to support system design. We have made two claims. One, MNs aid in evolving a system design. Two, we claim that it is data centered. It would, therefore, be worth our while to directly compare MN with UML (Universal Modelling Language). UML is currently the most sought after data centered model based system design language. It is deeply bound to, and embedded in, the object oriented modeling.
paradigm [13]. The comparison is warranted on the grounds that MNs too support classes, polymorphism, and inheritance and a data centered model based system design framework. We shall also look at another promising methodology, i.e., Coats Mellon (CM) approach [14]. However, as the CM approach and parts of UML could be well used as complementary means we feel that the first one fits much better for most of our current applications which are in the field of distributed systems. Nevertheless, we should note here that, in contrast to MNs, both come with very little or no semantics, respectively. Our following observations here are based on [15] in which we elaborate on the exercise in [7]. The Guide [8] has several other comparisons.

4.1 MN and UML

UML has a very rich notational framework which is deeply embedded in, and tied to, object oriented methodology. UML community has identified a “metamodel” which is a generic model, in the sense that all UML descriptions actually emanate from this model.

UML expresses models through a set of diagrams called class diagram, sequence diagram, and collaboration diagram finally, arriving at a fairly detailed activity and/or state diagram. At the top most level we have class diagrams and closer to an implementation would be activity diagrams and state diagrams. Class diagrams identify classes, their attributes and operations. To that extent, UML identifies objects explicitly. MNs do not have a class diagram like structure. Their arcs basically capture the possible interactions between data items. At the level of class depiction UML is structurally more direct in expression of all the objects that interact. However, it must be remarked that identification of the classes and interactions amongst them may require considerable work in the background like studying a process in all its detail [13]. MNs can capture objects as a consequence of their data definition. To that extent MNs are not necessarily bound to OO modeling methodology. However, top down design with MNs requires information akin to the domain model [13]. This identifies the data used to model the behaviour of entities and their respective activities (i.e., major states and its encapsulated event patterns).

Sequence diagrams explicitly identify the interactions between objects and classes on vertical time lines which advance from top to bottom. It is possible to see concurrency across the time lines. Collaboration diagrams, on the other hand, more explicitly identify all possible interactions amongst objects or classes. MNs identify collaborating objects by ensuring that they share a transition place pair that communicate. The partial order of events and concurrency is explicit. If we were to use UML sequence diagrams then for the example in fig. 9, we shall have the collaboration recognised between AGVs and input stations from the way arcs would interconnect these systems.

An activity diagram in UML is a discrete event diagram that identifies the conditions and events which they can trigger. It can resemble a PN in respect of a net like structure with nodes, transition bars and arcs. However, the transitions are used as synchronising conditions. It does not use any explicit tokens to identify conditions that trigger an operation. By itself, it is devoid of details of any concrete structure of the data items involved, though one can obtain this information by following through right up to class diagrams. This is the closest UML ever gets to an implementation. Note that the major state interconnection captured in a diagram that describes a MN-based design has a correspondence with state diagrams in UML. One can arrive at a UML diagram from a given MN diagram in two steps: (i) For every major state, identifying an activity in
a given MN, draw a state with the same name for UML state diagram. (ii) Interconnect the UML states with arcs corresponding to each of the transitions that connect major states in MN. Name the arc in UML with the event that is identified with the corresponding transition in MN.

MNs on the other hand have very rich data content at this stage of design. With a strict protocol on the way the data interactions happen, MNs certainly offer some scope to check out on correctness of specifications. In the exercise [?] which we carried out we came across a few clearly identifiable dead-lock conditions. We believe it was largely due to the MNs explicit data models and their interactions.

At this stage a remark that applies to both UML and MNs is in order. In using either of the two, we require some initial planning. UML uses use-case (UC) and class-responsibility-collaboration (CRC) charts. Though MNs evolve from specifications, using intermediaries like UC and CRC charts can be very useful.

![Actor diagram](image)

**4.2 Coats-Mellon Approach**

During our investigations we found that one may use the operational specifications as defined by Coats-Mellon approach as a step before drawing MNs.

In the operational specifications by Coats and Mellon (CM Operational Specification), the initial step is the identification of actors, within a system. Basically, the system is modelled around the responses to actor generated stimuli. Actors usually are identified with the devices and entities that interact. To that extent operational specifications using CM approach stipulate a closed world with known actors. The inputs from environment in MNs have a correspondence with the stimuli received from another actor in the CM approach.

Like the UML notation, CM approach too requires a set of diagrams to be drawn with a specific set of notations as follows.

**CREATE AN ACTOR DIAGRAM:** This requires that the entities that play a role are identified and arrows depict the direction of communication. For the example in fig.10, the two actors would have a connecting bidirectional arrow.

**CREATE AN ACTOR INHERITANCE DIAGRAM:** This requires that we identify objects to inherit properties from generic actor class definitions.

**CREATE AN EVENT CATEGORY DIAGRAM:** This idea is very similar to the identification of an activity in MN design. Though it should be remarked that the event category lists the events and event sequences as a catalogue.

**CREATE AN ACTOR EVENT DIAGRAM:** The actor event diagrams are like the sequence diagramm of UML but often capture a feed back explicitly. The concurrency is evident from the time lines that run parallel. In fig. 10 we have shown a typical actor event diagram for an AGV docking with an input station where we first check the id of AGV and proceed to dock only when the id is ok else follow a sequence identified with an error recovery routine.

---

*In UML, actors are identified with each use case.*
CREATE SYSTEM RESPONSE DIAGRAMS: System response diagrams are high level descriptions and are more like a wish list for the final system. After having created these diagrams the CM approach recommends to validate the behaviour.

Clearly, in using CM approach, we are operating at a fairly high level and should have considerable clarity in respect of structural aspects of a design. Our recommendation is that CM approach be used in conjunction with MNs as MNs are strong on the behavioural aspects of system specification and much closer to a feasible implementation.

5 Conclusions
MN should be seen as a designer’s semi-formal abstraction tool which can support correct by construction capability. The basic rationale is to abstract the control flow and depict data flow semantics explicitly within a system. Two major consideration led to the choices made in defining MN structures. One was to serve the combined imperative need to support system design activity and the other was to support an automated verification at some later time. MN models are closer to implementation as MN tokens have a class and a value-based semantics in the same way as in a class inheritance-based typed programming language.

Acknowledgement
We are grateful to Prof. Franz J. Rannamig and Georg Lehnenfeld from Paderborn University for their continued support.

References