A model-based visualization technique for mechatronic systems.

Jochen Stier
Department of Computer Science
University of Victoria, Victoria, Canada
jstier@cs.uvic.ca

Jens Jahnke
Department of Computer Science
University of Victoria, Victoria, Canada
jens@cs.uvic.ca

May 6, 2003

Abstract

Three-dimensional computer graphics are a valuable tool for the design and analysis of mechatronic systems, such as manufacturing, robotics or transportation systems. Computer-generated models can present an intuitive and flexible interface to design the structure and dynamics of a system before costly physical prototypes are developed. However, the dynamics of mechatronic systems can be complex and it is challenging to create functionally correct visualizations. Techniques are needed to describe the behavior of a system and integrate it with a computer graphic model. Petri Nets are an effective formalism to model many types of dynamical system. Interpreting a Petri Net produces continuous and discrete values that can be used to control, or animate a computer graphic model. The proposed research investigates using Petri Nets as a visualization language for rapid prototyping of mechatronic systems.

1 Introduction

Mechatronic systems are hybrids of electronics, mechanics and software. They are deployed in increasingly complex applications, and it requires reliable and scalable methods to support their design and development. While current graphic modelling techniques routinely produce static models, it is still challenging to incorporate the dynamics that are required for a functionally correct visualization.

The nature of a Mechatronic system is determined by a combination of physical and computational processes. Modelling this type of system requires techniques that are well suited to describe both discrete and continuous processes. This paper presents an approach for simulating mechatronic systems uses a hybrid modelling language [1]. Hybrid systems are dynamical systems that consist of interacting continuous functions and discrete events. They arise from the interaction between discrete planning and continuous control algorithms and they are well suited to describe the computer aided processes of complex mechatronic systems [2]. A part arriving at a work cell, for example, is a discrete event with respect to production planning, yet it also triggers a continuous and timed manufacturing process.

Research in the field of computer science, and in particular control theory have produced a number of formalism that facilitate the modelling of hybrid systems [1]. Among those are Petri Nets, a graph-based language for concurrent processes [3]. Petri Nets provide an intuitive set of constructs to describe concurrency, synchronization and conflicts in discrete-event data driven processes. Extensions to the formalism that incorporate time provide the continuous variables necessary to evaluate the functions of a hybrid process.

Advantages of Petri Nets include that they are symbolic, concurrent, and favour modularitzation. Their graph-based nature allows for an intuitive and flexible user interface that supports the design and maintenance of complex models. Integrating computer graphic techniques with a modelling language like Petri Nets enhances the power of the formalism. Three-dimensional images convey information intuitively and provide a natural context for the design a system. These advantages result in additional cognitive support to the design process and ultimately lead to improvements that reduce development time and cost.

This paper presents *RealFusion*, a structured language that integrates Petri Net theory with computer animation techniques. This work has been inspired by earlier applications of Petri Nets in modelling and controlling mechatronic systems, e.g., [4]. A modelling tool has been created that supports RealFusion and presents an interactive user interface to produce complex visualizations. The following section introduces the constructs of RealFusion, such as Scene Trees and Petri nets. Section 3 outlines the RealFusion language and its constructs. Interpretation mechanisms and tool support for executing RealFusion models are discussed in Section 4. While sections 3 and 4 can be seen as presenting the syntax and semantics of RealFusion, respectively, Section 4 discusses pragmatic issues on how a user would use our approach. This is outlined with a sample case study. Section 6 compares this approach to related work and Section 7 closes with concluding remarks and gives pointers to future work.

2 Concepts

This section gives an overview of Petri Net theory and Scene Trees. In our approach, a Scene Trees models the three-dimensional structure of a system and a Petri Net models its dynamics.

2.1 Computer Graphics

Three-dimensional computer graphics are the result of rendering the mathematical representation of a scene onto a display device. An animation occurs by changing the parameters that define a scene between consecutive rendering steps, or frames. There are different strategies for how to organize and animate a three-dimensional scene [5].

2.1.1 Scene Tree

Scene trees are a common technique to organize a three-dimensional scene and address its content. They are the specification language or internal data structure for many virtual reality languages and rendering environments, including Open Inventor and VRML. A scene tree maintains a three-dimensional model as a tree of differently typed nodes, such as transformation matrices, geometries, material nodes and light sources. Each type of node contains fields that store the current state of the node. Transformations, for example, include fields that contain matrix components for rotation, translation and scale. For each frame of an animation the tree is traversed and the state and order of the nodes determines the resulting image.

In general, the structure of the tree describes the spatial organization of a scene. A book residing on a table, for example, is a child of the table. Changing the location or orientation of the table also affects the book. Rearranging the structure of the tree creates or destroys these dependencies. This process is analogous to assembling or disassembling objects. With appropriate scene trees it is possible to describe complex structures such as manufacturing facilities, including the arrangements of all moveable parts and the realistic representation of their shapes.

2.1.2 Animation

An animation occurs when the structure of the scene tree or the content of its nodes changes between rendering steps. There are numerous techniques that produce many kinds of animation effects [5]. B-Splines curves, for example, are common numerical method to describe the shape of an object. The shape changes with the location of the control points that define shape of the spline. Inverse kinematics is an example of an algorithmic technique that controls the joints in a kinematic chain. The algorithm automatically computes the angular velocities at the joints in such a way that it causes the chain to reach for a point in space. The Open Inventor architecture supports animation engines and interpolators that can be used to encapsulate arbitrary forms of animation.

2.2 Petri Nets

A Petri Net is directed graph composed of passive and active vertices. The passive elements are called *places* and the active elements are called *transitions*. Arcs connect transitions to sets of incoming and outgoing places. The places contain *tokens* that can cause transitions to *occur* and trigger state changes in the Petri Net. During an occurrence, a transition consumes tokens from its incoming places and deposits tokens on its outgoing places. A transition may only occur when there is enough tokens in its incoming places, enough capacity in its outgoing places and a guard function evaluates to true. Formally, a basic form of Petri Net is defined as tuple

$N = \langle P, T, A, w, k, m_0 \rangle$
P - set of Places
T - set of Transitions
A - set of Arcs: $A \subseteq (P \times T) \cup (T \times P)$
w - weight function of A: $w(a) \rightarrow N^0 : a \in A$
k - capacity function of P: $k(p) \rightarrow N^0 : p \in P$
m_0 - initial Marking: $m_0(s) \rightarrow N^0 : p \in P$

The set of tokens residing on all the places in a net at an instance in time is called a *marking*. An initial marking is the very first set of tokens that is specified as part of the model. All other markings thereafter are the result of occurring transitions. The set of possible markings and the corresponding transition occurrences describe a *reachability* graph. Based on this graph, there exist a number of formal analysis techniques that allow mathematical proofs about the behavior of the system. Some of these methods can detect deadlocks or unreachable states.

Figure 3 shows a simple Petri Net of two places (P_1 and P_2) and one transition (T_1). The constructs of the diagram are annotated with the appropriate functions. In this example T_1 is enabled and can occur. P_1 contains enough tokens to satisfy the weight of arc A_1 and P_2 has enough capacity to accept the weight of A_2. However, the transition can only occur once before P_2 is full.

Since their original inception by Carl Adam Petri in 1962, Petri Nets have evolved into a wide range of dialects. While state changes in the original Petri

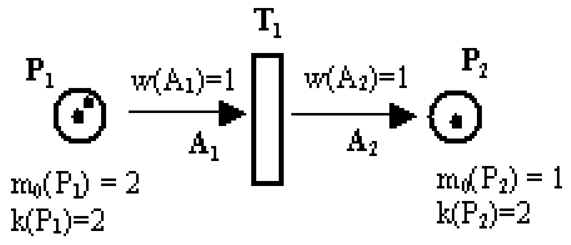


Figure 1: A simple Petri Net

Net are discrete and instantaneous, there are extensions that incorporate time with places, transitions or arcs. Other extensions, such as colored or object-based [6] Petri nets treat the originally type-less tokens as abstract data types or objects. Hierarchical Petri Nets [7] modularize the otherwise flat net structure by embedding separate Petri Nets within places or transitions, or by treating entire nets as tokens [8]. Continuous and Hybrid Petri Nets [4] incorporate special token and transitions types that are continuous rather than discrete.

In general, the two-dimensional structure of the net defines the conditions under which events occur, and describes in discrete steps how a process progresses over time. It is possible to model a hybrid process by attaching continuous functions to timed constructs. The functions become active as tokens travel and the timed variable continually evaluates the function. Following is an introduction of the Petri Net constructs and some of the possible extensions to their functionality.

2.2.1 Guard functions

Transition guard functions determine when and if enabled transitions can occur. Priority and probability functions, for example, resolve conflicts between concurrently enabled transitions. It is possible that the occurrence of a transition disables another one by removing some of the tokens it needed. In that case a priority or probability decides which one can occur. Transition delays are another example of guard functions that control the maximum occurrence frequency. After an occurrence, a transition remains disabled for the duration of the delay. Generally, a guard function may be any type of Boolean function.

2.2.2 Inhibitor and Test Arcs

Test and inhibitor arcs [9] enable or disable transitions according to the contents of the places they are connected to. The difference between these and normal arcs is that they do not consume or produce any tokens. Although these types of arcs are very different from the traditional ones, there exist reduction rules that produce equivalent Petri Nets without inhibitor arcs.

2.2.3 Colored Petri Nets

Colored Petri Nets [10] are the foundation of higher-level nets, such as object-based or object-oriented Petri Nets. Colors refer to typed tokens that result from folding places or transitions of a simple Petri Net into one. This reduces the size of a net and provides more expressive modelling constructs. Sufficiently large color sets can produce arbitrarily complex token types and arc weight functions.

2.2.4 Petri Nets with time

There are several ways to incorporate time into a Petri Net. For example, by associating delays with transitions or places. In a *timed* Petri Net [2], for example, delays are associated with transitions. When a timed transition occurs it removes the tokens from the incoming places and reserves capacity on the outgoing places. Only after the time interval has expired are the tokens committed to the outgoing places. Similarly, in a *time* Petri Net delays are associated with places. After a token enters a place it remains hidden from outgoing arcs for the duration of the delay.

2.2.5 Hierarchical Petri Nets

Hierarchical Petri Nets introduce nested modules to the language. A module, or *page*, is a Petri Net that publishes a set of places or transitions as an interface. Constructs outside the page can then connect arcs to the interface. Depending on the type of constructs that are published, a page may behave like a place or like a transition. Place refinement, or transition refinement [7] are specialized forms of hierarchical Petri Nets. In the case of place refinement, for example, a page only has an interface of places and therefore behaves like one.

2.2.6 Higher Order Petri Nets

Higher order Petri Nets allow entire pages to be tokens. Connecting to a page carrying token while it resides on a place, dynamically changes the structure of the Petri Net. With dynamic place refinement [8] tokens carry the content that refines a place. Specialized port places provide an interface equivalent to that of a page, except that connections to the interface are only instantiated when a page carrying token enters the port. The connections remain in place while the tokens reside on the port.

3 Modelling Constructs

This section introduces how the Petri Nets and scene trees are integrated into a modelling language. The essence of the approach is that nodes from the scene tree become tokens in the Petri Net and that animation functions annotate places and transitions. The animation functions model continuous processes and the strucutre of the Petri Nets model descrete logic.

3.1 Continous Control

The continous modeling constructs includes changing the structure of the scene tree as well as altering the content of its nodes. Open Inventor engines are functions that change the node content, and operations like insertions and deletions are the functions that change the tree structure. The default set of engines includes interpolations of translations, rotations and floating point values. An engine attaches its output values to the fields of a scene tree node and continuously changes the appearance or location of objects. Engines can also control rendering conditions like lighting, shading or levels of detail.

3.2 Descrete Control

The discrete modeling constructs include a number of Petri Net extensions. Applications of Petri Nets often combine several extensions to create formalisms that are best suited for a particular domain. The following section gives an overview of the the Petri Net constructs chosen for this application. The examples are illustrated with images from the RealFusion development environment.

3.2.1 Tokens

The token color space is divided into two classes, each with a fixed number of colors. The *resource token* class consists of scene tree nodes and the *control token* class consists of integers. The integer tokens are typeless and they are used to describe buffers or counters. The resource tokens contain computer graphic constructs like transformation matrices, material tables and vertex arrays to which animation functions are applied by an executing Petri Net. Each resource token contains an additional bit vectors to hold domain specific state information. Arc weights, place capacities and markings are always a sub set of these two classes of tokens.

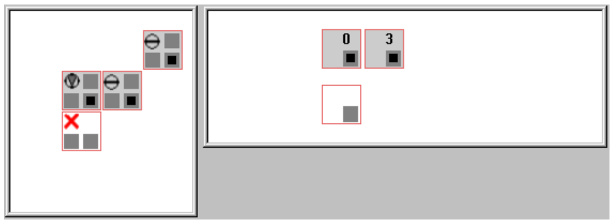


Figure 2: Resource Tokens (left) and Control Tokens(right)

Figure 2 shows the graphical representation of the two color spaces. Each single square including a set of indicators represents one token and its attributes. Common to all resource colors is an indication of the node type that occupies that color. Icons commonly used with Open Inventor indicate the different types. The development tool propagates that type information through the Petri Net and therefore always indicates the proper token types. Graphical widgets like those in Figure 2 display the contents of transitions, places and arcs. However, different attributes are shown for each construct.

3.2.2 Places

Places maintain a token queue and animation engine for each resource color. A time interval is associated with each position in the queue and when a token has reached the end of an interval it becomes available for outgoing arcs. If it is not removed, then it continues to the next location in the queue. The animation engine attaches to the fields of a token when it enters the place, and disconnects when a token exists the place.

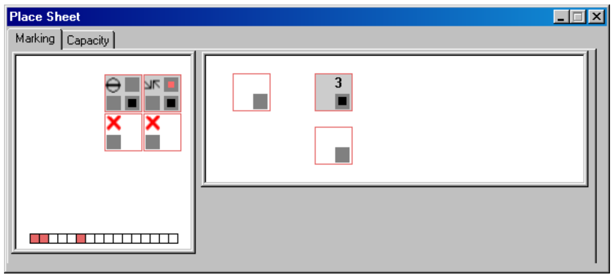


Figure 3: The content view of a Place

Figure 3 is an example of a place marking view. This place has a capacity for four types of resource tokens and three types of control tokens. A highlighted background indicates that the colors are currently marked and that tokens are residing on that place. The types of the tokens bind two of the resource colors to a transformation and separator node. The remaining resource colors are still un-typed, because there exists no token in the Petri Net that can reach this place on those colors. One marking is additionally annotated with a bit vector that will be part of arc expressions.

3.2.3 Arcs

The Arc weight is a subset of the color space defined for the place it is connected to. The weight of an arc leaving a place describes the marking necessary to enable the arc, and the weight of an arc entering a place describes the capacity required to enable the arc. The total weight is the combination of different weight functions for each color. Only if the weight of all specified colors is satisfied can the arc becomes enabled and cause a transition to occur.

An arc only deposits or removes single resource tokens, and the weight function for this type is either a Boolean true or false. An additional bit vector may also be part of the expression. Arcs that remove tokens compare the bit vector with that of a token and arcs that deposit tokens manipulate the bit vector. The arc weight also includes an index that refers to a position in the token queue of the place it connects to. An arc removes and deposits tokens at the location indicated by the index.

The weight function for control tokens is a number and an optional operator. If an operator is specified then the weight for this color is a test expression. A test weigh compares the marking or capacity of a place to the value of the arc weight. During an occurrence no tokens are moved when the weight function is a test.

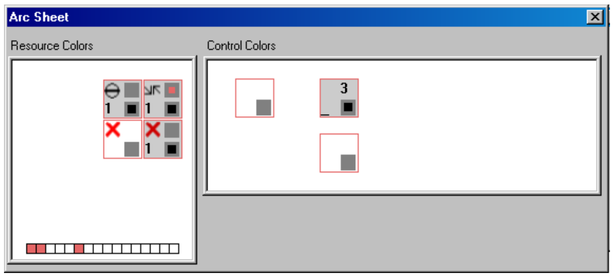
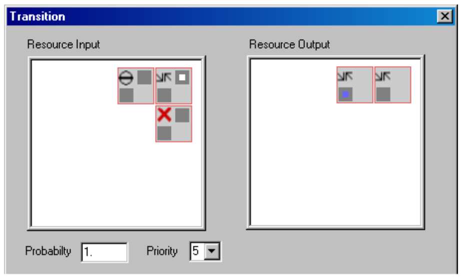


Figure 4: The content view of an Arc

Figure 4 shows the view of an arc originating from the place of Figure 3. The color types that compose the arc weight are also highlighted. This arc is not enabled because the place does not contain the necessary un-typed resource token. However, the remaining colors of the weight are satisfied. In addition, the view also shows how to incorporate the bit vector into the expression.

3.2.4 Transitions

Transition occurrences are instantaneous and guarded by a probability and a priority. When a transition occurs it applies instantaneous functions that change the structure of the scene tree to the sets of resource tokens it consumes and produces. This includes creating or deleting entire nodes or subtrees. Every resource token a transition produces must be initialized with a node of the proper type. This node is either obtained from one of the set of incoming tokens or from a function that creates a new node.



Probabilty 1.

Priority 5

Figure 5: The content view of a Transition

Figure 5 shows the view of the transition connected to the arc of Figure 4. The view is used to specify the mapping between the incoming and outgoing colors. In this example, the transition produces two outgoing tokens that are both mapped to an incoming transformation token. The new tokens are going to be copies of the incoming one. The other two incoming tokens do not propagate any further because they are not mapped to any of the outputs. There is no need to map the control tokens, because they are type-less. The transition view does therefore not include control tokens.

3.2.5 Pages

A page is a single Petri Net that always attaches to a node in the scene tree. The initial marking of resource tokens in the page originates from the sub-tree rooted at that node. A path starting at the root node identifies the location of each resource token in the marking. When a page is first instantiated, it marks the Petri Net with the resource tokens it extracts from its scene tree.

The interface to a page is specified in terms of links and references that are complementary building blocks to create dynamic connections. A link is a virtual place and a reference is a pointer to a real place. The interface constructs are embedded in the page where they can connect to places and transitions.

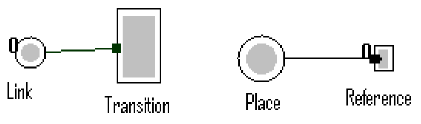


Figure 6: A Petri Net page with two interface constructs

Figure 6 shows a link connected to a transition and a reference connected to a place. The links always have zero capacity and can therefore not satisfy arcs that have a non-zero weight. As a result, transitions that are connected to links remain disabled and can not occur.

3.2.6 Ports

A port is a place that can create dynamic connections to tokens that carry pages. Part of a port is a set of user defined references and links that can connect with the places or transitions in the same page as the port. When a page carrying token enters a port the links and reference of the token are matched with the references and links of the port. After that, the arcs connected to the links become attached to the places pointed to by the references. This dynamically connects a page to the constructs of the Petri Net in which it resides as a token. As long as a page marks a port the connections to it remain.

Figure 7 is an example of a Port with two references and two links. A place is connected to reference 0 and a transition is connected to link 4. If the page of Figure 6 was to enter this port, then the place in Figure 7 would become connected to the transition in Figure 6.

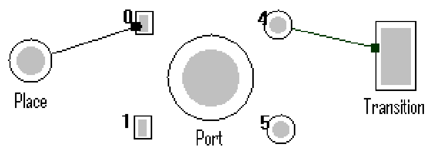


Figure 7: A Port with four interface constructs

4 Simulation

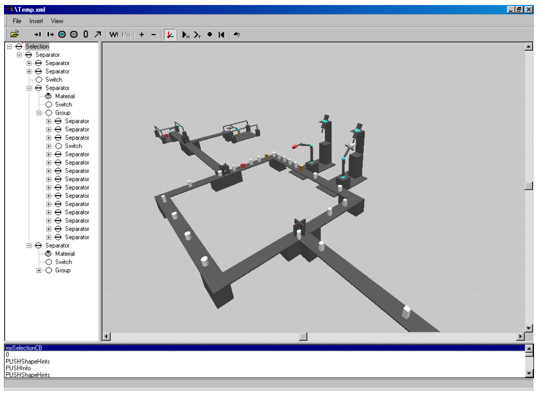
The RealFusion development environment supports all of the introduced language constructs. It integrates a Petri Net modeler and interpreter with the Open Inventor graphics library into one application. The close integration of the different components supports interactive and iterative modelling.

The process of building a simulation starts with a Scene Tree that resembles the appearance and structure of the system under development. Any general-purpose 3D modeler capable of producing Open Inventor files is suitable to build a scene. It is important that the tree closely resembles the structural dependencies of the system. This ensures that the Petri Net will be created in the proper context, and therefore closely resembles the processes that are occurring in the system.

Given an appropriate Scene Tree, the next step is to identify objects by attaching pages to selected nodes in the scene tree. For each page a process model and interface is created. The model includes the graph structure of the net, the annotation of places, transitions and arcs with functions and expressions, and an initial marking of tokens.

4.1 Interface

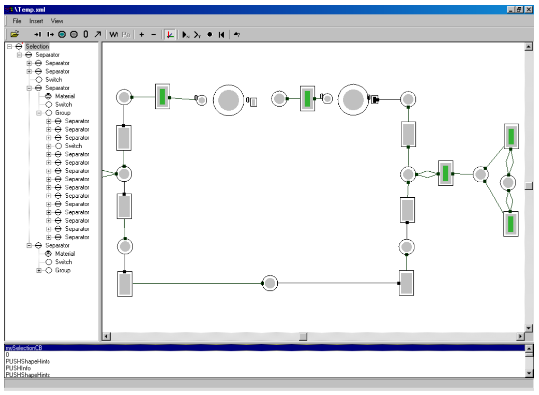
The user interface consists of a tree view and display window. The tree view shows the structure of the scene tree and the display window toggles between the rendered Scene Tree (Figure 8) and the Petri Net (Figure 9). During interpretation, the display window either shows the animated scene or the state changes of the Petri Net. The Petri Net view shows the enabling and disabling of arcs and occurrence of transitions as they execute.



```
mySelectionCB
0
PUSHShapeHints
PUSHInfo
PUSHShapeHints
```

Figure 8: Graphic view for a manufactuing system

The Petri Net View shown in Figure 9 displays the contents of a selected page. The page of this example contains several ports that are marked with the pages that model the different work cells of the manufacturing system show in Figure 8.



```
mySelectionCB
0
PUSHShapeHints
PUSHInfo
PUSHShapeHints
```

Figure 9: Process view for a manufacturing system

Places and transitions have additional views to edit their content. In particular, the views necessary to select the engine types and their I/O mappings to the fields of a scene tree node.

4.2 Interpreter

The semantics of the Petri Net formalism describes the rules of how to interpret and execute the language constructs. An interpreter implements these rules in a procedural programming language and executes a concurrent Petri Net on a sequential computer. The interpreter maintains the graph structure of the net, controls the occurrence of transitions, the marking of places and the movement of tokens. Although each page gives the impression of being a self contained Petri Net, all of their transitions, places and arcs are maintained in a flat net structure.

The timing of the Petri Net is linked to the frame rate of the Scene Tree renderer. The interpreter and renderer advance in lock step to avoid temporal antialiasing, where changes to the scene happen faster than the frame rate. Between rendering each frame, the interpreter advances one time cycle. During that cycle it allows transitions to occur and places to update the functions they compute. Internally the interpreter maintains a priority queue of transitions that are ready to occur, and a list of places that currently compute any functions. To avoid deadlocks each transition is only allowed once in the ready queue during each time cycle.

5 Results

The first model produced with this formalism is shown in Figure 8. It is a manufacturing facility that contains robotic arms, conveyor belts and packing stations. The facility consists of four cells that are responsible for coloring and packaging two different types of resources. Junctions connecting the conveyor belts sort the resources according to their state of manufacturing. The most notable results are the ease with which to model the parallel control processes that guide the activities in this system. The Petri Net naturally computes concurrently and processes synchronize automatically with the presence or absence of tokens. Also, the entire process model is symbolic and the structure of the Petri Net reflects the physical layout of the system. Following are initial observations of how the individual language constructs apply to describe concurrent, sequential and dynamic tasks.

5.1 Sequential tasks

A place can apply the same function to a set of different tokens sequentially. Because tokens enter the place at different times they are at different stages or time indexes of the function. An animation function for such a place, for example, is the path of a conveyor belt. The longer a tokens has been on the place, the further along the path it will be. The queuing capacity determines the maximum number of elements allowed on the conveyor belt. Arcs to and from the place are enabled when there is room at the front of the belt or, when a resource has reached the end.

5.2 Concurrent tasks

The different colors of a place can apply different functions to a set a tokens concurrently. Objects that move in synchronization reside next to one another on the same place and they become part of the same transition occurrences. By placing all of the moveable parts of a robotic arm onto a place, for example, it can position the arm be rotating all of the joints at once. The animation functions in this case are the different rotations about the joints.

5.3 Dynamic tasks

The port places represent a different context for a machine or cell. The objects inside a page are still changing state, or operating, while the page rests on a port. The connections between the port and the page influence the pages' internal behavior and allow the exchange of resources. Moving pages between ports is similar to moving a machine from one context to another. Depending on where it resides, different activities can be activate and different resources manipulated. An example of utilizing a port is in the case of a flexible production system where a robotic arm must pick up any one of a number of different tools. Separate pages model the behavior of the tools and that of the robotic arm. The page of the robotic arm, however, contains a port place. Depending on what tool it currently uses, a different token resides on the port.

6 Related Work

Many vendors and research groups that develop system design tools, are taking steps towards integrating three-dimensional computer graphics. The tools and vendors include, but are not limited to, the Simulink environment from Mathworks, 20 Sim from Controllab, and Quest/Delmia from Deneb.

Simulink is a graphic modelling environment that allows interactive assembly, simulation and analyzing of complex system models. The building blocks are predefined domain specific block diagrams. The language builds on the MATLAB toolkit and provides an extensive library of functionality. Recent additions to the language include virtual reality sinks and sources that allow a model to communicate with the MATLAB Virtual Reality Toolbox. The toolbox is a distributed VRML-based virtual environment and the sinks and sources trigger scripts that execute inside the virtual environment.

20 Sim [11] is a graphic modeling tool for hybrid systems based on bond graphs. The tool allows interactive development, simulation and analysis of a system model. 20 Sim also allows the mapping of process variables into a 3D scene.

Quest/Delmia from Deneb is touted the most advanced manufacturing simulation software available today. Quest/Delmia produces a range of products to support the life and development cycle of a manufacturing system, including three-dimensional visualization. Deneb has taken an object-oriented approach towards simulation where objects are defined using GSL (Graphic Simulation Language) [12], a procedural programming language. The GSL language constructs are integrated with other aspects of the manufacturing simulation such as production databases.

With respect to Petri Nets, integration into computer graphics has been sparse so far. Although Petri Nets have been used to model mechatronic systems [1], the extent to which they incorporate computer graphics is limited to animating the Petri Net itself, or graphically plotting the variables inside the net.

However, there has been some research to explore the use of Petri Nets for animation control [13]. The author modelled the processes that control the articulation of human figures by using a Petri Net that invokes human character animation sequences.

7 Conclusion

This work presents a new application area for Petri Nets and also a novel technique to define and execute three-dimensional models of mechatronic manufacturing systems. To the best knowledge of the authors, Petri Nets have not yet been investigated in the manner presented here. During the course of creating a development environment for this formalism, new interpretation and user interface techniques with respect to Petri Nets have also been developed.

The formalism presented here leverages the advantages of a formal and symbolic modelling language with that of three-dimensional computer graphics to support the design of manufacturing systems. Most notably, this formalism supports concurrency, is type safe and has an intuitive user interface. Processes models are created in the context of the physical structure of the system, and they are therefore easy to comprehend and maintain. In conjunction with effective tool support, the design task becomes interactive and facilitates an iterative and evolutionary style of development. The language is also is based on a proven technique, and incorporates the constructs necessary to describe the hybrid nature of a manufacturing process.

8 Future Research

An important area of future research is to explore more advanced animation functions and to incorporate sensing techniques such as collision and proximity detection. The modelling language presented here can be extended by incorporating new types of tokens and animation functions. There are also techniques to model sensors by deducing spatial information from the Scene Tree. It remains to be determined how to incorporate sensors with a Petri Net.

Hybrid Petri Nets are another important area of research. They include special place and transition types that incorporate continuous variables other than time. They are important to compute the multivariable differential equations that are often necessary to model physical processes. Much research has been conducted with respect to Hybrid Petri Nets, and they are an important extension to the formalism introduced here.

Simulation-based control is based on the concept of developing and controlling a system using only simulation software. The advantages to this approach are that the design process directly produces the control software without additional development steps. The Petri Nets described here, at least partially, reflect the logic necessary to control the system under development. It remains to be investigated how to utilize these models as control software.

References

[1] P. Antsaklis and X. Koutsoukos. Hybrid systems: Review and recent progress. *Software Enabled Control, IEEE Press*, To appear 2002.

[2] M.D. Lemmon X.D. Koutsoukos, K.X. He and P.J. Antsaklis. Timed petri nets in hybrid systems: Stability and supervisory control. *Journal of Discrete Event Dynamic Systems: Theory and Applications*, 8(2):137–173, 1998.

[3] C. A. Petri. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Advances in Petri Nets 1986, Part I, Petri Nets: Central Models and Their Properties.*, volume 254 of *Lecture Notes in Computer Science*, pages 4–24. Springer-Verlag, 1987.

[4] E. Schnieder M. Chouikha. Modelling of continuous-discrete systems with hybrid petri nets. *Proc. of CESA'98: Computational Engineering in Systems Application*, 3:606–12, 4 1998.

[5] R. Parent. *Computer Animation, Algorithms and Techniques.* Morgan Kaufmann, 2001.

[6] C. A. Lakos. The object orientation of object petri nets. In *Workshop on Object Oriented Programming and Models of Concurrency*, 1995.

[7] W. M. Zuberek and I. Bluemke. Hierarchies of place/transitions refinements in petri nets. *Conference on Emerging on Technologies and Factory Automation*, pages 355–360, 1996.

[8] J. W. Janneck and R. Esser. Higher-order petri net modeling—techniques and applications. *Workshop on Software Engineering and Formal Methods*, 2002.

[9] C. A. Lakos and S. Christensen. A general systematic approach to arc extensions for coloured petri nets. *Application and Theory of Petri Nets*, pages 338–357, 1994.

[10] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use.* Springer Verlag, 1997.

[11] J. van Amerongen. Modelling, simulation and controller design for mechatronic systems with 20-sim 3.0. *1st IFAC conference on Mechatronic Systems*, 9 2000.

[12] P. M. Griffin L. F. McGinnis D. A. Bodner, M. Damrau and A. McLaughlin. Virtual prototyping of electronic assembly systems. *Proceedings of the 1998 Industrial Engineering Research Conference*, 1998.

[13] A. Raposo A. de Lima Bicho and L. P. Magalhes. Control of articulated figure animation using petri nets. *XIV Brazilian Symposium on Computer Graphics and Image Processing*, 2001.