Geometry-driven Petri Nets and a method for modelling Mechatronic Control Systems

by

Jochen Stier
B.Sc., University of Victoria, 1995
M.Sc., University of Victoria, 1998

A Dissertation Submitted in Partial Fulfillment
of the Requirements for the Degree of

Doctor of Philosophy

in the Department of Computer Science.

Geometry-driven Petri Nets and a method for modelling
Mechatronic Control Systems

by

Jochen Stier
B.Sc., University of Victoria, 1995
M.Sc., University of Victoria, 1998

Supervisory Committee

Dr. Jens H. Jahnke, Department of Computer Science, University of Victoria

Supervisor

Dr. Hausi A. Müller, Department of Computer Science, University of Victoria

Co-Supervisor

Dr. Kin F. Li, Department of Computer Engineering, University of Victoria

Departmental Member

Dr. Colin Bradley, Department of Mechanical Engineering, University of Victoria

Outside Member

Dr. Albert Zündorf, Department of Computer Science, University of Kassel

Additional Member

Supervisory Committee

Dr. Jens H. Jahnke, Department of Computer Science, University of Victoria
Supervisor

Dr. Hausi A. Müller, Department of Computer Science, University of Victoria
Co-Supervisor

Dr. Kin F. Li, Department of Computer Engineering, University of Victoria
Departmental Member

Dr. Colin Bradley, Department of Mechanical Engineering, University of Victoria
Outside Member

Dr. Albert Zündorf, Department of Computer Science, University of Kassel
Additional Member

Abstract

The development process of mechatronic control systems often relies on physical prototypes to test the interactions between the control software and mechanical components. However, the logistics of synchronizing a concurrent development process and the risks of integrating only partially completed sub-systems often limits effective prototyping. The consequent lack of feedback can lead to overly complex and unreliable systems which may have to undergo expensive re-designs.

The interactions between mechanical systems and control software can also be recreated artificially by combining a hybrid modelling language with computer graphics technology. A dynamic 3D environment can generate sensor telemetry for input to a control system, which in turn alters the state of the environment through virtual actuators. This kind of simulation allows engineers to explore a larger design space early during the development process without committing significant resources to physical prototypes.

This dissertation introduces a method for simulating mechatronic systems using Petri Nets and Scene Trees. The following chapters formally define the modelling

language and illustrate the software architecture and user interface of a novel simulation development environment. The research is validated through qualitative reasoning and by demonstrating a simulation that detects design flaws in a mechatronic system which may have otherwise lead to expensive redesigns in the physical system.

Contents

Title i

Supervisory Committee ii

Abstract iii

Table of Contents v

List of Figures ix

List of Tables xi

Acknowledgements xii

Dedication xiv

1 Introduction 1

1.1 The Problem 1

1.2 The Solution 2

1.3 Overview 3

1.4 Evaluation 4

1.5 Contributions 5

1.6 Dissertation Outline 7

2 The Virtual Environment **8**

2.1 Scene Trees 9

2.2 Rendering 11

2.2.1 Natural Terrains 13

2.2.2 Architectural Structures 14

2.3 Animation 14

2.3.1 Rigid Body Dynamics 16

2.4 Manipulators 18

2.5 Sensors 20

2.6 Open Inventor 22

2.7 Summary 22

3 Hybrid Modelling Languages **24**

3.1 Language Histories and Evolution 24

3.2 Petri Nets 26

3.2.1 Behavioral Properties 29

3.2.2 Structural Properties 30

3.3 Colored Petri Nets 32

3.4 Summary 34

4 Geometry-driven Petri Nets **35**

4.1 Formal Definition 36

4.2 Runtime Kernel 41

4.3 Performance 45

4.3.1 Virtual Reality 45

4.3.2 Petri Net 46

4.3.3 Geometry-driven Petri Nets 47

4.4 Summary 48

5 User Interface **49**

5.1 Application Frame 50

5.2 Content Views 52

5.3 Discussion 56

5.3.1 Abstract Gradient 58

5.3.2 Closeness of Mapping 59

5.3.3 Consistency 60

5.3.4 Error-proneness 60

5.3.5 Hidden Dependencies 61

5.3.6 Progressive Evaluation 63

5.3.7 Role Expressiveness 64

5.3.8 Secondary Notion and Escape from Formalism 65

5.3.9 Viscosity 65

5.3.10 Visibility and Juxtaposition 67

5.4 Summary 69

6 Software Architecture **70**

6.1 Module View 70

6.2 Data View 73

6.3 Process View 76

6.4 Extensibility 79

6.5 Summary 83

7 Case Studies **84**

7.1 Robotic Arm 86

7.2 Painting Station 88

7.3 Production Line 91
7.4 Summary 94

8 Related Work **95**
8.1 CPN/Tools 95
8.2 Delmia 98
8.3 Webots 99
8.4 Gazebo 100
8.5 Gamebots 101
8.6 Summary 101

9 Evaluation **103**
9.1 Background 104
9.1.1 How is Software Evaluated? 104
9.2 Discussion 106
9.2.1 Petri Nets 107
9.2.2 Mechatronic System Simulation 108
9.3 Summary 109

10 Future Research **110**
10.1 Character Animation 110
10.2 Rigid Body Dynamics 112
10.3 Plant growth 113
10.4 Communication Infrastructure 114
10.5 Hybrid Modelling Language 114
10.6 Autonomic Computing 115
10.7 Control Synthesis 117
10.8 Molecular Manufacturing 118

10.9 Summary 119

11 Conclusions **120**

11.1 Contributions 122

11.2 Final Thoughts 123

A Class Listing **135**

B Kernel Source Code **141**

List of Figures

1.1 Control feedback loop in Mechatronic Systems 2

2.1 A structured and visual representation of a Scene Tree 9
2.2 3D manipulators of $3D^{os}$ 19
2.3 3D collision sensors of $3D^{os}$ 21

3.1 Three steps in a simple Petri Net 28
3.2 Common Petri Net patterns 31

4.1 Simulation architecture 36
4.2 Page, Reference and Link 40

5.1 $3D^{os}$ screenshot 51
5.2 Contents View 53
5.3 Sorting 3D model 54
5.4 Sorting python script 55

6.1 Module decomposition 72
6.2 Module interface visibility 73
6.3 $3D^{os}$ Data view 74
6.4 Fields 75
6.5 Event call sequence 77

6.6 Paint call sequence 78
6.7 Load call sequence 79
6.8 ScnText definition 80
6.9 ScnText declaration 82

7.1 3D model of material handling system 85
7.2 3D model and Petri Net of robotic arm 87
7.3 Color set of transition *deliver* 87
7.4 3D model of painting station 89
7.5 Petri Net of painting station 90
7.6 Petri Net of production line 91
7.7 Images of production line 92

8.1 Design/CPN screenshot 96

11.1 The Spiral Model of Software Development 121

List of Tables

6.1 $3D^{os}$ modules 71
6.2 3rd party modules 71

7.1 Selected occurrence inscriptions 88
7.2 Factory model cycles times 93

8.1 Design/CPN-$3D^{os}$ feature matrix 97
8.2 Tools feature matrix 102

A.1 Classes of module Geo 135
A.2 Classes of module Scn 136
A.3 Classes of module Scn 137
A.4 Classes of module Man 137
A.5 Classes of module Cmd 138
A.6 Classes of module Win 138
A.7 Classes of module Rtn 139
A.8 Classes of module App 140

B.1 Kernel File Listing 141

Acknowledgements

I would like to thank my supervisors Drs. Jens Jahnke and Hausi Müller for their patience, support and advice during the past four years. I would also like to thank those members of the Computer Science Department, who have been mentors during my academic career. Particularly, Mantis Cheng, John Ellis, Frank Ruskey and Valery King.

I would also like to thank my friends, Todd, Tanya, Jason and Duncan, who have listened to me talk about Petri Nets and Computer Graphics without end. I hope it was interesting at times.

And of course there is Maryam! So much more is dedicated to her than this dissertation...

To

Oon Maymoone Zeeba

Chapter 1

Introduction

"Experiments and not conjecture are certainly the best basis to determine the performance of a system and simulation is the preferred method of experimentation." [1]

The development process of mechatronic systems traditionally relies on physical prototypes for testing. However, in many cases prototyping is costly and time consuming, and a lack of feedback often leads to insufficient validation before a final solution is produced. Yet, correcting errors at a later stage of development can be orders of magnitude higher than if they had been discovered earlier. An opportunity to analyze a model for flaws should therefore always be taken, especially when physical prototypes are unavailable [2]. This dissertation introduces a modelling methodology that is exclusively based on software. The necessary technologies have now evolved to the point where a simulation can project design decisions forward in time and help to evaluate different solutions effectively.

1.1 The Problem

Most of the current industrial simulation tools are designed for repetitive manufacturing system in which devices and resources follow predetermined paths through space [3, 4]. Research efforts now focus on methods for modelling autonomic [5] or adaptive [6] mechatronic systems, which are characterized by sensor-driven logic without centralized control. Sensor-driven systems continuously sample the environment and adapt their behavior according to local conditions and priorities. Sim-

ulating such a system requires a tool that models the natural artifacts which lead to sensor telemetry, and that executes hybrid control logic at the same time. The goal is to artificially recreate the feedback loop between a control system and the environment which physical prototypes implicitly close through natural processes.

Figure 1.1 shows the components of a mechatronic system. Software controls and monitors the mechanical component through electro-mechanical actuators and sensors. Sensor telemetry causes state changes in the logic, which in turn alters the state of the mechanical component via actuators. In order to test the control system, it is necessary to maintain a correct cause-and-effect relationship between the actuators and the values recorded by the sensors. The effect of an actuator leads to expecting a certain sensor input at a later point in time.

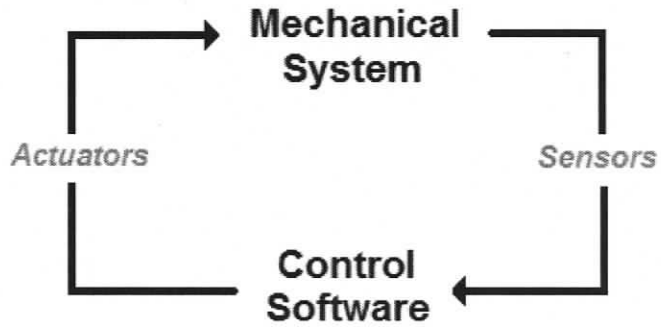

Figure 1.1: Control feedback loop in Mechatronic Systems

1.2 The Solution

A solution for simulating a mechatronic system is to combine a hybrid modelling language with a virtual reality component. The virtual environment maintains a detailed 3D descriptions of the mechanical components and computes the necessary

physical processes. The hybrid modelling language describes the control software using a combination of a discrete-event and procedural programming language [7]. The low level of control consists of a layer of continuous functions that produce and interpret electronic signals. The high level consists of a discrete-event system that switches between the low level functions in order to execute a desired work flow. A hybrid modelling language combines both types of control structures into a single formalism capable of modelling complex distributed processing systems.

Integrating a hybrid modelling language with a virtual environment through artificial sensors and actuators creates a tool to simulate mechatronic systems. The state of the control logic depends entirely on the state of the virtual environment and vice versa. This concept is not new, and several other research groups are currently investigating how to construct useful simulation tools [8, 9, 10]. This dissertation investigates how to use Petri Nets as a hybrid modelling language and how to integrate the language constructs with a virtual reality system.

1.3 Overview

The focus of this dissertation is to present the underlying research form the viewpoint of Petri Nets. Among the various discrete-event formalisms proposed so far, Petri Nets have arguably proven as the most effective. The language has a formal mathematical basis and provides good cognitive support when developing large hierarchical processing systems. Geometry-driven Petri Nets are a new form of Petri Net especially designed to control the processes in a virtual environment. The Petri Net graph expresses communication and concurrency, and Python source code expresses the continuous functions that control actuators and interpret sensors. As the Petri Net executes, the Python scripts alter the state of the environment and close the feedback loop which drives the simulation.

Much of the research was to construct a development environment that incorporates all of the computational concepts in Geometry-driven Petri Nets into a single desktop application. A virtual reality software system was designed that integrates a number of computer graphics related algorithms and supports a variety of sensors. A Petri Net editor and runtime kernel for hierarchical Colored Petri Nets was also developed and integrated with the Python interpreter. Much attention was paid to designing an effective user interface for managing the large amounts of heterogeneous information that comprises a virtual environment and a hybrid control system.

This dissertation introduces Geometry-driven Petri Nets and the accompanying software development environment called $3D^{OperatingSystem}$, or $3D^{OS}$ for short. The Petri Net is formally defined and the software architecture of the environment is documented.

1.4 Evaluation

The hypothesis of this dissertation is that combining a virtual reality component with a hybrid modelling language based on Petri Nets is an effective tool for simulating the control of mechatronic systems. However, this assumption already seems to be valid, because simple reasoning may lead to the conclusion that a certain combination of methods will lead to the desired result. The focus of this research was therefore not to discover a model that validates the hypothesis, but to engineer one that demonstrates it. In this case, being successful means that the simulation closes the feedback loop between actuators and sensors. This approach to software engineering is not new, and a number of scientists are advocating it as a valid method for conducting and evaluating research in this field [11].

The research that led to this dissertation has resulted in an unprecedented integration of Petri Nets and virtual reality technology. The $3D^{OS}$ development en-

vironment makes it possible to bind the constructs of a 3D model to the constructs of Petri Nets, and thereby model mechatronic systems in the absence of physical prototypes. To the best knowledge of the author, no one else has yet attempted the same approach. Nothing equivalent therefore exists against which to perform a meaningful quantitative evaluation in terms of a tool comparison. However, simulation tools are probably better evaluated by measuring how closely they reflect the artifacts that are being simulated. A metric for this purpose is proposed at the end of this dissertation.

$3D^{os}$ has been evaluated by simulating different mechatronic systems, including autonomous vehicles navigating a natural landscape and a material handling system that displays complex work flows. Not all of the models are introduced in this dissertation, but they are included with the software distribution. Part of the evaluation presented here illustrates how more reliable Petri Net models can be constructed in conjunction with a virtual reality component, and how a subsequent cycle time analysis can produce more accurate statements about the throughput of a system. Geometry-driven Petri Nets and $3D^{os}$ are also characterized by a number of other qualities, which are mentioned throughout this dissertation; especially in the *Evaluation*, *Related Work* and *User Interface* chapters.

1.5 Contributions

Foremost, this dissertation contributes a new simulation method for designing mechatronic control systems using a new hybrid modelling language. Some components of this method are formally defined and others are illustrated. During the five years of research and development leading to $3D^{os}$, many problems with respect to presenting and integrating the different computational artifacts had to be resolved. Most of the approaches and solutions are outlined in this dissertation.

The second contribution is the $3D^{os}$ development environment including the user interface widgets and interaction models which were especially designed to construct simulations in terms of Geometry-driven Petri Nets. $3D^{os}$ is one of only a few Petri Net modelling tools in existence today, and the only one that includes a virtual reality component. It is therefore an important contribution towards adopting the Petri Net language in new areas of engineering design. This dissertation only covers a small range of the potential application domains.

The third contribution is a methodology for applying Geometry-driven Petri Nets to model mechatronic systems. A case study demonstrates how to construct a material handling system using the interface of the development environment. The model illustrates how to leverage the wealth of analytical methods and the cognitive support inherent to the Petri Net formalism. Several design patterns, which are likely inherent to this language, are also introduced. Many other models, even beyond the area of mechatronics, are included in the software distribution of $3D^{os}$.

Finally, this dissertation contributes a new step in the evolution of the Petri Net language in general. Traditionally, the formalism does not incorporate geometric descriptions of the systems being modelled. Variables representing sensors and actuators are therefore approximated or randomized in the models which have been constructed so far. Geometry-driven Petri Nets instead bind many of the parameters to a virtual environment containing a dynamic representation of the system and its surroundings. As a result, more realistic process models can be synthesized, because a Petri Net can be validated by cross-referencing its state with that of the virtual environment.

1.6 Dissertation Outline

Chapter 2 introduces the virtual reality system of $3D^{os}$ including selected algorithms and 3D user interface widgets. Chapter 3 explains the nature of hybrid systems and introduces the Petri Net formalism. Chapter 4 formalizes Geometry-driven Petri Nets and provides a running time performance analysis. Chapter 5 and Chapter 6 introduce the $3D^{os}$ development environment, including the software architecture and user interface. Chapter 7 discusses relevant related work with respect to Petri Nets and mechatronics simulation tools. Chapter 8 shows how to apply Geometry-driven Petri Nets, by outlining a model of a material handling system. Chapter 9 presents a qualitative evaluation and discusses a metric for quantifying the performance of simulation tools in general. Finally, Chapters 10 and 11 discuss future research, present the contributions of this dissertation and draw conclusions.

Chapter 2

The Virtual Environment

"Artificiality connotes perceptual similarity but essential difference, resemblance from without rather than within." [12]

In the context of this dissertation, the term *virtual reality* refers to software that models the dynamic behavior of mechanical systems and the environments they operate in. Its purpose is to reflect the state of actuators and generate sensor telemetry for input to a control system. The capability of this component and the types of telemetry it produces defines the types of systems that can be modelled with Geometry-driven Petri Nets. Improving the virtual reality component is therefore an ongoing project which will not be completed for some time to come.

A virtual reality system combines many different algorithms and data structures designed to represent physical objects and processes into a single runtime environment. Significant development effort is therefore required to design and implement the software and much care has to be taken that the system executes in real time and with limited resources. In addition, comprehensive user interface widgets are required to assemble and navigate complex 3D environments. This chapter introduces the algorithms, data structures and user interface widgets that comprise the virtual reality component of $3D^{os}$.

2.1 Scene Trees

A Scene tree is a data structure that organizes the information of computer graphics software system. A hierarchy of differently typed nodes relates artifacts such as the geometric description of surfaces, colors and material properties, light sources and viewpoints. The rendering process, which generates a single image from the Scene Tree, *applies* each node to the current drawing state in the order that they are encountered during a tree traversal. Both, the contents of the nodes as well as the structure of the tree therefore determine the visual properties of a scene. Animations are produced by changing the structure or the content between rendering steps.

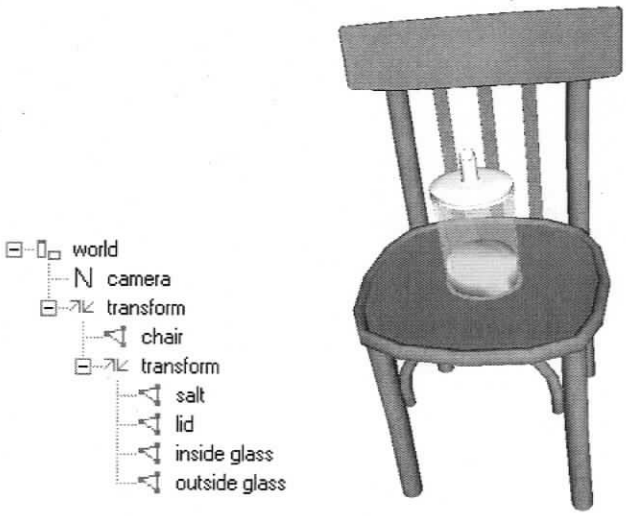

Figure 2.1: A structured and visual representation of a Scene Tree

Figure 2.1 shows a Scene Tree of a sugar dispenser resting on a chair. The root node defines global parameters of the environment such as gravity or rendering state. The first child of the world is the camera that defines the location and viewing frustum through which to project the scene. The next child is a transform

node containing the sub-tree of the chair. A transform consists of a homogeneous 3D transformation matrix which defines an orientation and a position in space. The children of the transform consist of a triangle mesh defining the shape of the chair, and another transform node containing the sugar dispenser. The dispenser is further divided into four triangle meshes. Two are for the inner and outer surfaces of the glass, and one each for the lid and the shape of the sugar inside. Different meshes are required when different surface properties are used to define a shape. Most implementations of Scene Trees as well as the underlying OpenGL graphics language only allow one material property per mesh. The inner and outer layers of glass have different levels of transparency, and the lid and sugar have different diffuse, specular and ambient lighting properties.

The example above demonstrates how a Scene Tree reflects structural dependencies among objects. The sugar dispenser resting on the chair is represented by a node which is a descendent of the chair. During a rendering step all of the vertices in a mesh are transformed by all of the transformations on the path from the root to the mesh. By changing the values of the transformation node above the chair, both the chair and the dispenser move. However, only the dispenser moves when the transformation node above the salt shaker is changed.

The tree data structure maintains a recursive hierarchical decomposition which supports efficient divide and conquer algorithms [13]. One example is the rendering process, which only considers the volume of space visible from the current viewpoint. Invisible artifacts can quickly be eliminated from further consideration by structuring the scene as a space sub-subdividing quad tree [14]. Other examples include selecting an object or computing sensor telemetry. By proving an interface for constructing a Scene Tree, users can optimize traversals specifically for each application.

Most 3D graphics tools apply some form of Scene Tree, and adopting this struc-

ture is an important step towards leveraging other modelling tools. Rendering any foreign data format generally requires parsing input files and constructing the corresponding $3D^{os}$ Scene Tree. So far, $3D^{os}$ supports the 3D Studio Max .3ds file format and the Virtual Reality Modelling Language (VRML). In the future, the SolidWorks format should also be supported, because it is one of the most popular for designing mechanical components. $3D^{os}$ would consequently integrate more effectively with existing development tools and process.

The following is a formal definition of a Scene Tree S, which is used for the remainder of this dissertation:

(i) A Scene Tree is a tuple $S = \langle V, E, T_f \rangle$.

(ii) V - set of nodes.

(iii) E - set of edges: $E \subset V \times V$ such that $\forall v \in V : |\{(u, v) \in E : u \in V\}| \leq 1$

(iv) T_f is a tree traversals.

The above definition also includes a set of tree traversal functions T_f, which will later also be referred to as *actions*. The traversal are designed to compute properties of the tree and to propagate editing changes to the individual nodes. Collision sensors and 3D editing functions, for example, are encapsulated by traversals.

2.2 Rendering

Rendering a 3D scene refers to the process that generates a single image from the Scene Tree for display on a computer monitor. The most common rendering algorithms are *depth-buffering* and *ray-tracing*, which focus on speed and realism, respectively.

Ray-tracing is based on casting rays from a viewpoint within the scene through a virtual computer screen. The algorithm first calculates which surface a ray intersects

and then combines the color properties of that surface with the results of further recursive ray casts originating from the intersection point. The surface properties of an intersection determine the angles and density at which new rays are cast. Ray tracing basically simulates the physical processes that describe the movement of light through space. This method has become so effective that it produces photo-realistic images.

Depth-buffering first sorts all of the surfaces in the scene with respect to their distance form the viewpoint, and then paints each onto the screen beginning with the most distant. Unlike ray-tracing, the color of a surface is not influenced by the location and color of other surfaces in the scene. Only light sources are taken into consideration when computing color gradients. This method alone can therefore not produce realistic shadows or reflections. Depth-buffering is analogous to a painter drawing a picture by starting with the background and finishing with the closest artifact in the foreground; often times painting over the previously laid background. Most graphics hardware natively supports depth-buffering through the OpenGL or DirectX graphics languages and is capable of painting tens of millions of triangles per second.

Researchers are already taking steps towards real-time ray-tracing, but so far it is still too inefficient to render non-trivial scenes in real time [15]. Depth-buffering was therefore chosen as the primary rendering method for $3D^{os}$. A real-time simulation provides immediate results and ensures a small delay between design and evaluation. $3D^{os}$ implicitly uses depth-buffering through the OpenGL 2.0 language.

Another important development with respect to rendering technology is point-based rendering [16]. The overhead of managing the connectivity in polygonal meshes has lead many researchers to question the future utility of polygons as the fundamental graphic primitive. Especially as the quest for more realism leads to algorithms that tessellate shapes to a level of detail at which a surface covers hardly

more than a single pixel. Much of the connectivity information is therefore irrelevant and rendering point sets directly has gained much attention in recent years. It is likely that dedicated rendering hardware for point sets will soon become available.

The following sections introduce the more complex geometric primitives supported by $3D^{os}$, and discuss the challenges of rendering them. The basic geometries including spheres, cylinders, cubes and triangles meshes are represented by different node types, and more complex structures are defined by sub-trees. In order to simulate realistic lighting conditions, surface normals and tangents are pre-computed for all geometries and then passed as parameters to OpenGL functions.

2.2.1 Natural Terrains

The challenge in rendering large terrains is to minimize the number of surfaces which have to be drawn without losing visual detail. The brute force approach is to triangulate a two-dimensional elevation grid and always paint all triangles regardless of surface features or distance to the viewpoint. However, high resolution elevation sets consist of millions of points that lead to a prohibitive number of triangles for real time applications. Also, rendering uniformly dense triangulations can lead to undesirable aliasing effects caused by mapping too many color values to a single pixel. The general solution is to eliminate areas which are invisible from the current viewpoint, and to compute the level of detail according to distance and surface features. Planer regions, for example, are represented using larger triangles.

Some tessellation methods calculate the level of detail dynamically at runtime using recursive subdivision or by leveraging the features of advanced graphics hardware [17] [18]. Other methods perform preprocessing steps to determine optimal tessellations for different levels of detail ahead of time [19][20]. $3D^{os}$ uses the Real-time Optimally Adapting Meshes (ROAM) algorithm introduced by Lindstrom et al. [21]. ROAM tessellates elevation data every time at each rendering step by recur-

sively subdividing areas which require a higher level of detail. The input parameters are only a two-dimensional grid of elevation points which can be modified during runtime without incurring a performance penalty due to repeating a preprocessing step.

$3D^{os}$ also supports a wide variety of elevation data formats using the *gdal* raster data management library [22]. Almost all of the elevation data sets published by the United States Geological Survey can be integrated into a simulation [23]. Many types of mechatronic systems are designed to operate in natural environments, and the data sets collected by remote sensing devices can create a realistic context for simulation.

2.2.2 Architectural Structures

$3D^{os}$ renders architectural structures using binary space partitioning (BSP). BSPs are a variation of Scene Trees which partitions large architectural structures for fast rendering and collision detection [24]. The tree structure is determined by an algorithm that takes into consideration which areas are visible from any other area. Given a vantage point inside a building, a search algorithm on the partition quickly determines all of the surfaces visible from that location. BSPs, however, do not lend themselves well to modelling dynamic structures which frequently change shape. BSPs are still at an experimental development stage in $3D^{os}$.

2.3 Animation

The traditional method for computer based animation is called *key-frame* animation, in which an animator specifies the parameters of an object, for example location and orientation, at two different points in time. An algorithm then approximates the missing values for each frame in between the keys. Depending on the interpolation technique, a smooth and continuous animation can be produced. However, key frame

animation is laborious and time consuming, especially when an animation has many parameters; as it is the case with natural phenomena. Also, key-frame animation is designed for static environments, such as the individual scenes of a motion picture. It is often insufficient for dynamic real-time environments.

Advances in computing power and software now enable *algorithmic* animation methods, which continuously compute mathematical models that describe a natural process leading to an animation. A classic example is rigid body dynamics, in which systems of linear equations implicitly enforce physical laws without any intervention by an animator. Other forms of algorithmic animation include neural networks to animate skeletal systems [25], L-Systems [26] to model plant growth, finite elements [27] to model fluid dynamics, and particle systems [28] to model collective intelligent behavior. At this time, only rigid body dynamics is supported by $3D^{os}$.

An example that illustrates the advantages of procedural animation is the problem of animating a human character walking along uneven terrain. At every step, the human body corrects its stance to compensate for the shifting center of gravity caused by the walking motion. Due to the uneven terrain, the range of motion within the joints of the skeleton differs at every step. Using key-frame animation, an animator has to define the path for each joint explicitly, taking into consideration the effect of gravity as well as the terrain. Only skilled and experienced practitioners can synthesize a convincing animation in this manner.

By adding a component that implicitly computes the laws of rigid body physics, this problem is reduced to determining only the forces that apply at the joints. This task is still daunting but arguably less complicated, because the effects of gravity, masses, inertia and joint limits are implicit. Current research is attempting to synthesize an adaptive process that automatically controls the motion of the body [29] [30]. In this case, the animation automatically adapts to the surface features of the underlying terrain.

2.3.1 Rigid Body Dynamics

Mechanical systems naturally depend on properties such as friction, masses and forces. The laws of physics are important factors which are explicitly expressed in the design and behavior of mechatronic systems. Control logic often has to monitor physical quantities and an effective mechatronics simulation tool has to express the appropriate processes as well. Physics-based animation libraries, or *physics engines*, are now emerging as software components that simulate rigid body dynamics in terms of geometries, masses, kinematic relationships and friction. Most engines also support different types of mechanical joints, including hinges, ball bearings, linear and universal joints.

The majority of physics engines are based on discrete time step algorithms [31]. At the beginning of each step, collisions are detected and the forces at each contact point are computed using the momentum and mass of the colliding objects as well as the elasticity and coefficients of friction at intersection points [32]. The contacts and the restrictions imposed by mechanical joints comprise a system of constraints which determines the reaction forces of the current step. However, since time advances discretely, objects can also interpenetrate. Each set of contacts therefore includes all the collisions since the last time step.

For example, two colliding tea pots could have penetrated at several places with different depths. The solution space for the reaction forces is therefore *over constraint*. Simply using the current penetration depths to compute the reaction forces is not always the best solution, because had a collision been detected earlier, potentially deep penetrations may not have occurred at all. Generally, Linear Complementary Programming (LCP) techniques are used to solve the over-constraint system by intelligently relaxing selected constraints [33] [34].

$3D^{os}$ integrates the Open Dynamics Engine (ODE), which is freely available

under an Open Source license [35]. The library can simulate hundreds of joints and bodies with good real-time performance. It has already produced satisfactory results when modelling the kinematics of mechanical systems and articulated characters [36]. The terrains and architectural structures are both closely integrated with the dynamics engine, by ensuring that the proper contact points are generated during collisions. Animation control is therefore simplified to applying forces rather than defining key-frame animations. However, due to the problems mentioned above, bodies sometimes display erratic and unnatural movement.

The dynamics engine is integrated via the geometric node types such as cube, sphere, cylinder and triangle mesh, as well as some special node types such as world, body and joint. The order of the Scene Tree implicitly creates the relationships between the dynamic artifacts. For example, if the parent node of a cube is a body node, then a cube-shaped body is also being created in the dynamics engine. If the parent is not a body, then the cube becomes a static object in the environment. Other bodies will still collide with the cube, but it will never move itself. Another example is how joints are connected to bodies. A joint implicitly connects its parent and child nodes, if they are both bodies. If the parent node is not a body then the joint is statically connected to the environment.

When a world node is encountered during the rendering traversal then the *timestep* function of the dynamics engine is called. At that point, the engine generates contact points and solves the resulting system of constraints in order to obtain the reaction forces. As a result, the positions and orientations of the bodies is being updated and, as the traversal continues, the geometries are painted at new locations. When a joint is encountered, then optional linear or rotational forces can be applied. At the next call to *timestep* the dynamics engine takes the new forces into consideration when computing the location of the corresponding bodies.

2.4 Manipulators

Manipulators are 3D interface widgets used to edit objects in the context of the 3D model. Each manipulator consists of interactive parts which respond to mouse and keyboard input in order to perform different editing functions such as scaling, transformation or rotation. A manipulator is also a Scene Tree node that replaces existing nodes when selected by a user. The manipulator adds the selected node to its list of children and then scales its interactive parts to fit the bounding box of the node. At the end of an editing operation the manipulator traverses the scene tree and applies the corresponding editing changes before removing itself from the tree and reinstating the selected node.

Panes A - E in Figure 2.2 show the types of manipulators currently supported by $3D^{os}$. Manipulator A is used to scale objects and manipulator B is used to translate or rotate. Manipulators C and D are extensions of B designed to specify the parameters of mechanical joints, including position and orientation in space as well as the movement limits around the joint axes. Manipulator E is used to specify a spot light source, including the direction and size of the spot.

All green colored parts of a manipulator are sensitive to dragging using the mouse. The corners of the cube defined by manipulator B, for example, are used to freely rotate an object while the edges restrict rotation around the cylinder defined by the edge and the center of the manipulator. By clicking and dragging anywhere on one of the six surfaces of the cube, an object can be translated along the corresponding plane. In manipulator A, any of the green tabs can be used to scale an object. The tabs on the center of an edge scale along a single axis and the tabs in the corners scale along two axes.

The conceivable shapes and functions of 3D manipulators are as diverse, if not more so, than those of 2D interface widgets. It is not yet clear if the manipulators

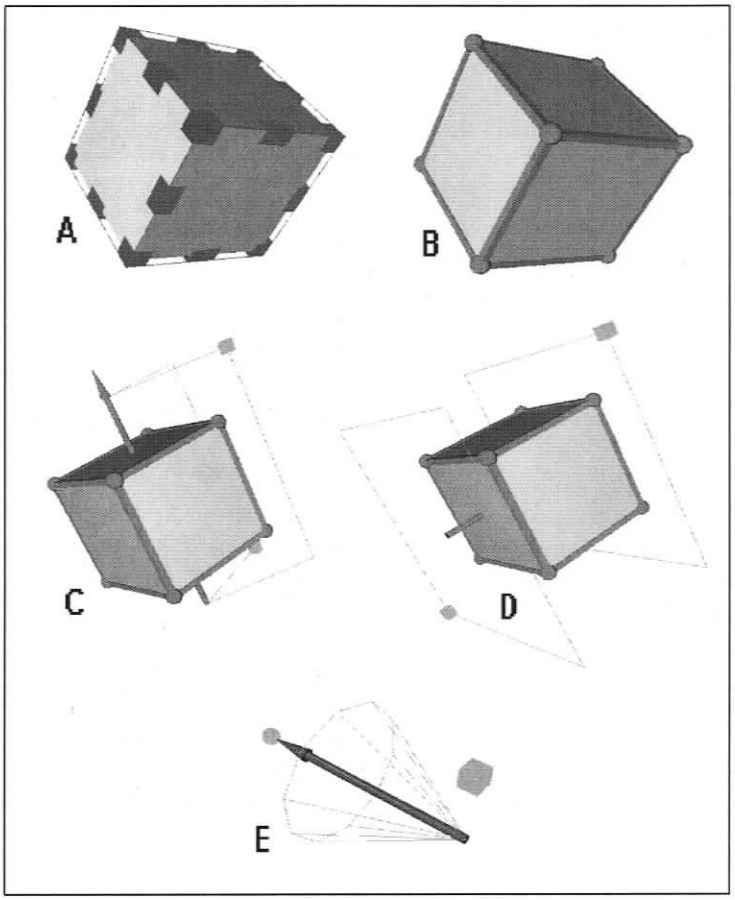

Figure 2.2: 3D manipulators of $3D^{os}$

introduced here are going to be the ones that will finally be adopted. Also, as the development environment evolves and as new node types are developed, new types of manipulators are likely to follow.

2.5 Sensors

The defining characteristic of a virtual reality system is the ability to generate sensor telemetry. This section discusses which types of sensors are already supported and how they are integrated. So far, only a few simple sensors such as collision detectors and range finders are fully implemented, but more complex types such as Gyros, GPS receivers and vision sensors are contemplated. In order to model relevant mechatronic systems, more sensor types must ultimately be developed.

Sensors are basically Scene Tree nodes which encapsulates a tree traversal that computes the corresponding value. The accuracy of the measurement depends on how well an artifact is being modelled. For example, a range finder can only detect distances with the resolution at which objects are tessellated. For a cube, the measurement can be quite accurate, but for a sphere it depends on how many triangles are used to represent its surface. The position of sensor nodes within the Scene Tree can also help to optimize computing the corresponding values. For example, a range finder can be configured to apply only to its siblings or descendent. Organizing a Scene Tree accordingly can greatly reduce the time required to compute the sensors.

Figure 2.3 A - D depicts the four different types of collision sensor supported by $3D^{os}$. Each sensor detects when an object enters the region defined by the corresponding shape. Figure 2.3 A, B and C represent basic geometric shapes and D represents an infinitely long ray. The collision detection mechanism used is the same as the one of the Open Dynamics Engine except that no contact points are generated when an object collides with a sensor. The same manipulators as shown in Figure 2.2 can also be used to position the sensors and define the size of their respective activation regions.

The notion of sensors also extends to keyboard and mouse input. For example, clicking on a valve in the virtual environment can simulate an operator opening

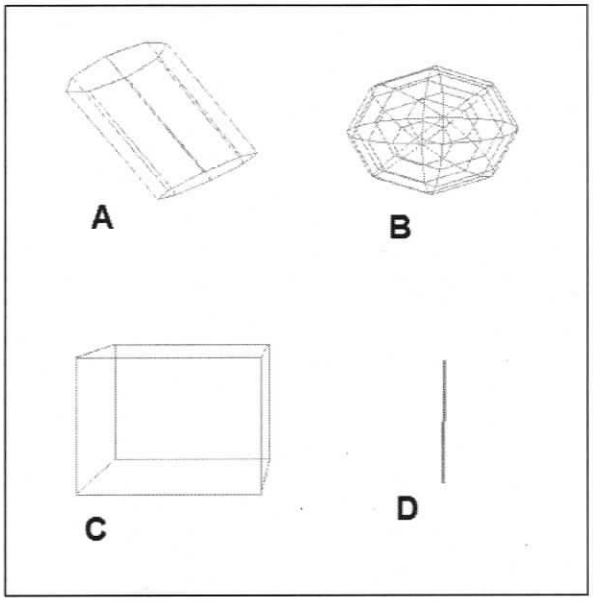

Figure 2.3: 3D collision sensors of $3D^{os}$

that valve. Mouse and keyboard sensors are based on nodes that define regions of activation. For example, a keyboard sensor only detects key presses if the bounding box defined by the sub-tree of its parent node intersects with the viewpoint form which the scene is being rendered. In other words, a keyboard sensor is only active if the observer of the simulation is close enough to a certain area.

A special type of node and corresponding sensor are gradient fields. A gradient field encapsulates a 2D array of vectors that annotates a scene with contextual information such as a desired path through space [37]. The corresponding sensor samples the field at (x,y) grid positions. Gradient fields are not used in any of the examples outlined in this thesis, but the software distribution of $3D^{os}$ contains a model that shows how autonomous vehicles navigate a terrain using gradient fields.

2.6 Open Inventor

At first, the *Open Inventor* toolkit was used as the virtual reality component for $3D^{os}$ [38]. However, after some development and use, it became clear that Open Inventor had drawbacks which led to question whether it can meet future requirements. The library hides its internal data structures too strictly and limits access to the underlying OpenGL graphics language. Design decisions therefore had to be made that led to inefficient rendering and duplication of data. Open Inventor also performs automatic memory management, which should be a function separate from a graphics library.

The drawbacks of *Open Inventor* led to prohibitive computational overhead when rendering large-scale models. Thus, it was decided to re-implement the library and customize it for Geometry-driven Petri Nets. However, much of the interface and class hierarchy of Open Inventor is well structured and therefore has been adopted as part of the redesign. The new implementation contains all the mayor concepts of the original toolkit as well as the additional data structures. However, they were especially designed to integrate well with the Petri Net kernel and Python. For the remainder of this dissertation, the term $3D^{os}$ *Inventor* denotes this new derivative of Open Inventor.

2.7 Summary

This chapter introduced the virtual reality component of $3D^{os}$. At this point, the software already supports enough features for demonstrating an application of Geometry-driven Petri Nets. However, there are still many types of mechatronic systems which cannot be expressed yet, because some of their components cannot be simulated. In some cases, the technology required to compute a certain artifact in real time does not even exist yet. Developing the virtual reality component is

therefore an ongoing project which will not be completed for some time to come.

Chapter 3

Hybrid Modelling Languages

Hybrid system languages are used to model interdependent discrete and continuous processes [7]. A good example is the automation processes in an automobile manufacturing plant which assembles cars from smaller components. At a high level of abstraction, the process consists of sequences of discrete steps which often have to be synchronized. A modelling language such as Petri Nets or Finite Automata can describe the corresponding work flow using graph structures. The homogenous iconic constructs support formal analyses and help to organize and comprehend complex models. The continuous control processes in the plant consist of timed tasks such as transportation, welding or milling. Due to the nature of computing devices, discrete time step algorithms must be used to compute the input signal space for the corresponding electromechanical actuators. A hybrid modelling language combines a language for specifying algorithms with the constructs of a discrete-event language.

3.1 Language Histories and Evolution

Hybrid modelling languages have emerged form formalisms that were originally designed to express exclusively discrete-event or continuous control flow. The most common discrete-event formalisms are State Machines and Petri Nets. However, the

two languages are so closely related that there is little purpose in discussing them individually. Both are based on a transition network in which state changes occur instantaneously and in discrete step. The only difference is that Petri Nets include additional graphical artifacts that help to express concurrency and synchronization, which traditional State Machines cannot; at least not intuitively. Both languages have also evolved to better describe timed discrete-event systems in which state transitions occur over a period of time. An interval generally represents a continuous process in the physical world that leads from one state to the next. In a practical application such as a simulation tool, it is possible to evaluate a continuous function over the duration and effectively model a hybrid system.

Another unique hybrid modelling language is based on Bond Graphs [39]. A Bond Graph is a graph-based formalism designed to capture the energy structure of a system in terms of effort and flow. Different types of Bond Graphs are used to expresses the continuous flow in electrical, mechanical, thermal or hydraulic systems. For example, an electrical motor driving a pump can be expressed using a Bond Graph. One edge denotes the relationship between electrical power and the torque of the motor, and another edge denotes the relationship between the torque of the motor and the pressure generated by the pump. The flow relationships are expressed by first order differential equations and the goal of a Bond Graph is to maintain equilibrium among the edges. Bond Graphs have been adopted to model hybrid systems by adding a switching component that rearranges the graph structure in discrete steps [40]. The switching component, which may well be a traditional discrete-event language, models discrete supervisory control tasks such as flipping a switch. In addition to time, Petri Nets and State Machines have also been extended using continuous flow along the edges, as in hybrid Petri Nets [41] and Hybrid Automata [42].

The research presented here investigates how to use Petri Nets as hybrid mod-

elling language simply because it can intuitively express a complex distributed system and because there is no literature that applies the language in conjunction with a virtual environment. Bond Graphs, on the other hand, are already used to control 3D visualizations in Matlab. Also, only timed Petri Nets have been considered in this dissertation. However, Hybrid Petri Nets are likely to become a more important modelling formalism in this domain, because they can express both, continuous and discrete processes using almost exclusively iconic artifacts.

3.2 Petri Nets

Petri Nets were first introduced by Carl Adam Petri in 1962 in his Ph.D. thesis submitted to the Department of Mathematics and Physics at the Technical University of Darmstadt, Germany [43]. Since then, Petri Nets have received increasing attention from researchers around the world, and now there exists a number of extensions to the original language as well as software development tools to edit, simulate and analyze the different dialects.

A Petri Net is a directed graph consisting of two different node types called *places* and *transitions*. Under the right conditions, a transition can *occur* or *fire*, and consume *tokens* from its incoming places and deposit new ones on the outgoing places. An additional *capacity* function defines the number of tokens a place can hold, and a *weight* function defines how many an arc consumes or produces. The *marking* of the Petri Net refers to the set of tokens currently residing on all the places. As transitions occur they continuously change the marking from m_i to m_{i+1}. The formal definition of the Place Transitions Petri Nets (PT-PN) introduced by Petri is a tuple

(i) A PT-PN is a tuple $\langle P, T, A, w, k, m_0 \rangle$

(ii) P - set of places

(iii) T - set of transitions

(iv) A - set of arcs: $A \subseteq (P \times T) \cup (T \times P)$

(v) w - weight function of A: $w(a) \mapsto N^0 : a \in A$

(vi) k - capacity function of P: $k(p) \mapsto N^0 : p \in P$

(vii) m_0 - initial marking: $m_0(p) \mapsto N^0 : p \in P \wedge m_0(p) \leq k(p)$

The definition so far only describes the structure of a Petri Net and it remains to specify the execution semantics that move tokens throughout the graph structure. Figure 3.1 a-c illustrate this process using three markings of a simple Petri Net consisting of four places and two transitions, T1 and T2. The center of an enabled transition is colored and the one of a disabled transition remains transparent. Formally speaking a transition t is enabled by a marking m if

$$\forall \{p,t\} \in A, m(p) \geq w(\{p,t\})$$

and

$$\forall \{t,p\} \in A, m(p) + w(\{t,p\}) \leq k(p)$$

In other words, a transition is enabled when all of its surrounding arcs can consume or produce the amount of tokens specified by their respective weight function. When t occurs, it changes the current marking m to

$$m'(p) = \begin{cases} m(p) & \forall \{p,t\} \notin A \wedge \{t,p\} \notin A \\ m(p) - w(\{p,t\}) & \forall \{p,t\} \in A \wedge \{t,p\} \notin A \\ m(p) + w(\{p,t\}) & \forall \{p,t\} \notin A \wedge \{t,p\} \in A \\ m(p) - w(\{p,t\}) + w(p,t) & \forall \{p,t\} \in A \wedge \{t,p\} \in A \end{cases}$$

In Figure 3.1 a) transition T1 is enabled because there is a token available on each of its two incoming places and there is capacity available on its outgoing place. In

this case, the weight of all arcs and the capacity of all places is assumed to be one. When T1 occurs it enables T2 in Figure 3.1 b) by adding a token to its incoming place and by removing the one from its outgoing place. After T2 has occurred, no other transition is enabled and computation is halted. While this example is trivial, it illustrates how a simple graph structure and a set of rules can define a concurrent discrete-event processing system. The structure of the Petri Net defines the possible flow of control and the tokens represent the current state. Transitions occurring in parallel model concurrency and their enabling rules synchronize.

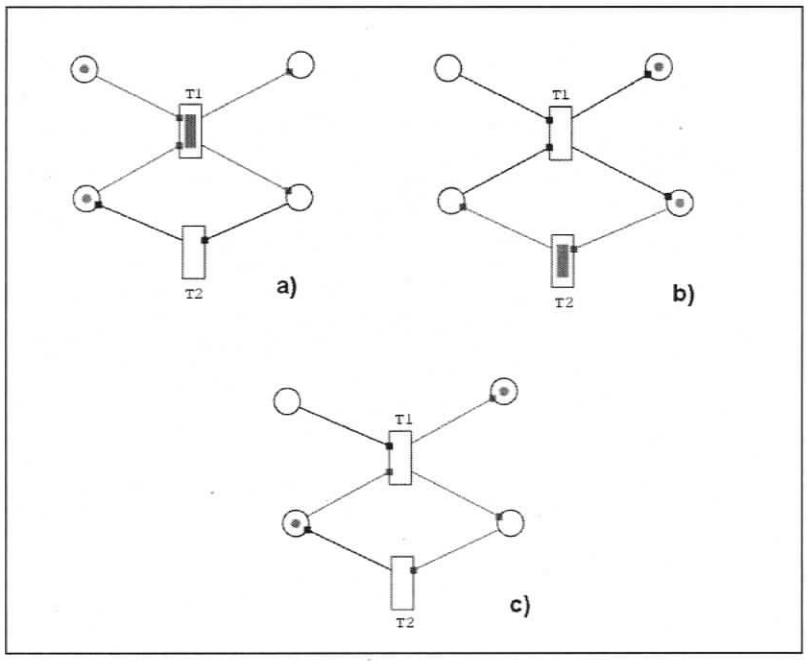

Figure 3.1: Three steps in a simple Petri Net

The advantage of Petri Nets lies in their ability to describe a complex concurrent processing system in an intuitive and predictable manner using only a few graphical constructs. The tokens model concurrency and synchronization and carry

information in a manner that is unlike any other formalism in this domain. Petri Nets also have a formal mathematical basis which supports a wealth of analyses. Various applications have already been documented in which Petri Nets have been used to model and analyze different types of systems ranging form communication protocols [44] and industrial work flows [45] to biological [46], molecular [47] and even quantum [48] processes.

3.2.1 Behavioral Properties

A complete coverage of the behavioral properties of Petri Nets is outside the scope of this dissertation. Yet, a general introduction is necessary to show their practical value in analyzing control systems. An excellent reference on this topic is a paper by Tadao Murata [49], who distinguishes between *structural* and *marking dependent* analysis. The former depends only on the structure of the graph and the latter depends on the initial marking as well.

A structural analysis can make general statements about the quality of the underlying process model based on the topology of the graph structure. It determines if particular markings are reachable from other markings under certain conditions. For example, a Petri Net is said to be *Controllable* if any marking can be reached from any other marking. If not, then there exists irreversible state transitions in the process model. Similarly, a Petri Net is said to be partially *Repetitive* if there exists a marking from which an infinite sequence of transition occurrences is possible. If it is not repetitive then the control flow will come to a halt.

A marking dependent analysis is based on the reachability graph of the initial marking. It explores the possible state space and determines properties that may only hold true for this particular initial condition. For example, a marking is *Reachable* when a sequence of occurring transition leads there from the initial marking. If a reachable marking corresponds to an invalid state, then there is an error in the

control flow. Similarly, a Petri Net is said to be *Lively* if there exists an occurrence sequence from any reachable marking which includes every transition. If a Petri Net is lively, then there are no deadlocks in the control flow.

There are many other examples of analysis, most of which are based on exploring the reachability graph. However, an exhaustive search is not always possible, especially when a Petri Net contains too many cycles, or in the case of Petri Net types which contain stochastic process. In this case, a simulation of the dynamic behavior is used instead of a formal analysis. Interpolating the dynamics naturally explores the most likely state space, but the process is generally not exhaustive and cannot conclusively verify a property. However, it serves to validate results with reasonable confidence. In the case of this dissertation, the Petri Nets is first and foremost used to drive the animation of a virtual environment and, at the same time, validate the Petri Net models through simulation.

3.2.2 Structural Properties

The term *structural properties* refers to the patterns of Petri Net constructs that describe artifacts of a distributed processing system, such as concurrency, communication and synchronization. Following is an explanation of how the structural patterns in Figure 3.2 compare to procedural programming language constructs.

A *sequence* is simply a linear arrangement of alternating connected places and transitions. This configuration is analogous to sets of statements in a sequential program. A *repetition* is a cycle in the graph structure where tokens can repeatedly trace the same path. It is analogous to a loop construct in a programming language.

A *conflict* occurs when several transitions simultaneously attempt to remove or deposit tokens at the same place. If that place does not have enough capacity, then some of the transitions are prohibited from occurring. This configuration is analogous to mutual exclusion in procedural programs where only a fixed number

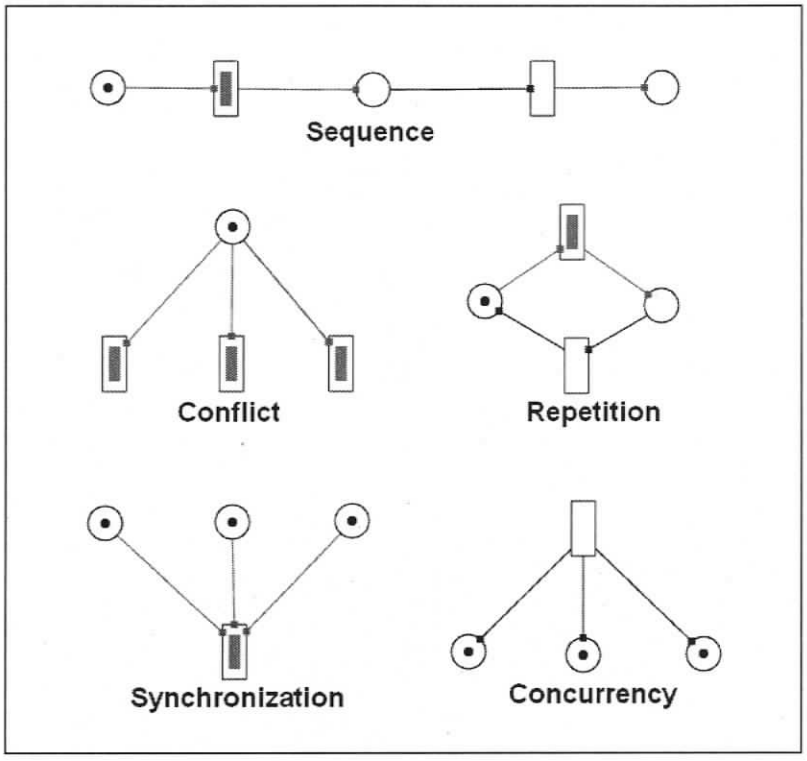

Figure 3.2: Common Petri Net patterns

of threads may access protected memory at the same time.

Synchronization occurs when one transition requires tokens from a number of different places. In this case, the transition cannot occur until all of the places are occupied. This configuration is analogous to semaphores in procedural programming, where a thread is blocked until others have released it.

Concurrency occurs when one transition deposits tokens onto a number of different places at once. New areas of computation within the graph structure are activated by allowing more than one other transition to occur. This configuration is analogous to spawning new threads in a procedural programming language.

3.3 Colored Petri Nets

Colored Petri Nets (CPN-PN) were first introduced by Kurt Jensen in 1981 [50, 51, 52]. Colors, or types, emerge from a simple Petri Net by folding, or merging, different places or transitions into one. The tokens of the original constructs are represented as different colors in the merged ones. Large enough color sets can define arbitrary data types including objects. CPN-PNs therefore integrate especially well with textual programming languages. A consequence of introducing types is more complex arc expressions, which define enabling rules in terms of the values of tokens rather than only their absence or presence. Formally, the structure of CPN-PN is defined as

(i) A CPN-PN is a tuple $< \Sigma, T, A, c, k, e, g, m_0 >$

(ii) Σ is a set of types called a color set

(iii) P is a finite set of places

(iv) T is a finite set of transitions

(v) A is a finite set of arcs : $A \subseteq (P \times T) \cup (T \times P)$

(vi) c is a color function : $c(x) \mapsto \Sigma^* : x \in (P \cup T)$

(vii) k is a place capacity function : $k(p) \mapsto c(p)_{MS}$[1]: $p \in P$

(viii) e is a arc expression function : $e(a) \mapsto c(p)_{MS} : a \in A$

(ix) g is a transition guard function : $g(t) \mapsto \{true, false\} : t \in T$

(x) m_0 is the initial marking : $m_0(p) \mapsto c(p)_{MS} : p \in P \wedge m_0(p) < k(p)$

[1]The subscript "MS" indicates a multi-set, which Jensen defines as "allowing multiple appearances of the same element" [50] pp. 22

The above definition is basically identical to the structure of a PT-PN, with the exception that color sets instead of tokens occupy the places. In addition to tokens, CPN-PNs also contain variables which can be parameters to the arc expressions. While the marking is a result of the computation within the Petri Net itself, the variables are bound to external systems. The enabling of transitions therefore depends on the values of the variables as well as the current marking. A *binding* specifies the values of a set of variables at an instance in time. A step Y, contains a multi-set of bindings and transitions for which

$$\sum_{(t,b)\in Y} e(p,t)\langle b\rangle < m(p)$$

and

$$\sum_{(t,b)\in Y} e(t,p)\langle b\rangle < k(p) - m(p)$$

where $m(p)$ denotes the current marking of place p. This definition implies that for a particular binding there must be enough tokens on the incoming places and enough room on the outgoing places in order to enable a transition. The condition is evaluated by comparing the color sets produced by the arc expressions with the marking or capacity of the corresponding places. $e(t,p)\langle b\rangle$ means that the arc expression is being evaluated under the binding $\langle b\rangle$.

In the original PT-PN, this comparison is clearly defined by the arithmetic meaning of the operators $<$ and $>$. However, in CPN-PNs the meaning differs for each element in $c(p)$, and can therefore not be defined formally until the color space Σ is known. For example, for strings a comparison may be based on the number of characters or on an alphabetical ordering. The transitions contained within a step may fire and change the marking of the net m_x to m_{x+1} according to

$$m_{x+1}(p) = \left(m_x(p) - \sum_{(t,b)\in Y} e(p,t)\langle b\rangle \right) + \sum_{(t,b)\in Y} e(t,p)\langle b\rangle$$

The above definition is also similar to the PT-PN with the exception that $+$ and $-$ are not simply arithmetic operators. They also have a different meaning for each type.

Conflicts between transitions may arise when they require overlapping sets of tokens from the same place. The first occurring transition consumes some of the tokens required by the others, and consequently disables them before they can occur. The guard function g controls the order in which enabled transitions should occur. The guards may include random variables and priorities, or they may even be bound to external variables.

3.4 Summary

This chapter discussed the characteristics of hybrid systems and explained how a hybrid modelling language can be used to model control systems. The Petri Net formalism has been introduced and it has been illustrated how its graphical language constructs can describe concurrent discrete-event processing systems. CPN-PNs have also been introduced, because they integrate especially well with the data structures of procedural programming languages. The next chapter defines an extension of Colored Petri Nets called Geometry-driven Petri Nets.

Chapter 4

Geometry-driven Petri Nets

Geometry-driven Petri Nets (GD-PN) are a new extension of Timed and Colored Petri Nets. The novelty is that the variables, or the binding $\langle b \rangle$, which determine the next step in the Petri Net depend on the state of the virtual environment. The tokens and variables consist of Scene Tree nodes and the inscriptions contain user defined logic that interprets sensors and controls actuators of a virtual mechanical system.

Figure 4.1 depicts a high level overview diagram of the simulation architecture. Operating within a virtual environment are software *agents* capable of sensing and manipulating their surroundings. Agents communicate directly by exchanging tokens, or indirectly by altering the state of the environment. The entire system executes in a single thread of execution and all concurrency is exclusively expressed using Petri Nets. Geometry-driven Petri Nets can also be thought of as an operating system that uses Petri Nets as a scheduler and Python scripts as the end user programming language; hence the name $3D^{OperatingSystem}$.

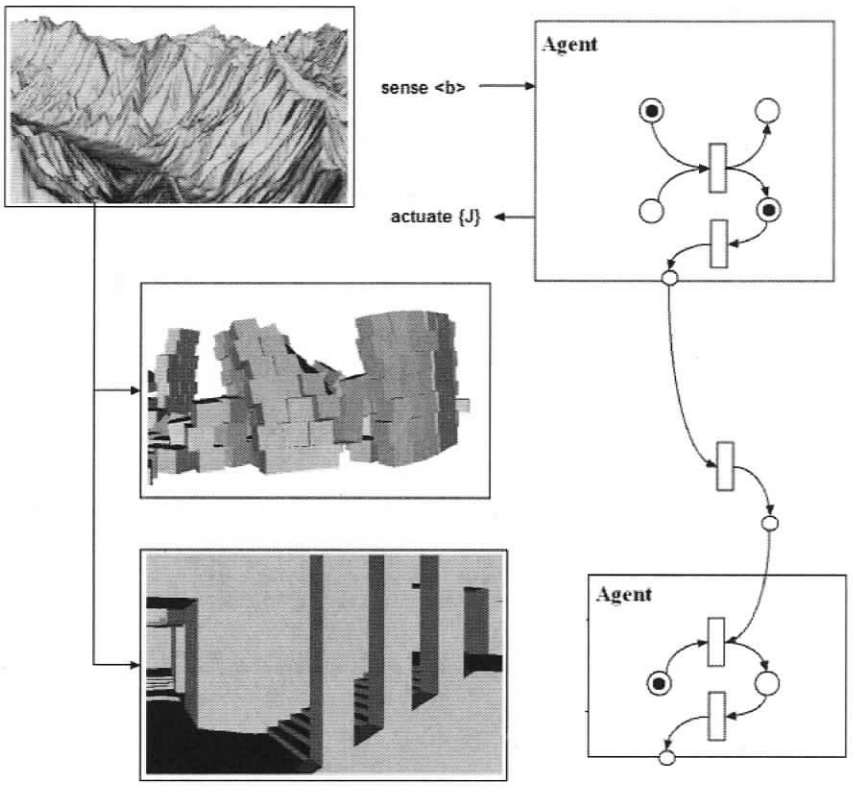

Figure 4.1: Simulation architecture

4.1 Formal Definition

Timed transition occurrences were originally introduced by Ramchandani in order to describe and analyze timed dynamical systems more effectively [53]. In Geometry-driven Petri Nets, the durations are also used to execute discrete time step algorithms. The occurrence intervals generally reflect the durations of physical processes and are therefore useful in evaluating the corresponding animation functions.

Firing a timed transition is a three-phased process in which tokens are immediately consumed when a transition occurs, and deposited when the firing duration

has expired. During that time, capacity has to be reserved on the outgoing places so that enough space remains in order to deposit tokens at the end of the occurrence interval. The formal definition therefore requires an additional *reserve* function and a set of durations:

(I) D is a set of firing durations : $D \subseteq (\delta \times R^0)$

(II) r is the reserve function : $r(p) \mapsto c(p)_{MS} : p \in P$

The reserve function r specifies the amount of capacity reserved on a particular place while the surrounding transitions occur. A less common extension to Petri Nets applies time to places instead of transitions [54]. In this case, tokens *submerge* in a place and remain hidden to the outgoing arcs until they *emerge* again at the end of the duration. It seems that the different semantics of timed transitions and places still lead to similar behavior, and the decision was made to support only timed transitions until a need for timed places arises.

The reserve must also become part of the enabling rules. A step Y is now enabled if

$$\sum_{(t,b)\in Y} e(p,t)\langle b\rangle \leq m(p)$$

and

$$\sum_{(t,b)\in Y} e(t,p)\langle b\rangle \leq k(p) - m(p) - r(p)$$

When a transitions fires, it changes the marking of the places from which it consumes tokens and the reserve of the ones at which it deposits tokens. Step Y therefore changes the marking of the net to

$$m_{x+1}(p) = m_x(p) - \sum_{(t,b)\in Y} e(p,t)\langle b\rangle)$$

and the reserve to

$$r_{x+1}(p) = r_x(p) + \sum_{(t,b)\in Y} e(t,p)\langle b\rangle$$

However, the step is not yet complete. It will remain active until all of the transitions have completed firing. Y_τ denotes a subset of Y that contains those transitions that have completed firing between time τ and τ_{-1}. The lower bound of this interval is the frame rate of the animation. Y_τ changes the reserve of the Petri Net to

$$r_{x+1}(p) = r_x(p) - \sum_{(t,b)\in Y_\tau} e(t,p)\langle b\rangle$$

and the marking to

$$m_{x+1}(p) = m_x(p) + \sum_{(t,b)\in Y_\tau} e(t,p)\langle b\rangle$$

The operators $<$, $>$, $+$, $-$ are implicitly defined by the corresponding data types, and the arc expression functions e are only evaluated after an initial comparison has been satisfied. From the point of view of a user, arc expressions are therefore Boolean functions that determine whether the colors meet application specific conditions for enabling transitions.

So far, all tokens are treated as discrete entities similar to the type-less tokens in PT-PNs. The operators $<$, $>$, $+$ and $-$ are therefore arithmetic functions that add and remove entities from the color sets and that compare the size of the sets. In the future however, the meaning of these operators may change. For example, adding a node containing a triangle mesh to a place, adds that node as a discrete entity to the color set of the place. Yet, it is also conceivable that this operation merges the triangles of the mesh with another node. In this case, only one node remains containing the triangles of both meshes.

In addition to the arc expression, *Geometry-driven* Petri Nets also includes four sets of Python inscription functions, which are attached to transitions and places. Depending on the graph structure and firing sequences of the Petri Net, different sets of tokens are passed to the inscriptions as parameters.

(I) J_p are place inscriptions : $J_p(c,p) : p \in P \wedge c \in c(p)$

(II) J_{t-} are pre-fire inscriptions : $J_{t-}(t) : t \in T$

(III) J_t are in-fire inscriptions : $J_t(t,\delta) : t \in T \wedge \delta \in D$

(IV) J_{t+} are post-fire transition : $I_{t+}(t) : t \in T$

The place inscription $J_p(c,p)$ of place p is invoked when the color c changes. For a token color, this is the case when the marking changes and for a variable color when the binding changes. Since variables represent sensors, some of the place inscriptions are triggered by changes in the environment.

The pre-occurrence and post-occurrence inscription is invoked once before and after a transition occurs, and the in-occurrence function is continuously evaluated at the frequency of the frame rate while the transition is occurring. Conflicts between transitions are either resolved by guard functions or by the order in which they have become enabled. Users chose the guard functions when constructing a model similar to how they choose the initial marking. The supported guards include probability and priority functions that enable a transitions x in step Y when

$$\forall t \in Y, t \neq x : Priority(t) < Priority(x)$$

or when

$$Probabilty(x) < \rho$$

where ρ is a normal random variable between [0, 1].

In order to avoid redundancy and add structure to large scale systems, it is important to support hierarchal nesting of Petri Nets. A *page* is represented by a single graphical icon that encapsulates an entire Petri Net including an interface of selected constructs [55][56]. *Place refinement* or *transition refinement* are specialized forms of hierarchical Petri Nets, where pages publish only places or transitions [57]. Higher-order Petri Nets even allow entire pages to become tokens [58]. However, this extension requires a new form of *port place* in order to provide an interface to the *page-carrying* tokens.

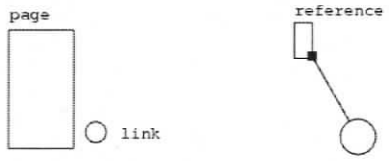

Figure 4.2: Page, Reference and Link

Hierarchical Geometry-driven Petri Nets are based on place refinement, in which only places from inside a page are published. From within, a page looks like a normal Petri Net with the addition of *references* attached to selected places. From the outside, a page appears as a large rectangle surrounded by one *link*, or hole, for each reference as depicted in Figure 4.2. Drawing an arc between a transition and a link connects the corresponding place inside the page to the transition. Since the references and links are proxies for transitions and places, they also have similar shapes. Formally a page is defined as follows

(i) A Petri Net page is a tuple $\langle PTN, R \rangle$

(ii) PTN is a Petri Net.

(iii) R is a set of references : $R \subseteq P$, where P is the set of places in PTN.

The challenge of supporting hierarchical Petri Nets is to develop the data structures, algorithms and interface widgets necessary to instantiate and connect nested systems of pages. It essentially requires a type system to support this construct in order to manage instantiations and propagate editing changes to all instances.

4.2 Runtime Kernel

Following is an outline of a runtime kernel for Geometry-driven Petri Nets. The algorithm describes how to emulate the dynamic behavior of the Petri Net constructs efficiently during simulation. An attempt is made to use the same symbolism of the formal definitions introduced earlier in order to avoid additional terminology. Programming language construct are only used to denote data members of transitions. An additional set E, which contains all of the currently enabled arcs, is also introduced. Following each definition is an informal explanation of the algorithm and Appendix A contains the source code of the corresponding C++ classes.

UPDATE_REMOVE(p, A, E, Y)

$\forall\{p,t\} \in A$
  $if(m(p) < e(p,t)))$
    $if(\{p,t\} \in E)$
      $if(t.disabledArcs = 0)$
        $Y = Y - t$
      $t.disabledArcs = t.disabledArcs + 1$
      $E = E - \{p,t\}$
$\forall\{t,p\} \in A$
  $if(k(p) - m(p) - r(p) >= e(t,p)))$
    $if(\{t,p\} \notin E)$
      $t.disabledArcs = t.disabledArcs - 1$
      $if(t.disabledArcs = 0)$
        $Y = Y + t$
      $E = E + \{t,p\}$

Function $UPDATE\_REMOVE$ evaluates the enabling rules of the transitions surrounding a place p when tokens are removed. The function only tests if arcs leaving the place become disabled, or if arcs entering the place become enabled by the *lesser* marking. The opposite cases are not tested, because they cannot occur when tokens are removed. When an outgoing arc of p is disabled, then the *disabledArcs* count of the neighboring transition is incremented. If the count was zero before, then the transition becomes disabled as well, and must be removed from the current step Y. Similarly, when an incoming arc of p is enabled, *disabledArcs* is decremented. If the count has become zero, then the transition is added to Y. Each place maintains a list of incoming and outgoing arcs in order to compute the update functions efficiently.

UPDATE_DEPOSIT(p, A, E, Y)

$\forall\{p,t\} \in A$
  $if(m(p) >= e(p,t)))$
    $if(\{p,t\} \notin E)$
      $t.disabledArcs = t.disabledArcs - 1$
      $if(t.disabledArcs = 0)$
        $Y = Y + t$
      $E = E + \{p,t\}$
$\forall\{t,p\} \in A$
  $if(k(p) - m(p) - r(p) < e(t,p)))$
    $if(\{t,p\} \in E)$
      $if(t.disabledArcs = 0)$
        $Y = Y - t$
      $t.disabledArcs = t.disabledArcs + 1$
      $E = E - \{t,p\}$

Function $UPDATE\_DEPOSIT$ evaluates the enabling rules for the transition surrounding a place in the case when tokens are deposited. Conversely to $UPDATE\_REMOVE$, this function only tests if arcs leaving the place are enabled or if the ones entering the place are disabled. It also adds and removes transitions form the current step Y.

FIRE_TRANSITIONS(t, A, E, Y)

$\forall t \in Y$
  $Y_\tau = Y_\tau + t$
  $t.firestart = \tau$
  $t.disabledArcs = t.disabledArcs + 1$
  $J_{t-}(t)$
  $\forall \{p, t\} \in A$
    $m(p) = m(p) - e(p, t)$
    $\forall c \in e(p, t)$
      $J_p(c, p)$
    $UPDATE\_REMOVE(p)$
  $\forall \{t, p\} \in A$
    $r(p) = r(p) + e(t, p)$
    $\forall c \in e(t, p)$
      $J_p(c, p)$
    $UPDATE\_DEPOSIT(p)$
  $Y = Y - t$

The function $FIRE\_TRANSITION$ fires all of the transition in the current step Y. In order to avoid that a transition becomes enabled again while it is occurring, its *disabledArcs* count is incremented to create a virtual arc which will remain disabled while the transition fires. The transition is still visited when $UPDATE\_DEPOSIT$ or $UPDATE\_REMOVE$ is called, but it will not be added to the current step Y. The function also removes tokens from all the incoming places of a transition and reserves capacity on the outgoing ones. It also invokes the pre-occurrence and corresponding place inscriptions.

UPDATE_TRANSITION(A, E, Y)

$\forall t \in Y_\tau$
  $J_t(\tau, t)$
  $if(t.duration + t.firestart < \tau)$
    $J_{t+}(t)$
    $\forall \{p, t\} \in A$
      $r(p) = r(p) - e(p, t)$
      $m(p) = m(p) + e(p, t)$
      $\forall c \in e(p, t)$
        $J_p(c)$
      $UPDATE\_DEPOSIT(p)$
      $Y_\tau = Y_\tau - t$
    $t.disabledArcs = t.disabledArcs - 1$
    $if(t.disabledArcs = 0)$
      $Y = Y + t$

The function $UPDATE\_TRANSITION$ invokes the in-occurrence inscription of the currently firing transitions. After the occurrence interval has expired, the post-occurrence inscription is invoked and the *marking* and *reserve* of the outgoing places are updated. The *disabledArcs* count is also decremented to remove the virtual arc that was created when the transition began firing. If the count has become zero, then the transition is again added to the current step.

ADVANCE_TIME

$\forall n \in \Sigma$
  $render(n)$
$\forall s \in \Theta$
  $compute(S)$
  $\forall p \in P$
    $if(s \in C(p))$
      $J_p(s)$
      $UPDATE\_DEPOSIT(p)$
      $UPDATE\_REMOVE(p)$
$UPDATE\_TRANSITIONS$
$FIRE\_TRANSITIONS$

Function $ADVANCE\_TIME$ advances the simulation by a single time step. First the nodes of the Scene Trees are rendered in order to generate a new image for

display on the screen. This process also updates all of the algorithms embedded in the nodes, such as rigid body dynamics or binary space partition, and computes the values of the sensors. For each place that contains a sensor in its color set, the place inscriptions are invoked and the surrounding arcs are updated. Finally, all of the occurring transitions are updated and the ones in the set Y are allowed to occur.

4.3 Performance

Performance is still a critical factor for 3D graphics applications and therefore requires careful attention. In order to achieve a continuous animation, the screen has to be refreshed with a new image at least 30 times a second, and for acceptable performance, at least 10 times a second. In this case, the amount of time required to render each image depends on how much time it takes to render the 3D scene and update the Petri Net. Both operate in lock step and the Petri Net has to be updated after every rendering step. Hence, the frame rate and overall performance of the simulation depends on the performance of both.

The runtime performance of software is related to the number of input artifacts n which must be processed. This function is not always linear with respect to n and it is often difficult or impractical to derive an accurate model. The running time is therefore defined in terms of a function, $O(n)$, which specifies an upper bound of the worst possible running time. The following two sections provide separate analyses of the worst-case runtime performance of the graphics component and the Petri Net.

4.3.1 Virtual Reality

The most important factor in the performance of a virtual reality system is rendering the number of triangles t that define the shapes in a scene. In the case of depth buffering, each rendering step must first sort all of the triangles in a scene before they can be painted and sorting n elements takes $O(n \log n)$ time. However, opti-

mizations, such as occlusion culling and space partitioning can considerably limit the set of triangles to be sorted. Occlusion culling only sorts those parts of the scene that are actually visible to an observer, and space partitioning pre-computes spatial relationships among triangles that speed up sorting in general. However, optimizations do not work in all cases and the most time consuming step in the rendering process is still the upper bound for sorting triangles, $O(t \log t)$.

After each rendering step, all the variables in the Petri Net may have to be re-evaluated. The time required to compute the value of a variable, however, depends on its type. A time sensor, for example, requires a single operation, or $O(1)$. The value of a range sensor, on the other hand, requires an intersection test with every triangle in the scene and therefore takes $O(t)$. Yet, given appropriate space partitioning, this time can be reduced to $O(\log t)$. At this point, it is assumed that updating the variables never takes more than $O(t \log t)$ time as well.

4.3.2 Petri Net

The main factor in the performance of a Petri Net is the time spent computing the arc expressions and inscription functions. The actual content of the functions depends on the specific applications and their running time can therefore not be included in a performance analysis. However, it is possible to construct a mathematical model for the number of times that functions are invoked. The amount of time it takes to evaluate an inscription or expression is considered constant in this analysis.

A variable only triggers a computation in the Petri Net when its value changes. In this case, the place inscriptions of that color and the expressions of all surrounding arcs must be evaluated. In the worst case, all of the variables V are in one place and all of the arcs A are connected to it. If every variable changes during a step, then $V * A$ arc expressions and V place inscriptions are evaluated in the worst case. Of course, this scenario is rather unlikely and the ratio R between the number of

arcs and the number of places P is generally much lower than A. The worst-case performance of updating the variables in the Petri Net is therefore $(A/P) * V$.

After each rendering step, every enabled transition can occur at least once and the decision was made not to allow multiple occurrences per step, in order to avoid repetitive firings that can cause an infinite loop. The time required to fire a transition depends on the number of tokens it consumes and produces. Although it may produce more than it consumes or vice versa, this number always remains constant during a simulation and the time it takes to move a single token from one place to another is also constant. However, adding or removing tokens from a place requires updating the surrounding arc expressions as well. Since a transition can only fire at most once per step and always moves a constant number of tokens, the largest possible number moved during a single step is a constant K. In the worst-case scenario all transitions are connected to one place P and they all fire at once. In this case, $K * A$ arc update operations are performed. However, similar to the argument given for the enabling performance, realistically this number is $(A/P) * K$.

4.3.3 Geometry-driven Petri Nets

The runtime performance of Geometry-driven Petri Nets depends on the size and connectivity of the graph structure as well as the artifacts of the virtual environment. The previous sections have shown that the upper bound on the running time is defined by

$$O\left((A/P) * (K + V) + t \log t\right)$$

where A is the number of arcs, P the number of places, K the number of tokens and V the number of variables in the Petri Net. t is the total number of triangles in the 3D scene. The relationship between t and K is such that tokens are generally a collection of triangles. All the models constructed so far have shown that the ratio

between the tokens and triangles is orders of magnitude smaller than the number of triangles. The running time of a simulation is therefore bound by the complexity of the 3D model alone, or

$$O(C + t \log t) \approx O(t \log t)$$

4.4 Summary

This chapter defined and analyzed a new form of Petri Net and a hybrid modelling language, called Geometry-driven Petri Nets. It extends Colored and Timed Petri Nets with data and functions that can control and sample a virtual reality system. The definition of Geometry-driven Petri Nets alone, however, is only one property of the language. Petri Nets are dynamic in nature and it requires software tools to emulate their behavior over time. The next chapter introduces a development environment for Geometry-driven Petri Nets.

Chapter 5

User Interface

"Data is not information. Data must be presented in a useable form before it becomes information, and the choice of representation affects usability." [59]

A comprehensive development environment is a defining characteristic and an essential component of Geometry-driven Petri Nets; more so than for any other type of Petri Net. Virtual environments and hybrid systems contain large amounts of heterogeneous information including Scene Trees, procedural source code and graph structures. In order to construct even simple models, users have to manage diverse data sets and therefore require an effective user interface.

In a recent manufacturing simulation tool survey, 61.1% of users stated *ease of use* as the most important feature and 22.2% complained that the software they used is difficult to learn [60]. It is therefore important to develop mechanisms which support complex manipulation and navigation tasks as well as learning. In general, an effective user interface is critical in ensuring productivity and fostering the adoption of a tool.

A heuristic user interface evaluation was applied during the development of $3D^{os}$, with the caveat that the developer of the software was the only user [61]. The evaluation was based on repeatedly constructing the same simulation during the development process. Operations which took a disproportionate amount of time were then improved in the next development cycle. By continuously repeating the same tasks anyone can find better usage patterns through intuition and common sense. If that person is also the software developer, then an expert who requires

basically no learning, constantly evaluates the interface. Inevitably, a new bottleneck is discovered during each development cycle.

Some parts of the user interface have already been introduced in previous chapters. The graphical constructs of Petri Nets and the 3D manipulators, for example, are also considered part of the user interface. This section first introduces the remaining interface components and then evaluates the quality of the entire systems in terms of Greens' cognitive dimensions. Although some details may not be readily recognized in the following illustrations, the purpose is to give an impression of the scope and state of the development environment. Of course, the tool is best explored and understood when demonstrated on a workstation. A version of $3D^{os}$ can be obtained at http://www.geist3d.org/ and an installer program will automatically copy all of the software components required to run the application onto a standard Microsoft Windows platform. So far it has run successfully on Windows 2000 and Windows XP.

5.1 Application Frame

Figure 5.1 shows the application after a model has been loaded. At the top of the main window are a menu and a tool bar containing groups of buttons that allow users to activate different editing and control functions. The right part of the window toggles between the *World View*, the *Petri Net View* and the *Contents View*. The *Tree View* on the left displays the Scene Tree of the current model and the *Object Inspector* on the bottom of the window displays the data members of the currently selected node. The *Mini View* to the right displays the same contents as the *Petri Net View* in a smaller, moveable window which serves to show the Petri Net at same time as the 3D model.

The *Petri Net View* displays the traditional graphical artifacts for places, tran-

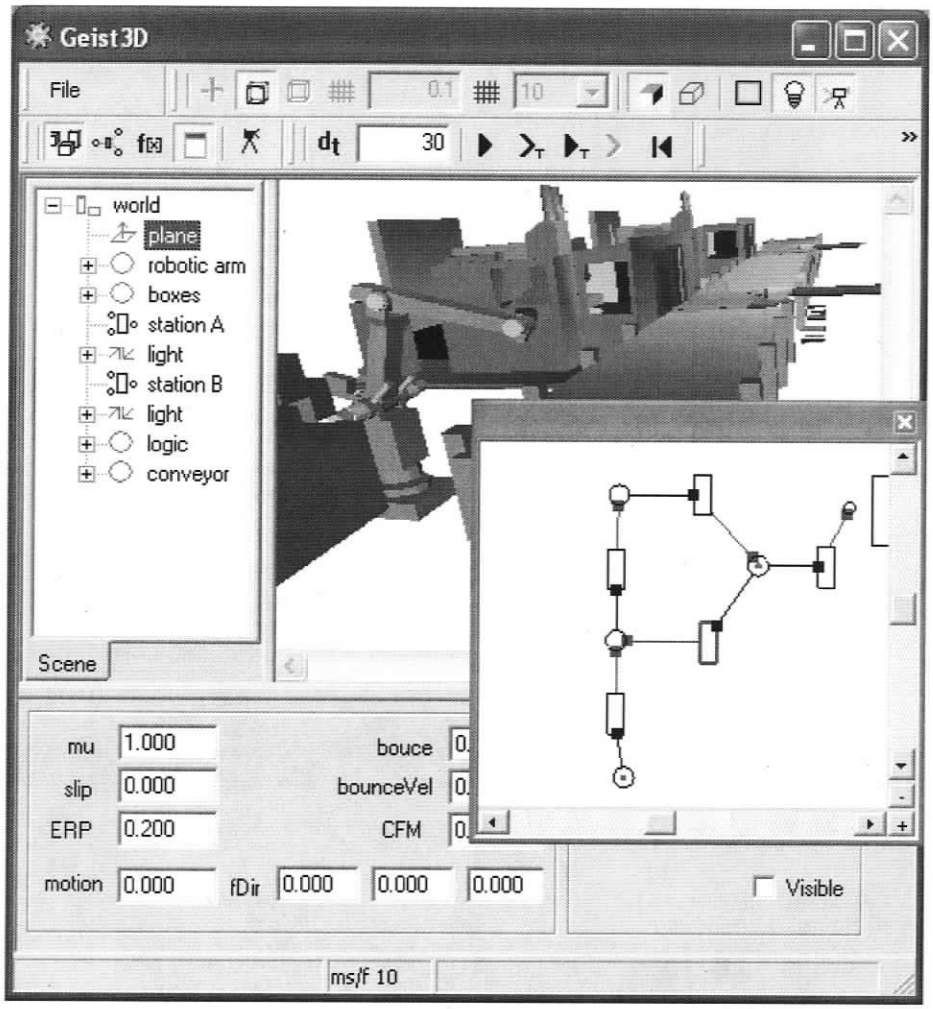

Figure 5.1: $3D^{os}$ screenshot

sitions and arcs. During a simulation, this view provides feedback about the current step Y. Differently colored dots indicate the marking of places, and different highlights indicate the enabling state of transitions and arcs. For example, the center of a transition is colored green when it is enabled, transparent when it is disabled

and red while it is firing. Similarly, arcs are colored in green when their respective expressions evaluate to true and gray when they evaluate to false. Petri Nets are edited and navigated using the mouse and keyboard. The features of the editor include drag-and-drop methods to construct the graph as well as the ability to zoom.

All of the views are linked through various interactions. Selecting a node in the *Tree View*, for example, highlights the corresponding 3D object in the *World View*, and vice versa. In addition, selecting a Petri Net page in the *Tree View*, displays its contents in the *Petri Net View*. The names of the nodes which are tokens or variables in the Petri Net are displayed in bold face in the tree view. Selecting a Petri Net construct in the *Mini View* automatically switches the application window to display the *Contents View*.

Numerous editing and simulation control functions are encapsulated by the buttons of the toolbar. For example, the kernel can advance at different levels of granularity by firing only the selected transition or by executing only the current step. It is also possible to advance time independently of the Petri Net and allow the rigid body dynamics computation to continue while the mechatronic system is *turned off*. Users can also toggle between different manipulators, snap objects to a grid or sphere, or record movies of a running simulation.

5.2 Content Views

The *Contents View* as depicted in Figure 5.2 displays the contents of the currently selected Petri Net construct including the color set and the Python scripts. Depending on the type of construct, additional widgets become visible in order to define properties such as capacity or firing duration. An 8 by 4 grid of differently colored square slots at the top of the *Contents View* represents the color set. In order to display individual slots in Figure 5.2 more clearly, the image has been shrunk vertically

and 2 columns of color slots have been removed.

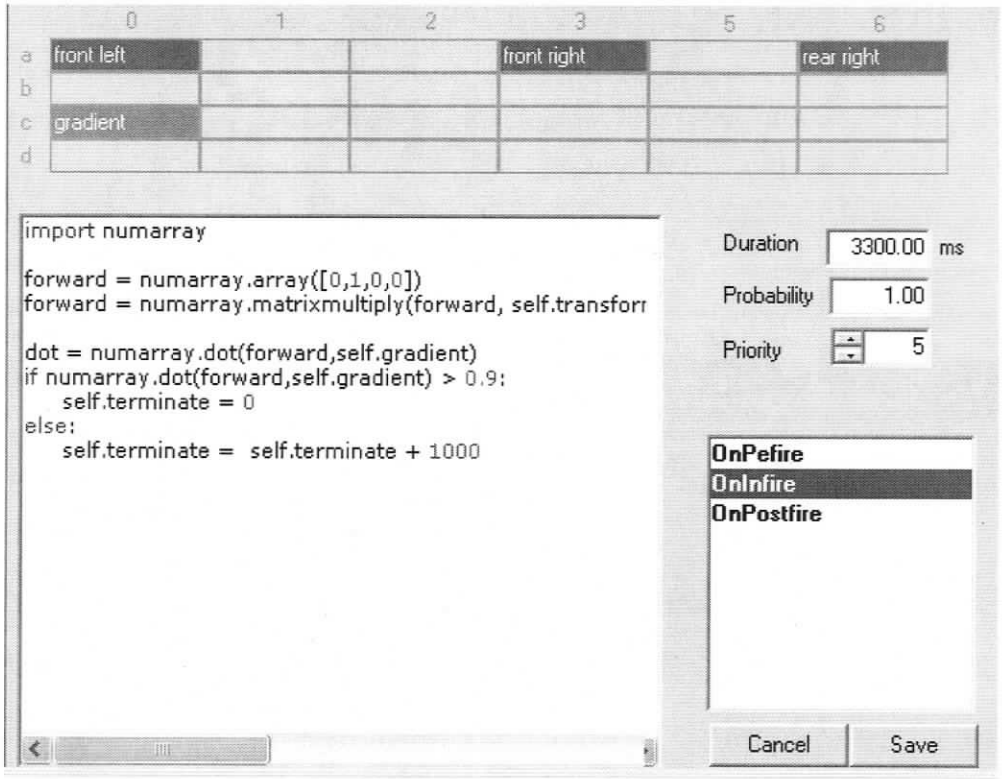

Figure 5.2: Contents View

Users define the type of a slot by dragging and dropping a node from the *Tree View* onto the grid, or by selecting a type via a popup menu. Currently, $3D^{os}$ supports *Node Tokens* that contain Scene Tree nodes, *Control Tokens* that represent the type-less tokens of PT-PNs and *Sensor Variables*. Node Token slots are colored dark or light blue, depending on whether they are marked or not, Control Token slots are colored turquoise and Variable slots are colored orange.

Every slot also provides members and methods to access the tokens and variables

from within the inscriptions and expressions. The letter in front of each row and the number above each column define a unique identifier for a slot. For example, *self.a0.vel1* refers to the angular velocity of the joint in slot a0. The API of the Node Tokens and Sensor Variables is a subset of the corresponding C++ classes and the interface of Control Tokens is *set* and *get* functions. The token slots of places further provide access to their content via the procedural programming construct *operator[]*. *self.a0[0].vel1*, for example, accesses the velocity field of the first joint node in a0.

The list box on the bottom right of Figure 5.2 is used to select the script which is displayed in the source code editor to the left. A transition contains pre-fire, post-fire and in-fire inscriptions and the slots of places and arcs contain different scripts depending on the type. Token slots contain *OnDeposit* and *OnRemove* inscriptions and sensor slots contain additional inscriptions depending on the type of sensor. A collision sensor, for example, contains *OnEnter*, *OnExit* and *OnSample* handlers.

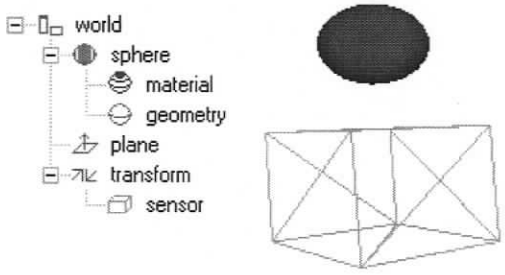

Figure 5.3: Sorting 3D model

Figure 5.3 shows the Scene Tree and 3D model of a sorting device that changes the path of resources by applying forces such as air pressure. When the sphere intersects the sensor, a vertical force is applied depending on the material properties of the sphere. The color set of a place has been marked with the sensor in slot

d0. As the ball falls downwards, it enters the area of the sensor and triggers the *OnEnter* script shown in Figure 5.4. In this case, the geometry is a sphere which has a body node as parent and a material node as sibling, pointed to by data members *parent* and *prev* respectively. The script parameter *item* contains an array of all the geometries that currently intersect with the sensor. The first element is always the one that triggered the *OnEnter* event.

```
sphere = item[0]
if sphere.prev.ambient[2] == 1.0:
    sphere.parent.setForce(0,100,0)
else:
    sphere.parent.setForce(100,0,0)
```

Figure 5.4: Sorting python script

The *OnEnter* script tests if the ambient component of the material property is blue and then applies different forces to the parent body of the geometry, effectively sorting out the blue balls. Another option is to implement the *OnSample* script and continuously apply a force while the ball is inside the sensor. An impulse force at the entry point to the sensor area may be sufficient to redirect a ball, but in order to blow it up a ramp, for example, a consistent force over a long period of time might produce better results.

One way of interpreting a place is as a processor. The slots represent I/O registers connected to communication devices and sensors, and the inscriptions are interrupt handlers which are invoked by data flow or changing telemetry. Subsystems of a Petri Net can also represent different processor states in which the arcs and transitions represent switching policies. Ultimately, there are many ways of interpreting the programming language and interface of $3D^{os}$. In general, all inscriptions are event handlers, while the Petri Net and virtual environment determine

the invocation order.

The development environment also propagates token type information throughout the graph structure of the Petri Net. The color set of a place, for example, fully determines the color set of the outgoing and incoming arcs, and partly that of the connected transitions. A token slot defines the slots of the surroundings arcs, and a variable slot defines those of the surrounding arcs as well as transitions. When a token slot is selected as part of an arc weight, then the slot of the adjacent transition is assigned the same type. The development environment automatically prevents that the markings of the surrounding places cause conflicts by assigning two different types to the same slot of a transition.

The token types are automatically propagated throughout all the constructs reachable from the initial marking. Depending on the structure of the Petri Net, token slots may contain any number of different tokens of the same type. For example, the type of slot a0 is a joint node, which generally has geometric descriptions of objects as children. However, other joint nodes with different children may also be able to reach this slot via an alternate path. The text of a slot toggles between the names of all tokens that can possibly occupy this slot. This type information can also be used to verify variable types in the Python scripts.

5.3 Discussion

The quality of a user interface is a subjective measure and it is difficult to claim that a particular system is easy to use. Some Human Computer Interaction (HCI) researchers answer this question by conducting experiments that rank an interface according to users' performance and attitude while completing tasks. Evaluation strategies along this line have already been applied to manufacturing simulation software. Yucesan et al. [62], for example, have derived an extensive quality metric in

the form of questionnaires that ask practitioners to rank features of a simulation tool. However, such usability findings can vary widely when different evaluators study the same interface. Sometimes, even when they use the same evaluation techniques [63]. The problems are likely caused by small sample sizes and the varying experience levels of subjects.

Although a claim about effective cognitive support cannot be substantiated conclusively, comparisons can be drawn to other results of HCI research. Most of the interface is based on context-sensitive 2D and 3D widgets and interaction models which were specially designed for particular editing functions [64]. Green et al. introduced a framework to evaluate the quality of visual programming languages (VPL) using a set of cognitive dimensions [59]. This framework was especially designed for no-specialists, while yet capturing a significant amount of the psychology and HCI of programming. This section evaluates $3D^{os}$ along the same cognitive dimensions.

The entire development environment is treated as one single programming language including Scene Trees, Petri Nets and Python, as well as the infrastructure provided for navigation and data manipulation. This approach stands somewhat in contrast to how Green et al. applied the framework by comparing Prograph and LabView, two diagrammatic languages, with Basic, a textual programming language. Although the authors mention the importance of an effective development environment, they rarely compare actual features of the environments themselves, but rather focus on the language constructs. The following sections first explain each cognitive dimension in the words of the authors and then discuss the advantages and disadvantages of $3D^{os}$. Although the dimensions are said to be orthogonal, a few properties of $3D^{os}$ seem to stand out in several different areas. Three out of the 13 dimensions have been omitted because they seemed too similar.

5.3.1 Abstract Gradient

> "An abstraction is a grouping of elements to be treated as one entity, whether just for convenience or to change the conceptual structure. Programming languages can be grouped as abstraction-hating, abstraction-tolerant and abstraction-hungry, based on their minimum starting level of abstraction and their readiness or desire to accept further abstraction." [59]

$3D^{os}$ supports three distinct mechanisms of abstraction; the virtual environment, the Petri Net discrete-event modelling language and the Python textual programming language. Petri Nets as well as other diagrammatic languages are inherently abstraction hating. None of the four basic language constructs can be refined any further, and the only effective means of abstraction is a page that represents a collection of places and transitions. Python on the other hand, can be classified as somewhere between abstraction-hungry and abstraction-tolerant. Textual programming languages contain flexible control abstractions such as conditionals and loops. In addition, subroutines can abstract reusable logic into a single function. Object-oriented languages, further seek abstraction in terms of classes and packages. Not surprisingly, many users are repelled by abstraction-tolerant or abstraction-hungry systems, because they tend to be harder to learn. However, abstraction is essential in structuring a system and in promoting reusability.

$3D^{os}$ basically presents users with a choice of abstraction mechanisms. For example, it is probably possible to implement the entire Factory model using the Python script of a single color slot. Of course, it is doubtful that this approach would result in a comprehensible model, because it would lack the benefits inherent to the layout and control structures of a Petri Net. Another important aspect of $3D^{os}$ is that abstractions can be layered. At the highest level is the 3D model,

followed by Petri Nets and the Python scripts. A vertical slice through the layers defines a type, which may represent a mechanical subsystem such as a work cell or machine. Although there are no guidelines of how to adequately abstract a system, it is likely that user will find the most suitable level of abstraction among the available mechanisms.

5.3.2 Closeness of Mapping

> "Programming requires mapping between a problem world and a program world. The closer the programming world is to the problem world, the easier the problem solving ought to be. Ideally, the problem entities in the user's task domain could be mapped directly onto task-specific program entities, and operations on those problem entities would likewise be mapped directly onto program operations." [59]

$3D^{os}$ especially excels in this dimension, because the mapping between a real world mechatronic system and the programming environment is for the most part one-to-one. The 3D model represents mechanical components with detailed geometric accuracy and also enforces rigid body physics. The places of the Petri Net represent different processor states and the transitions represent the flow of information. The problem of positioning the effector of the robotic arm, for example, is solved by determining appropriates torques for the motors at the individual joints. This solution directly maps to the problem domain. Assuming that the virtual environment represents the laws of physics accurately enough, then the real robotic arm should be positioned correctly with the same forces as the one in the simulation. Closeness of Mapping was an important motivation for the development of $3D^{os}$ in the first place.

5.3.3 Consistency

> "The notion of consistency or harmony in programming language design goes back a long way, but it has been difficult to define. No doubt it would be widely agreed that a language is inconsistent if (like Pascal) it allows reals and integers to be read and written, but not booleans, even through booleans are treated in the same way as integers and reals for most other purposes ... For our purposes we can take it as a particular form of guessability: when a person knows some aspects of a language, how much of the rest can be guessed?" [59]

Most diagrammatic programming languages consist of only a few well-defined constructs which inherently promote consistency. For example, it is not difficult to guess that arcs only connect transitions and places. With respect to textual programming languages, Python is certainly more consistent than Pascal; mainly because the language is much younger and more experience with respect to programming languages has influenced its designed. The level of consistency of the development environment is arguably quite high as well. For example, all 3D artifacts as well as the Petri Net constructs are Scene Tree nodes and all nodes contain fields and publish a Python interface. Given that knowledge, it is not difficult to imagine, for example, how to increase the capacity of a place when a key is pressed on the keyboard: Simply drop the place and a keyboard sensor into different color slots of another place and implement an event handler that changes the capacity field when a key is hit.

5.3.4 Error-proneness

> "There are, of course, errors and errors. It is conventional to distinguish between 'mistakes' and 'slips', where a slip is doing something you didn't

> mean to do, where you knew what to do all along but somehow did it wrong; contrast with those parts of a program design and coding that are deeply difficult, where mistakes of problem analysis are quite common. The distinction is by no means perfect but will serve." [59]

Textual programming languages generally contain a number of syntactic design features, such as parenthesis and semicolons, which cause slips to occur or make them hard to find once they have occurred. However, Python seems to have been especially designed with this problem in mind. There are next to no superfluous language constructs, and it is much less likely to create a slip in Python than, for example, in C or C++. Since a textual programming language is essential to provide a sufficient level of abstraction, Python is one of the best choices available with respect to error-proneness. In terms of Petri Net, the problem of slips seems to be nonexistent. For example, it is not possible to accidently connect two places, because the editor simply prohibits it.

An important feature still lacking in the development environment itself is the ability to recover from slips such as accidentally deleting a node. Most software tools include an *undo* feature that allows users to roll back to previous states. Unfortunately, this functionality is not yet part of $3D^{os}$ and there is an urgent need to integrate an undo stack. The development environment therefore still scores weakly in this dimension.

5.3.5 Hidden Dependencies

> "A hidden dependency, as its name suggests, is a relationship between two components such that one of them is dependent on the other, but that the dependency is not fully visible. In older text-based languages, the side-effect was a much frowned-upon form of hidden dependency,

> in which a function or subroutine surreptitiously altered the value of a global variable." [59]

Examples of hidden dependencies in procedural programming languages are not only global variables but also functions. It is not easy to know where a function is being called from and it is difficult to assess the impact of a change. However, a Petri Net inscription is generally a single function that is only invoked by a single color slot. Hidden dependencies caused by deep nesting are therefore not a problem yet, although it is conceivable that future applications demand global functions which are accessible from any script in the Petri Net. In some respect, global control structures already exists in the form of the available Python extension packages. However, in this case the hidden dependencies are external to $3D^{os}$.

The main hidden dependency in a Petri Net is caused by the token flow. It is difficult to determine where a token has come from and where it will go to when examining a single color set. In order to alleviate this problem, $3D^{os}$ propagates type information throughout the graph structure. Each color slot contains type information that allows a user to make a mental connection between a token and its path through the Petri Net. The type information can also be used to verify that object member and method calls in the Python script belong to the proper types.

There are only a few hidden dependencies in the development environment itself, and they have already led to problems. For example, it is not possible to delete a Scene Tree node that is part of a marking, because the environment does not search for the appropriate place and implicitly remove the node. A node first has to be removed from the marking before it can be deleted. A few users have already complained about not being clear why it was impossible to delete certain nodes or sub-trees.

5.3.6 Progressive Evaluation

> "It is a standard finding in many domains that novices need to evaluate their own problem-solving progress at frequent intervals. The possibility of 'progressive evaluation' is downright essential for novices. Experts can usually live without it if they have to, but even they prefer to have it: for instance, experts tend to run their programs more frequently while debugging than novices do. Consequently, the programming environment needs to support 'progressive evaluation' of partially finished programs." [59]

Part of the design goal was to make the transition between editing a model and executing the simulation as simple as possible. For example, a single button click advances time and causes the Petri Net and dynamics engine to compute. At the same time, any active manipulator is automatically removed form the scene and most editing functions are disabled. Another button resets the simulation back to the edit mode, re-attaches the manipulator and enables the editing functions. Efficient caching mechanisms limit the resources required to reconstruct the initial Scene Tree and resetting even large scenes requires little time. The environment also supports toggling time off and on and setting breakpoints in the Petri Net, similar to the features supported by a debugger for textual programming languages.

$3D^{os}$ also supports a modular design and evaluation philosophy by providing infrastructure to combine a Petri Net, Python Scripts and a Scene Tree into a single persistent object. Object libraries can be developed in parallel and stored in repositories for reuse. Although the necessary functionality has not yet been fully developed and tested, this feature has been taken into consideration from the very beginning.

5.3.7 Role Expressiveness

> "In the folklore of programming it is widely agreed that some programming languages are hard to read; unfortunately, which ones are hard is not so widely agreed. The dimension of role-expressiveness is intended to describe how easy it is to the question 'what is this bit for?' There have been very few studies on comparative comprehension of equivalent programs expressed in different notations, except for cases where the cause of difficulty was clearly the presence of *hard mental operations*, poor *secondary notation*, etc." [59]

Textual programming languages allow users to determine layout of the source code and define new lexemes for function names and constant. It can therefore become difficult for others to identify a scope or the purpose of a name. Part of the design considerations of Python was to address this problem by forcing users to denote different scopes through consistent indentation. The Python editor in $3D^{os}$ further promotes comprehensibility through syntax highlighting, in which language specific keywords are clearly marked using differently colored text. Petri Nets on the other hand, naturally promote Role-expressiveness, because the language consists of only four differently colored and shaped icons. Users can further improve comprehensibility through adequate naming and layout. However, similar to the problem with textual programming languages, inadequate layout or names can confuse rather than clarify.

The development environment further supports Role-Expressiveness via the features of the windows graphical user interface development toolkit. For example, all Scene Tree nodes are labelled by differently colored icons that identify individual nodes and group them into categories such as geometric, material, dynamics, etc. The toolbar buttons also contains icons and display informative text when the mouse

hovers over a button. The development environment also displays only a minimal number of constructs at a time. For example, the buttons related to editing a Petri Net are not visible when the *World View* is active. Many other tools usually disable buttons but keep them visible in the toolbar. Presenting a choice which cannot really be made seems to be one of the most common absurdities in desktop application interfaces.

5.3.8 Secondary Notion and Escape from Formalism

> "Many programming languages allow extra information to be carried by other means than their formal syntax; indenting, commenting, choice of naming conventions, choice of programming construct, and grouping of related statements. There techniques have no place in the formal semantics of the algorithm, but they all convey meaning to the human reader." [59]

The case study in Chapter 7 illustrates how a Petri Net that controls a manufacturing process can also resemble the physical layout of the system. This is an excellent example of Escape-from-formalism, where the whole Petri Net conveys more information than the sum of its individual parts. In the presence of a 3D model it becomes almost natural to abstract a control system in terms of a 2D projection of the mechanical components. Even textual programs can convey additional information through consistent variable names or by grouping related blocks of source code. Yet, a developer has to make a conscious effort to choose a format and it may not always be clear to others why the source code has a certain layout.

5.3.9 Viscosity

> "The *viscosity* of a fluid is its resistance to local change. We apply it to programming languages (and other information structures) to mean how

> much work the user has to put in, to effect a small change. Obviously this depends on the precise change, but in general, some programming languages/environments need more work. One standard example of viscosity is having to make a global change by hand because the environment contains no global update tools. A 'smart' environment reduces viscosity be introducing abstractions." [59]

Viscosity is partly a property of the development environment as a whole and partly of the underlying modelling languages. It is difficult to design for viscosity from the beginning, because it is hard to know exactly how a tool will be used and where *resistance* will occur. Improving viscosity is an ongoing process that generally thrives to reduce the time required to complete a task based on experiences collected from using a tool. Repetition seems to be the most contributing factor to low viscosity and is also the easiest to alleviate. A repetitive task can often be automated to the point that the time required to complete the task decreases by an order of magnitude.

For example, in the message box that is used to import a 3D Studio Max or VRML file it was until recently impossible to select more than one file at a time. Now, multiple files can be imported at once and the repetitive task of opening the message box for each file individually has disappeared. Another example is the mechanism used to compile scripts. Although there is a compile button next to the edit window, the scripts are also automatically compiled whenever the edit window looses focus. It is easy to make the mistake of changing a script and then quickly switch back to the 3D view in order to observe the effect without having compiled first. The time required to switch back and forth is therefore saved by compiling implicitly whenever the edit window looses focus. On the other hand, it is still not possible to select and manipulate multiple Scene Tree nodes or Petri Net constructs

at once. Operations such as moving or deleting nodes therefore have to be performed one construct at a time.

With respect to the programming languages, viscosity is quite different for Petri Nets and Python. In order to insert a new Petri Net construct, several others may have to be moved or reconnected. Especially since the inscriptions are lost when an arc is deleted, it can become time consuming and repetitive to recreate artifacts which had to be deleted in order to facilitate the addition of a place of transition. Viscosity with respect to Python, just as any other textual programming language, is quite high. It is relatively easy to move blocks of code, insert new lines or rename variables.

5.3.10 Visibility and Juxtaposition

> "The visibility dimension denotes simply whether required material is accessible *without cognitive work*: whether it is or can readily be made visible, whether it can readily be accessed in order to make it visible, or whether it can readily be identified in order to be accessed. Long or intricate search trails make poor visibility. (Contrast with hidden dependencies. Visibility measures the number of steps needed to make a given item visible, hidden dependencies describe whether relationships are manifest.) Another important component is juxtaposability, the ability to see any two portions of the program on screen side-by-side at the same time." [59]

For reasons similar to why $3D^{os}$ excels at One-to-one mapping, it also excels at Visibility and Juxtaposition. Due to the close relationship between real world artifacts and the programming environment, the number of steps required to recognize and find an artifact is small and intuitive. The three distinct layers of abstraction

clearly separate different components in the systems and a top down search will never require more than a few steps in order to reach even the most hidden artifact.

An example of good Visibility is how to find a script in the color slot of a place belonging to a particular work cell of the model in Chapter 7. Searching the virtual environment can quickly lead to a desired cell, even when the search space is large. If a cell is not immediately visible then 3D landmarks will generally guide a user to the cell when navigating through the environment using the mouse and keyboard. Double-clicking on any object within the cell will focus the editor on that cell. After switching to the Petri Net view, it should then be possible to identify the place that contains the script. Note that the places and transitions are named, and users can design a graph layout that helps to identify subsystems quickly. After a place has been selected, a single mouse click displays the color sets. The layout of the grid and the color coding also convey contextual information that should help to select the proper slot and script.

The development environment also includes many implicit features that help navigate and edit different artifacts. For example, clicking on a node in the Tree View automatically attaches a manipulator to the corresponding 3D artifact and populates the Object Inspector with the fields of the node. The environment therefore implicitly presents visibility to all the aspects of a node without any additional user interactions. Furthermore, the field names in the Object Inspector are the same as the members of the corresponding Python class. The Object Inspector therefore acts as an interface specification as well as a mechanism to manually edit fields. The only artifacts still missing are the methods of a node.

There are still drawbacks with respect to Visibility and Juxtaposition, especially when viewing the content of a Petri Net construct. For example, it is not possible to view all the scripts inside a place side by side. It is also not possible to cut-and-paste blocks of source code between scripts, which can be an effective mechanism when

developing similar logic. It would also be useful to display all of the scripts that apply to an artifact along a path through the Petri Net. For example, show all the scripts that control the flow of boxes through the factory model.

5.4 Summary

This chapter introduced the $3D^{os}$ User interface, which is an essential component for constructing simulations using Geometry-driven Petri Nets. The interface was outlined in a format that should provide a working knowledge for new users. However, the system contains many features which are best understood by executing the development environment. The user interface, including the underlying modelling languages, was also evaluated in terms of cognitive dimensions that qualify general ease-of-use and comprehensibility.

Chapter 6

Software Architecture

"For all but the most trivial software systems, success will be elusive if you fail to pay careful attention to its architecture: the way that the system is decomposed into constituent parts and the ways those part interact." [65]

This chapter illustrates the $3D^{os}$ software architecture using a semi-formal description of *system views* as proposed by Clements et al. [65]. Each view documents the organization of a different dimension including modules, data structures and processes. $3D^{os}$ consists of approximately 200 classes and 40,000 lines of source code as well as a number of additional third-party libraries. This chapter contains a set of high-level diagrams designed to document the most important architectural artifacts. More detailed descriptions such as inheritance relationships and call sequences can be extracted from the existing source code via a software development environment or by reverse engineering diagrams using tools such as Rational Rose [66].

6.1 Module View

The source code and libraries are organized into different modules which define coherent units of functionality. Table 6.1 provides a short description of the custom-built, internal modules of $3D^{os}$ and Table 6.2 describes the third party modules which have been integrated. Appendix A further contains a complete listing of the classes in each internal package including inheritance relationships.

Table 6.1: $3D^{os}$ modules

Name	*Function*
Geo	Mathematical artifacts such as vectors, matrices, quaternions, planes and lines
Scn	Basic Scene Tree node types including actions and fields
Man	3D Manipulators including the interactive parts
Cmd	Petri Net runtime kernel
Win	Petri Net editor
Rtn	A type system that maintains instances of Petri Nets
App	The application frame and all of the user interface widgets

Every class name in an internal module begins with a three-letter identifier. For example, *Scn* stands for scene and *Rtn* for runtime. Initially, the purpose of the identifiers was to characterize the function of a package, but throughout the software evolution process some have lost their meaning. However, the identifiers still categorize the system and clearly classify a type when examining the source code.

Table 6.2: 3rd party modules

Name	*Function*
Python	A scripting language including runtime interpreter
ODE	A physics engine that computes aspects of rigid body physics in real time
OpenGL	An industry standard 2D and 3D graphics specification language
Windows	The Microsoft operating systems and all of its functionality

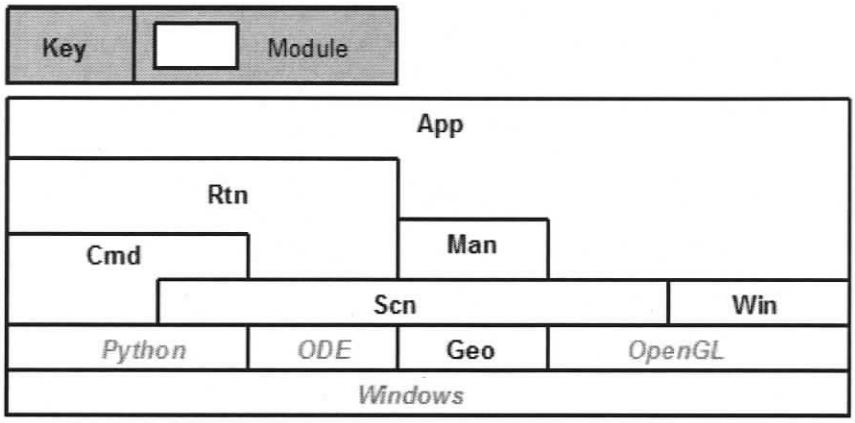

Figure 6.1: Module decomposition

Figure 6.1 depicts a hierarchical module decomposition diagram. An important design goal was to avoid cyclic dependencies among modules and thereby enforce a strictly hierarchical decomposition. Hierarchies naturally enforce separation of concerns and lead to incremental refinement. The higher up in the diagram a module appears, the more specialized it is. For example, module *App* contains the application frame and much of the user interface. This module is not even a library, but the final executable. Clearly, *App* is the most specialized, and cannot not be reused in another application. However, modules such as *Geo* or *Scn* could easily be re-used.

Most modules are aggregations of other modules and either publish or hide the interfaces of their sub-modules. Figure 6.2 describes the scope of the interfaces throughout the module layers. Almost every module is based on *Scn*, because most objects are derivatives of Scene Tree nodes. Many of the mechanisms for data persistence and organization, for example, are included in the *Scn* package. If an object has to be written to file then it must externalize the appropriate functionality of its *Scn* component; in most cases its base class. Every extension of *Scn* therefore externalizes the interface of *Scn* in Figure 6.2.

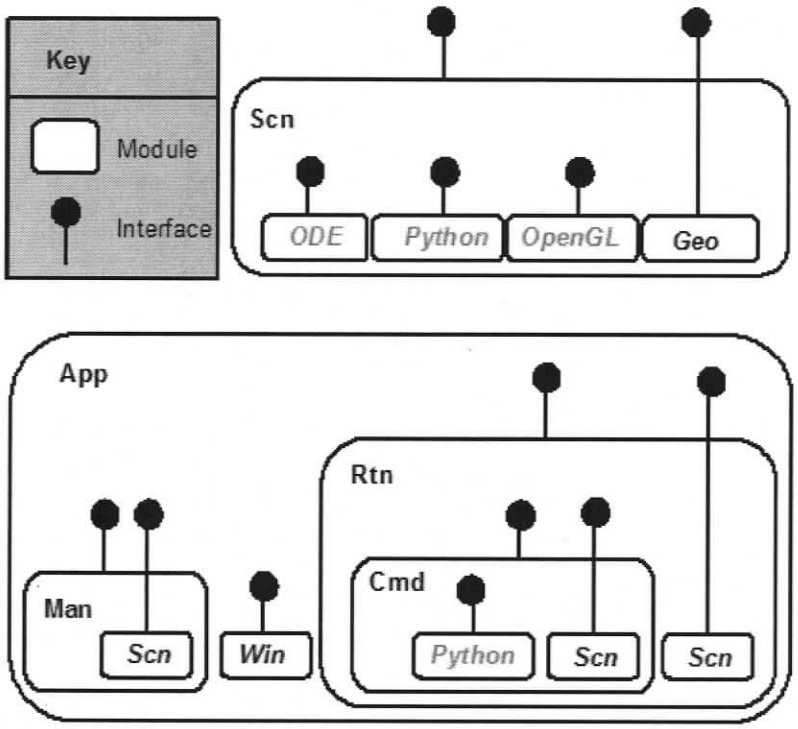

Figure 6.2: Module interface visibility

6.2 Data View

The data view documents the organization of information during runtime and conveys ownership relationships and update responsibilities. Mostly Scene Tree nodes are instantiated when a simulation is read from a file. Although the nodes are organized into a tree structure rooted at a single node, the ownership and update responsibilities are not always parent-child relationships. Figure 6.3 depicts how the internal node repository is organized and conveys which components are responsible for updating it.

The Petri Net view and the Content views for places, transitions and arcs man-

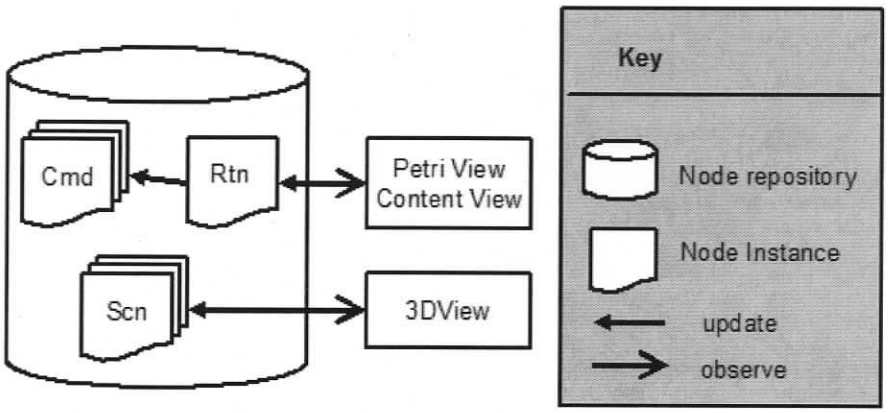

Figure 6.3: $3D^{os}$ Data view

age the *Rtn* node types, and the 3D view manages the graphical, *Scn*, node types. The update relationship between the views and the repository is bi-directional. When a simulation is running, then the views reflect the internal state of the repository. While editing, the roles are reversed and the views are used to change the internal state.

The *Rtn* module implements a type system for Petri Net constructs. A class in the *Rtn* module represents a type and the corresponding class in the *Cmd* module represents an instance of that type. For example, a simulation object such as the painting station in Chapter 7 consists of a Scene Tree containing a combination of *Rtn* and *Scn* classes. An object is instantiated by copying the tree and replacing the *Rtn* classes with the corresponding *Cmd* classes. The instances share type specific information such as Python scripts or Petri Net graph connectivity. Every *Rtn* class also maintains a reference to each instance and propagates editing changes.

Figure 6.4 shows the relationship between Scene Tree nodes and fields. Every field encapsulates a data type including knowledge about how to read and write content from a file and how to synchronize with the Python interpreter. Besides

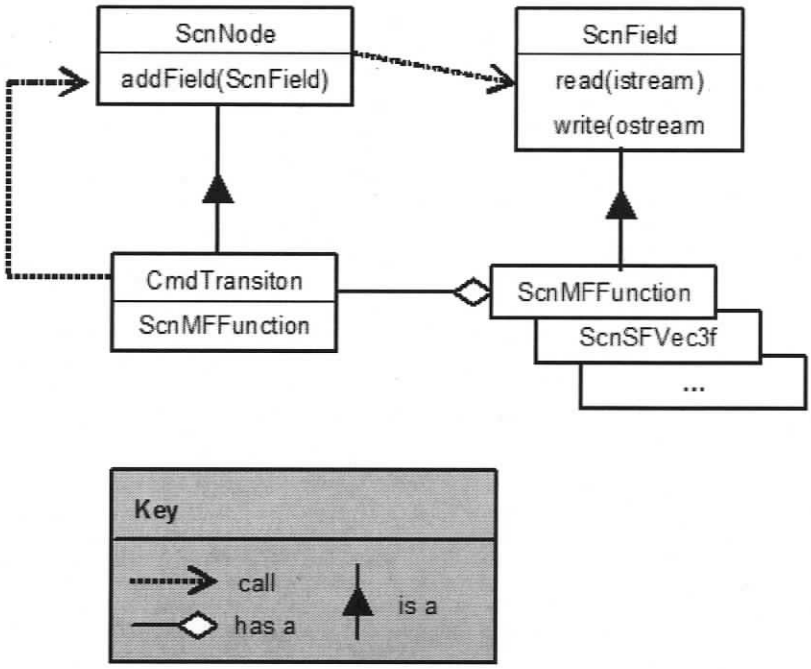

Figure 6.4: Fields

basic data types such as integer, boolean and float, fields also represent abstract types such as vectors, arrays and Python scripts. The concept of *fields* has been adopted from Open Inventor and some of the class names still reflect this fact. For example, the substring *SF* in a field name stands for *single field* and *MF* stands for *multi field*. A single field contains only one value of a type and a multi field contains an array.

Every node consists of a collection of fields and a set of methods. The base class, *ScnNode*, maintains a list of fields and every derived class registers its own fields with *ScnNode* using the *addField* function. After a field has been registered, it becomes part of the read and write mechanism. Unless new field types are required, new nodes can be developed without having to address persistence or data exchange

with Python. Several different artifacts of $3D^{os}$ reflect the concept of fields. For example, for each type of field there also exists a separate user interface widget to display and enter data. The interface for a node can be constructed simply by parsing the list of fields stored in the base class. New node types are therefore automatically supported by the interface without developing additional widgets.

6.3 Process View

The process view describes the interactions between modules during runtime using function call sequences. This view is closely related to the order of statements in the source code, and much runtime behavior can be obtained from reading the code. However, this section describes selected pieces of process logic using time sequence charts, which emphasize only the most important control flow and omit much of the additional detail contained in the source code.

Figure 6.5 shows the sequence of function calls that applies to a Scene Tree in response to events such as mouse and keyboard input or whenever time advances. All events in $3D^{os}$ lead to the creation of an *action* object, which is propagated throughout the tree structure in an in-order traversal via the *handle* function of each node. A node can update the state information within the action object in order to communicate with its siblings, children and parents. For example, if a user clicks on a object in the scene, the event traverses the tree until the selected geometry terminates the traversal.

Actions also visit the Petri Net constructs in the Scene Tree. Arcs, transitions and places often update their internal state during event traversals. For example, in-occurrence inscriptions are called during time events and arc expression may be evaluated during mouse-click and keyboard events. Using the same mechanisms to update the virtual environment and Petri Net reduces the amount of source code in

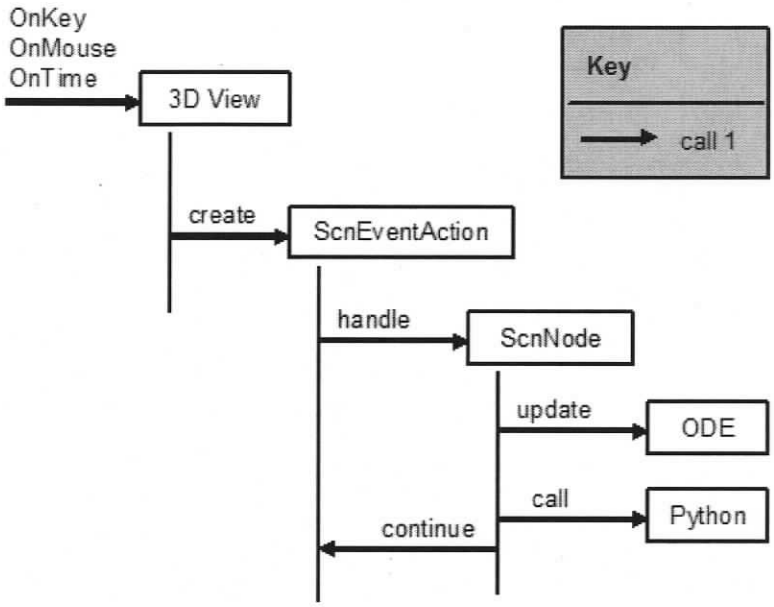

Figure 6.5: Event call sequence

the system and subsequently reduces its' complexity. Also, the Petri Net and virtual environment are both synchronized by the frame rate of the simulation.

Figure 6.6 shows the function call sequence of the rendering process. Every time the simulation advances or whenever the operating system has to repaint the canvas, a *render* action is initiated. The handlers of this action call different OpenGL functions which apply the effect of a node on the rendering state. For example, a triangle meshes calls *glVertexPointer*, *glNormalPointer* and *glDrawElements*, and a material nodes call *glMaterial*.

Before rendering any objects, the light sources and cameras have to updated, because their parameters have to remain static during a rendering step. For example, if a light source is attached to a moving body then the position of the light only

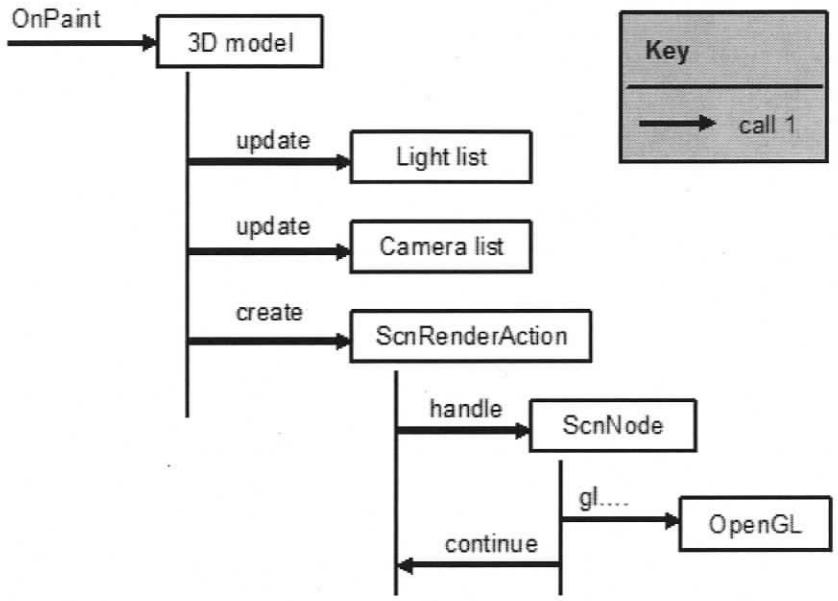

Figure 6.6: Paint call sequence

changes when the *render* action reaches the light node. Every geometry before that point would be rendered with the light at the previous position, and everything afterwards with the light at the new position. Obviously, this causes unrealistic lighting and leads to flickering. All lights are therefore updated before the rendering step begins.

Figure 6.7 shows the call sequence that loads a simulation from file. This process starts in the main window of the application after a file has been selected from a dialog box. First, the file is read and a Scene Tree is constructed using the mechanisms of the *Scn* package. The initial tree contains a type, which is instantiated by copying the tree. The first action applied is the *create* action, which performs different initialization functions such as compiling python scripts or computing bounding volumes. Following is the *mark* action which populates the Petri Net with tokens

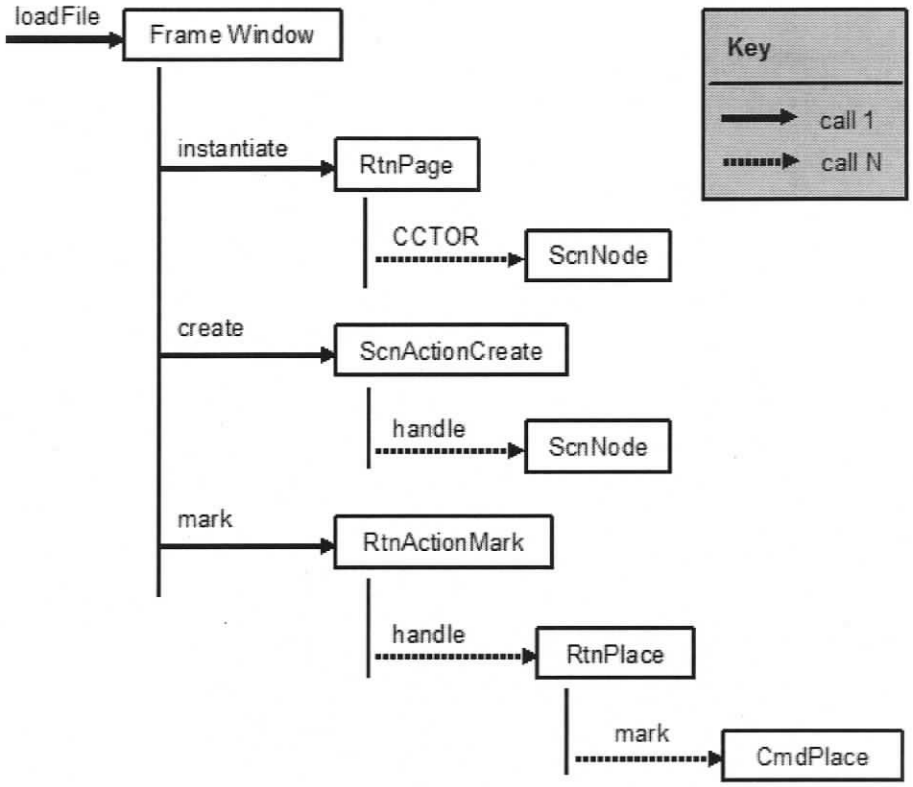

Figure 6.7: Load call sequence

and leads to evaluating the arc expressions for the first time.

6.4 Extensibility

In order to promote the adoption of a tool like $3D^{os}$, it must be extensible for third party developers. The application architecture has been designed with that requirement in mind. Although the Python programming language already provides great flexibility, certain users will likely find it necessary to create new node types in C++. This section outlines how to define a new type including the Python bindings

and action handlers. Every new class has the same functionality as a native node that can be instantiated, used as a token and accessed via Python scripts.

The following source code illustrates how to create a new node from an already existing base class and a set of predefined macros. This kind of extension framework is typical for C++ based software system. However, more flexible extension mechanisms are generally based on C++ templates. In the future, $3D^{os}$ may well be converted into a template library and a different, arguably less complicated extension mechanism will replace the one documented here. However, it is difficult to design and debug a template library and it is probably best developed after most of the classes and functions in the system have matured.

```
class ScnText : public ScnGeometry
{
    SCN_NODE_HEADER(ScnText)

    public:

        ScnText(ScnText& iText);
        ~ScnText();

        void handle(ScnActionRender& iAction);

        ScnSFVec3f mText;

    public:

        typedef struct : public ScnGeometry::PyStruct
        {
            PyObject* mText;
        } PyStruct;

        PYT_NODE_HEADER(0,0,1);
};
```

Figure 6.8: ScnText definition

Figure 6.8 displays the declaration of class ScnText, which is a subclass of Sc-

nGeometry. The macro *SCN_NODE _HEADER* at the top of the class defines standard members and methods responsible for type management and instantiation. Following are the constructor and the render action handler. As soon as a text node is inserted into the Scene Tree, the handler is automatically called whenever the text has to be drawn. The next members define the Python language bindings which determine the API available from within a Python script. A structure called PyStuct contains the data members of the corresponding Python class and macro *PYT_NODE_HEADER* defines three arrays containing the member names, methods and set-get functions. The three parameters of the macro denote the sizes of the corresponding arrays.

Figure 6.9 displays the definition of class ScnText. The constructor first initializes the base class and the text field member variable, and then instantiates a Python proxy class for ScnText using macro *PYT_NODE_CTOR*. The two lines of code following the macro assign the proxy class of the text string, which was instantiated in the constructor of ScnSFString, to the member variable in PyStruct. Referencing the class member *text* from within a Python script will therefore access the content of the field mText in the C++ class. Following the constructor is the render action handler, which first calls the corresponding base class method and then uses OpenGL calls to render the text. At the time when this function is called, the OpenGL context has already been initialized and the parent nodes have already been rendered as well.

Macro *SCN_NODE_SOURCE* defines the members which are declared by macro *SCN_NODE_HEADER*. Similarly, *PYT_NODE_SOURCE* defines the members declared in *PYT_NODE_HEADER*. The function initClass is called once to initialize static class members. The macros *SCN_NODE_INIT* and *PYT_NODE_INIT* initialize type information and register the new Python class with the interpreter. Following are three arrays that initialize the Python class with C++ function point-

```
ScnText::ScnText(ScnText& iText) : ScnGeometry(iText)
, mText()
{
    addField(mText);

    PYT_NODE_CTOR
    PyStruct* lStruct = (PyStruct*)mPyObject;
    lStruct->mText = mText.mPyObject;
}

void ScnText::handle (ScnActionRender& iAction)
{
    ScnGeometry::handle(iAction);

    PyStruct* lObject = (PyStruct*) mPyObject;
    if (lObject->mVisible)
    {
        glPushAttrib(GL_LIST_BIT);
        glListBase(ScnGlobal::sFont);
        glCallLists(strlen(mText),GL_UNSIGNED_BYTE, mText);
    }
}

SCN_NODE_SOURCE(ScnText)
PYT_NODE_SOURCE(ScnText)

void ScnText::initClass()
{
    SCN_NODE_INIT(ScnText, ScnGeometry)
    PYT_NODE_INIT(ScnText, ScnGeometry, "Text")
}

PyMemberDef ScnText::sPyMembers[] =
{
    {"text", T_OBJECT, offsetof(PyStruct, mText), 0, ""}, {NULL}
}
PyMethodDef ScnText::sPyMethods[] = { {NULL} }
PyGetSetDef ScnText::sPyGetSeters[] = { {NULL} }

PYT_GENERIC_INIT(ScnText, ScnGeometry)
PYT_GENERIC_FREE(ScnText, ScnGeometry)
```

Figure 6.9: ScnText declaration

ers and the memory offsets of member variables. For example, whenever the member variable *text* is referenced form within a Python script, the interpreter references the memory location defined in array sPyMembers. In this case, ScnText has only one member and no methods or set-get functions. The macros $PYT\_GENERIC\_INIT$ and $PYT\_GENERIC\_FREE$ define a generic constructor and destructor of the Python class. In some cases the macros may be replaced by user defined functions. Note that it is also possible to create a class without a Python binding by emitting all of the PYT macros.

6.5 Summary

This chapter introduced the $3D^{os}$ software system using selected architectural views and outlined an extension mechanism. This documentation provides an entry point for software developers who want to improve the application and add new components. However, gaining proficiency as a developer requires studying the source code and exploring the running application. Just like anything else, practice is the best tool for becoming an expert.

Chapter 7

Case Studies

This chapter demonstrates how to apply Geometry-driven Petri Nets and simulate a material handling system designed to paints boxes. This particular model was chosen because it displays many characteristics of an industrial manufacturing system [67], and because it is complex enough to warrant the use of hierarchical Petri Nets. The manufacturing process requires different concurrent tasks in order to control numerous devices and track the flow of resources. The purpose of this model is to demonstrate and validate Geometry-driven Petri Nets and $3D^{os}$ in a similar manner to how the *Stanford Bunny* is frequently used to evaluate computer graphics techniques such as tessellation, collision detection or shape deformation algorithms [68].

The 3D model shown in Figure 7.1 is part of a virtual reality project at the University of Munich. The original version is specified in the Virtual Reality Modelling Language (VRML) and animated using key-frame animation sequences. This chapter shows how to use Geometry-driven Petri Nets and rigid body dynamics to develop a similar model. The parameters of the simulation now include mechanical joints, masses and coefficients of friction, and the animations are controlled by applying forces instead of animation sequences. During the process of importing the

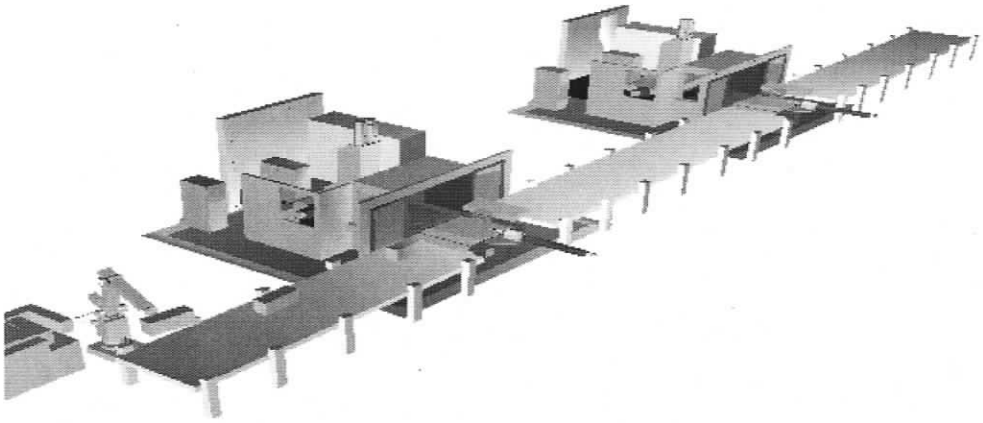

Figure 7.1: 3D model of material handling system

model into $3D^{os}$, the key-frame animations were removed and a few new mechanical components were added.

The system contains two painting stations, three conveyors and a robotic arm. The arm continuously places boxes onto the first conveyor, which transports them to the first painting stations. If the station is empty, then a box is inserted. Otherwise, it continues on to the next station via the second conveyor. A pneumatic rod in front of each station pushes boxes inwards at the same time as the doors open. After a fixed amount of time inside, a box changes color to indicate that the painting step is complete. Then, it is extracted and placed back onto the conveyor. All boxes exit the production line at the end of the third conveyor.

The challenges in this system are to avoid collisions with devices and to ensure that boxes are properly handled. A conveyor belt, for example, may have to stop in order to prevent boxes from colliding and piling up. The doors to the station must also open fast enough to make room for the entire width of a box, and while the pneumatic rod is pushing or pulling no other boxes can cross its path. Also, boxes

which have already been painted must be identified and should not be painted again.

A hierarchical Petri Net was constructed to model the control system using tokens to represent resources and devices, and variables to represent sensors. Most of the transitions either apply forces or rearrange the Scene Tree. The following sections show how to construct selected parts of the systems, including the Robotic Arm, the Painting Station and the Production Line.

7.1 Robotic Arm

Figure 7.2 displays the robotic arm and the Petri Net that controls its function. Three angular joints had to be added to the original model in order to define the proper kinematic system. The effector of the arm is positioned by changing the angular velocities around the joint axes and by specifying movement limits. Since the arm was not designed to pickup boxes using frictional forces, the solution was to create a temporary fixed joint between the box and the effector. The joint is created when the effector is next to the box during the pickup step, and destroyed when the box is positioned above the conveyor during delivery. The same approach was also used to attach the boxes to the pneumatic rod during insert and release operations.

The right side of Figure 7.2 displays the Petri Net responsible for controlling the robotic arm. Place P1 initially contains three tokens that represent the different joints of the arm, and place *buffer* contains a fixed number of boxes. As capacity becomes available, the boxes are moved one by one onto place P2. When transition *pickup* fires, it deposits the joints onto place P2 and applies rotational forces that position the arm above the pickup location. When transition *deliver* fires, it attaches the box to the effector and then rotates the arm over top of the conveyor. Finally, transition *drop* lowers the arm and places the box onto the conveyor. Afterwards, the tokens containing the arm are back on place P1 and the cycle begins again.

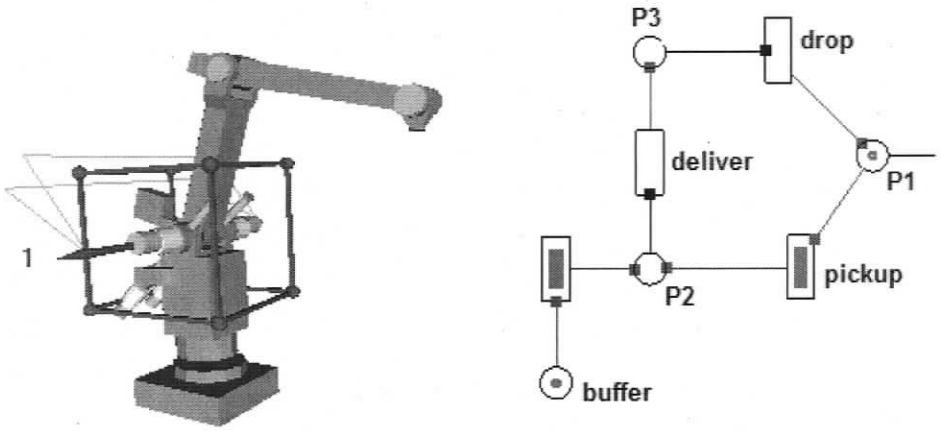

Figure 7.2: 3D model and Petri Net of robotic arm

Figure 7.3 displays the color sets of transitions *deliver* and *drop*. Each consumes and produces the tokens representing the robotic arm and a box. The color slot d0 contains the box tokens and displays "..." in order to indicate that possibly more than one different type of token can traverse this path through the Petri Net.

	0	1	2	3
a				
b	vertical	horizontal	effector	
c				
d	....			

Figure 7.3: Color set of transition *deliver*

Table 7.1 shows the inscriptions of transitions *deliver* and *drop*. The first line of the pre-fire inscription of transition *deliver* sets the velocity of the rotational joint and the second line attaches the box in d0 to the effector in slot b2. The scripts

of transition *drop* perform the reverse operation. The function *attach* creates a fixed joint between the two bodies and the function *detach* destroys the joint. None of the other scripts in the entire model are any more complex than the examples shown here. None contain loops or conditional statements and all complex logic is expressed by the structure of the Petri Nets and the token flow.

Table 7.1: Selected occurrence inscriptions

transition	***pre-fire***	***post-fire***
$T_{deliver}$	self.b0.vel1 = 2.0 self.b2.attach(self.d0)	
T_{drop}	self.b1.vel1 = -2.0 self.b2.vel1 = 2.0	self.b2.detach(self.d0)

7.2 Painting Station

Separating the boxes in front of a painting station is far more challenging in a dynamics animated simulation than in the original key-frame animated one. Using rigid body physics the boxes often move unpredictably and they can accidentally collide or slide. As a result, they are no longer evenly spaced or aligned with the conveyor, and it becomes more difficult to separate them and guide their movements. Mechanical stops that prevent other boxes from entering the platform during the insertion and extraction steps had to be designed and added to the original model. A *left stop* and *right stop* block the path to the left and right of a platform as shown in Figure 7.4. When a stop is lowered, it is part of the conveyor and boxes can pass over it. When a stop is raised, it blocks the path of the conveyor and boxes accumulate in front of it.

After some experimentation it became clear that the stops have to be slanted so that any box resting on top of the left stop after it has been raised would slide

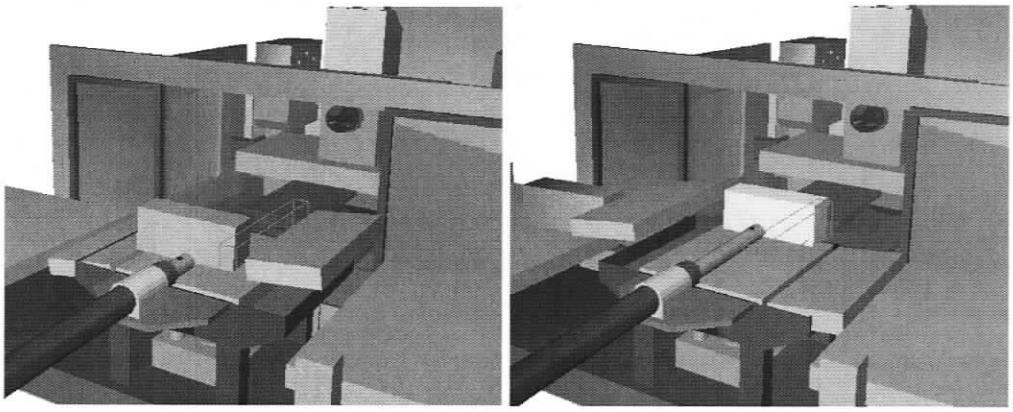

Figure 7.4: 3D model of painting station

backwards and the ones resting on top of the right stop would slide forwards. The platform also contains a *collision sensor* shown as a transparent red box to the left of the *right stop*. The sensor is triggered while an object occupies space within the rectangular shape.

When a work cell is ready to accept a box, the right stop is raised and the next box which moves onto the platform stops in front of the pneumatic rod and triggers the sensor. The *left stop* is then raised and other boxes are blocked from entering the platform. Next, the doors open and the pneumatic rod pushes the box inside. The extraction process involves raising the stops in the reverse order. Any box that was on top of the *right stop* when it was raised, will have either slid forward or will have been pushed forward by the box following it. Similarly, a box residing on the left stop will have slid backwards.

The Petri Net of the painting station is shown in Figure 7.5. The place P2 contains five node tokes representing the stops, doors and the pneumatic rod. The tokens always move together around the outer cycle of the Petri Net. Two transitions each are responsible for the push and pull operations when inserting or removing

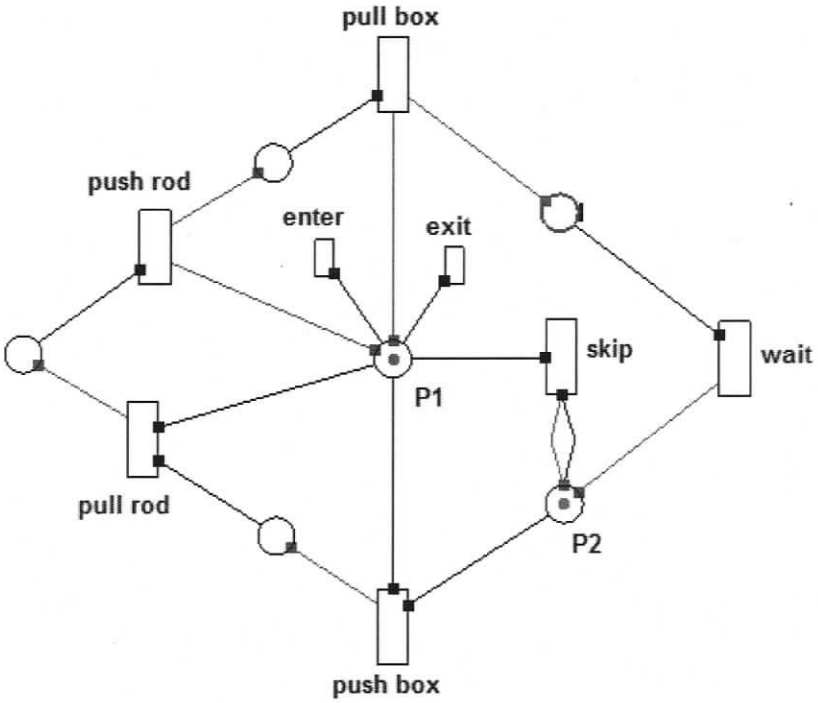

Figure 7.5: Petri Net of painting station

a box from the station. First, *push box*, opens the doors and pushes a box inside. Then, *pull rod*, closes the doors and retracts the pneumatic rod. The next two steps, *push rod* and *pull box*, are the reverse of the first and second.

The place P1 contains the collision sensor, which is part of the arc expressions of transitions *skip* and *push box*. If a box inside the sensor has already been painted, then *skip* will fire and temporarily lower the right stop to allow the box to pass. If the box has not been painted then *push box* fires and the insertion cycle begins. The transition *wait* introduces a delay between removing a box and beginning the next insertion cycle, which is necessary to allow the box that is being extracted to pass

over the right stop. P1 also contains two reference *enter* and *exit* through which box tokens are exchanged with the painting station.

7.3 Production Line

Figure 7.6 displays the Petri Net that models a production line. It contains two instantiations of the painting station named *Station A* and *Station B*. The link on the left side of each page represents the place inside the station referenced by *enter* and the one to the right represents the place referenced by *exit*. Four transitions connect the interface links to the places which model the conveyors.

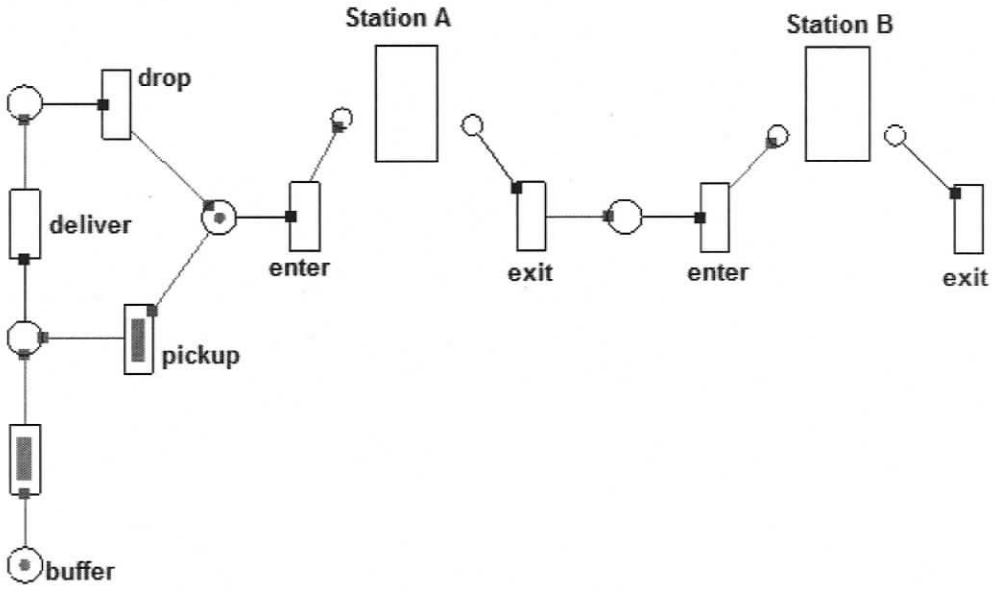

Figure 7.6: Petri Net of production line

In the keyframe-animated model all boxes originated from the same point in 3D space. However, it is physically impossible for two boxes to occupy the same location

and it became necessary to introduce a device which leads boxes to the robotic arm one at a time. Figure 7.7 shows a conveyor in front of the robotic arm which pushes boxes towards a pickup position. The conveyor is surrounded by guides that prevent the boxes from falling off.

The conveyors and platforms are modelled by surfaces which apply a directional force to all objects they collide with. After a box is dropped onto a conveyor, contact points are generated and the box begins to move into the desired direction. At the end of the production line, each box simply falls of the edge of the conveyor. The pile that builds up nicely illustrates how the simulation enforces the laws of rigid body physics.

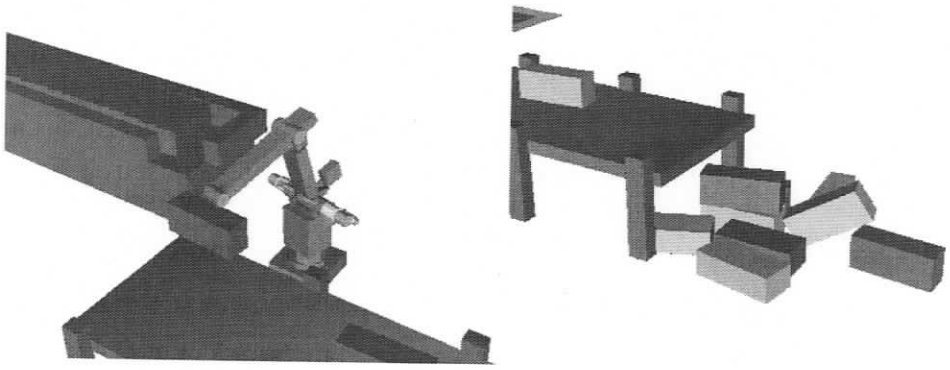

Figure 7.7: Images of production line

Without any knowledge about the physical structure of the system, it is difficult to verify that the operations can be completed correctly in the amount of time specified by the occurrence durations of the various transitions. For example, transition *push rod* may occur while a box is just sliding of the platform in front of station. Even after the left stop has been raised, there could still be a box on the platform. If transition *push rod* fires too early, or if it applies too strong of a force to

the rod, then the box may collide with the rod. Similarly, if the robotic arm moves too fast then it may pick up the next box before it had time settle into the proper pickup position. All these problem can be corrected by observing the 3D model and adjusting the firing durations of the appropriate transitions.

The firing durations are important for the cycle time of a Petri Net. Cyclic behavior is directly related to the throughput of a system, which reflects the maximum rate at which resources can be processed. Especially in complex systems, it is important to determine cycle times early on, because bottlenecks may force redesigns including expensive reconfiguration of the mechanical components.

If a Petri Net is covered by k conflict-free (see 3.2) cyclic sub-nets, then the cycle time τ_0 of the net is given by the maximum time of $(\tau_1, \tau_2, ...\tau_k)$, where τ_i is the cycle time of each sub-net, which is computed by summing the firing durations of the transitions. In the factory model, cycles can be determined informally and for more complex nets, P-Invariants can be used to determine cycles analytically [69]. Table 7.3 shows the three cycles of the model including their durations.

Table 7.2: Factory model cycles times

τ_{arm}	τ_{cell1}	τ_{cell2}
T_{pickup}	T_{push_box}	T_{push_box}
$T_{deliver}$	T_{pull_rod}	T_{pull_rod}
T_{drop}	T_{push_rod}	T_{push_rod}
	T_{pull_box}	T_{pull_box}
	T_{wait}	T_{wait}
4 seconds	15 seconds	15 seconds

The cycle of a work cell is 15 seconds and that of the robotic arm is 4 seconds. The cells have identical cycles roughly a quarter of the length of the robotic arms' cycle. Since the cells are arranged in series, it seems logical that only 2 out of every 4 boxes will be painted by this configuration and the given cycle times. This

property is in fact also observable when running the simulation. Of course, a cycle time analysis is formal and therefore more reliable. However, the analysis also relies on correct firing durations, which Geometry-driven Petri Nets naturally lead to.

Yet, there are problems with the above analysis. For example, the conveyors are modelled by places. *Box tokens* remain on the places until the dynamics engine has moved them into a sensor in front of a painting station. There is no firing duration associated with this process and the cycle time for the entire production line is therefore missing an important value. However, a cycle time analysis can also be based on running a simulation and measuring the interval between transition occurrences.

7.4 Summary

This section demonstrated how to model a mechatronic system using Geometry-driven Petri Nets. Whereas the original simulation was explicitly based on key-frame animation, this version was implicitly animated using rigid body dynamics. It seems that key-frame animation alone oversimplifies a simulation because it does not take into consideration the indeterminism of physical processes. The model introduced here therefore had to be complemented with additional mechanical artifacts.

The Petri Net configurations shown in this chapter are the first design patterns of Geometry-driven Petri Nets, which may be useful as a guide for modelling similar systems. Designing the model and deciding upon the initial marking requires an understanding of the processes involved but also demands a certain amount of intuition. Similar to most other design tasks, increased experience will better guide a developer to select appropriate patterns and to dismiss inelegant or ineffective solutions more quickly.

Chapter 8

Related Work

This dissertation covers a wide range of computational concepts and the related work is equally fragmented. Research on Petri Nets is well structured and organized around the International Conference on Theory and Application of Petri Nets. Computer graphics and virtual reality related research, however, is too diverse to categorize and it is difficult to capture the current state of the art. However, research that applies virtual reality to simulate sensor-driven mechatronic systems is confined to only a few research groups [8, 9, 10]. The following section introduces some of the more well-known and advanced related works and discusses their advantages and disadvantages with respect to Geometry-driven Petri Nets.

8.1 CPN/Tools

The most relevant tool with respect to Petri Nets is the *CPN/Tools*, which constitutes a redesign of the *Design/CPN* development environment for Colored Petri Nets. Kurt Jensen and his research group have been developing this tool for nearly a decade and over 50 person-years of labor have been invested in its design and implementation. CPN/Tools is currently used by dozens of industrial and academic organizations and has produced positive results with respect to modelling and veri-

fying many types of processing systems [70][71][72].

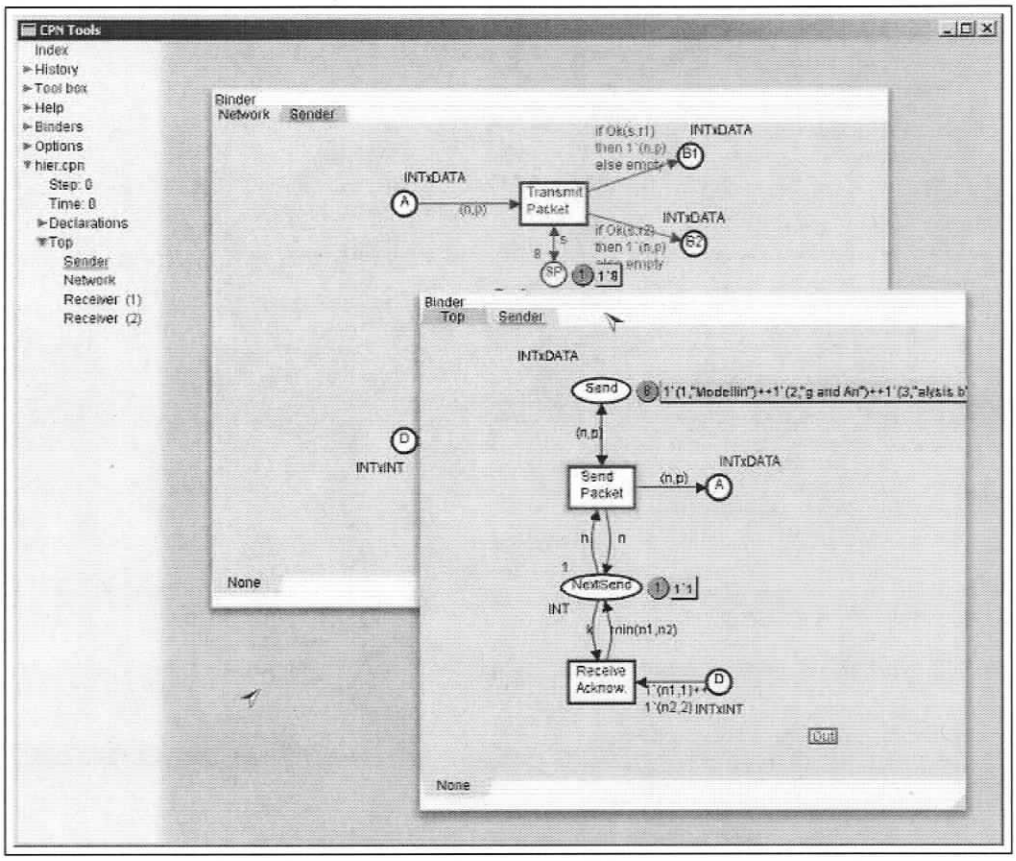

Figure 8.1: Design/CPN screenshot

Similar to $3D^{os}$, CPN/Tools also supports different abstract data types and integrates a scripting language to specify inscriptions and arc expression. The most important difference between the $3D^{os}$ and CPN/Tools is certainly the virtual reality component featured by $3D^{os}$. When using CPN/Tools, assumptions have to be made about parameters representing physical artifacts. There is no representation from which to derive meaningful values and many variables therefore have to be

randomized or approximated. However, CPN/Tools excels with built-in methods for analyzing a process model based on constructing the reachability graph. Computing the complete graph can be prohibitive, and much research has been invested in methods for optimizing this process in CPN/Tools.

The following table outlines the difference in features and functionality between $3D^{os}$ and Design/CPN.

Table 8.1: Design/CPN-$3D^{os}$ feature matrix

Function	$3D^{os}$	*CPNTools*
Inscriptions	Procedural (Python)	Functional (ML)
Hierarchies	Substitution Transitions Instantiations	Substitution Transitions Instantiations
Tokens	Scene Tree Nodes	Custom Data Types
Variables	Time Scene Tree Properties	Time Custom Data Types
Transitions	Guard Time Inscription	Guard Time Inscription
Places	Inscription (dS) Capacity	 Complement Places
Arcs	Inscription Inhibitor Test	Inscription Complement Places Complement Places
Editing	Pan / Zoom	Pan / Zoom Alignments / Layouts Views
Simulation	Marking Firing 3D Animation	Marking Firing

The user interface, especially the one for the color sets, also differs considerably between the two tools. CPN/Tools displays color sets as tuples of text, which makes them difficult to comprehend. $3D^{os}$, in contrast, uses actual colors and 2D grids

to differentiate between the differently typed tokens and variables. $3D^{os}$ can also displays 3D images of the Node Token color types. Recently, the CPN/Tools research group has begun developing a new user interface which also uses actual colors and new user interface widgets [73].

8.2 Delmia

The most widely used simulation tool for distributed mechatronic systems is the *Delmia* tool suite of the Dassault Systems company [74]. Delmia is the result of merging Deneb Robotics, SolidWorks and other companies in order to provide an integrated manufacturing simulation solution which supports the entire development cycle ranging from product design to manufacturing process control. The main components are the SolidWorks 3D modeler, the IGRIP work cell animation tool and the QUEST discrete event simulation framework. SolidWorks is a 3D modelling tool to construct geometric descriptions of mechanical components which are also used to generate CNC instructions for manufacturing devices such as lathes or drills. IGRIP is used to assemble SolidWorks models into work cells and to record the animation sequences for robotic actuator and human operators. QUEST imports IGRIP models to construct entire factory simulations by invoking the predefined animations according to a table-driven discrete event system.

The Delmia tools suite is also based on the Scene Tree data structure and includes a graphic simulation language (GSL) and command line interpreter (CLI) to define animation scripts and automate editing functions [75]. In addition, IGRIP contains predefined animation libraries consisting of device-specific kinematic models or human gesturing and walking animations. For example, a human operator is modelled by creating a series of positions while holding and assembling a product [4]. Key-frame animation then produces smooth transitions between the different

positions and poses. The Delmia tool also contains a large object library of industrial manufacturing equipment. The library is partly the result of the acceptance of SolidWorks as a design tool. The same models used to manufacture devices can also be used in a dynamic simulation of the device.

In general, the Delmia tool lacks the notion of sensors and cannot integrate feedback from the virtual environment with the logic that animates the scene. Only deterministic systems such as factories or warehouses, in which predetermined paths in space define most activities, can therefore be simulated convincingly. The tools also lack implicit animation techniques such as rigid body dynamics and autonomous character animation. Modelling human characters, for example, requires complex and laborious animation scripts [3]. The Delmia tools are therefore not useful for modelling autonomic or adaptive, sensor-driven system.

The QUEST factory simulations are based on a heterogeneous modelling language which encapsulates IGRIP models as arbitrarily complex high-level constructs with diverse functionality and behavior. It is not possible to edit IGRIP models form within the Quest application and it becomes difficult to analyze and maintain large fragmented simulations. A well-integrated homogeneous formalisms, such as hierarchical Petri Nets, on the other hand, consists of only a few formal constructs that support a wide range of mathematical analyses. Also, the language abstracts artifacts of event-driven computing rather than the functional components of a particular application domain.

8.3 Webots

Webots is a commercial 3D simulation tool for the Khepera mobile robots family. Several hundred institutions are currently using it as a teaching tool for robotics [10]. Webots includes a predefined set of sensors and actuators which are programmed

using a C/C++ API. The interface to the virtual sensors and actuators is the same as to the physical ones and a functioning control system for a Khepera robot can therefore be constructed and tested in a simulation. The Webots virtual environment also uses the Open Dynamics Engine to simulate rigid body physics. At this point, Webots is designed to support only a particular family of mobile robots and it also does not include a hybrid modelling language or the capability to model large terrains and architectural structures. However, Webots supports a user interface to construct simple environments consisting of boxes, spheres and cylinders.

8.4 Gazebo

Gazebo is a simulation toolkit developed by the Robotics Research Lab at the University of Southern California [9]. Its development has been motivated by the increasing use of robotic vehicles for outdoor applications. The toolkit therefore includes algorithms to model terrains and rigid body dynamics using the Open Dynamics Engine. Gazebo also supports sensors such as odometers and range finders, and recently it has been used to develop the control system for obstacle avoiding mobile robots.

Gazebo is an extension of the Player/Stage project, which is a client-server based toolkit to distribute multi-robot mechatronic simulations [76]. The project consists of the *Player* robotic devices server and the *Stage* multiple robot simulator. *Player* defines an API for different types of sensors and actuators, and *Stage* is a 2D simulation environment populated by robots which respond to the *Player* interface. Gazebo supports the *Player* API, but instead provides a 3D environment in which to execute simulations.

Similar to Webots, Gazebo does not include a hybrid modelling language or the ability to simulate complex architectural structures. It also does not include a user

interface to construct and navigate the environment. All models have to be specified textually using programming language statements that specify structure, location and a hierarchy of bodies and joints. In $3D^{os}$, on the other hand, vehicles can be constructed exclusively using manipulators in order to position bodies and joints.

8.5 Gamebots

Gamebots is a virtual reality toolkit based on the popular computer game *Unreal Tournament 2003*, which simulates intelligent avatars that operate in interior environments [8]. Gamebots is a modified version of Unreal Tournament that allows characters in the game to be controlled by other programs rather than human players. The engine relays sensor telemetry to client programs which decide what actions a character should take by issuing commands back to the game. Unreal Tournament supports a limited form of rigid body physics and a set of sensors. Environments are constructed of rooms, hallways, stairs and doors using 3D authoring tools design to construct large models. Gamebots is limited by the capability of Unreal Tournament, which was designed as an indoor game, rather than a general purpose mechatronics simulator. The engine supports no hybrid modelling language or user interface to construct mechatronic devices. The engine also lacks means of representing outdoor environments.

8.6 Summary

This section provided an overview of some related work with respect to Petri Nets and Mechatronics simulation. Some of the tools outlined include comprehensive user interfaces and analysis methods, but lack the means of generating realistic sensor telemetry. Others are based on interactive virtual environments, but lack the hybrid modelling language. With some limitations, $3D^{os}$ includes all the features and func-

tionally of the tools discussed in this chapter in a single development environment. Table 8.2 summarizes the differences among the related software tools.

Table 8.2: Tools feature matrix

Tool	***Sensors***	***Petri Nets***	***IDE***	***Physics***
$3D^{os}$	X	X	X	X
CPN/Tools		X	X	
Gazebo	X			X
Webots	X		X	X
Gamebots	X			

The related work is fragmented and one can argue that most virtual reality development tools including computer games engines should be investigated as related research. However, this dissertation focuses mainly on Petri Nets. In this domain the CPN/Tools is clearly the leader in with respect to colored Petri Nets, and $3D^{os}$ clearly improves upon this tool by adding a virtual reality component.

Chapter 9

Evaluation

"Research is creative work undertaken on a systematic basis in order to increase the stock of knowledge, including knowledge of humanity, culture and society, and the use of this stock of knowledge to devise new applications" – Organization for Economic Co-operation and Development."

The goal of research is to produce new knowledge. Ideally, knowledge that is general enough to apply in other contexts than the one it has been discovered in. Many times, the procedures used to validate a hypothesis are more desirable than the outcome of their initial application. There are also many forms of knowledge as well as different methods for verifying them. In the case of declarative knowledge, for example, an evaluation is based on the scientific method. For systemic knowledge, it is based on a mathematical proof and for procedural knowledge such as software, evaluation can mean a persuasive demonstration.

The simulation tool presented in this dissertation is procedural knowledge describing a software engineering method. The input to this procedure is Petri Nets and 3D models and the output is their dynamic behavior interpolated over time. The running software can be considered as an instance of the procedure and the source code of the development environment as a formal definition. This chapter presents a case for this type of evaluation and also outlines qualitative improvements with respect to other simulation methods. Before discussing the research presented here this chapter first outlines how software is evaluated by the software engineering community in general.

9.1 Background

The purpose of Computer Science is to develop the algorithms necessary for computing mathematical symbolisms. For example, the time required to determine containment of two sets is irrelevant to the correctness of a mathematical model. However, this property is important in order to utilize a processor efficiently. During the past decades, Computer Science and Software Engineering have introduced a number of data structures and algorithms designed to compute ever more complex mathematical system. Often times, under the umbrella of new terminology such as communication, operating systems or virtual reality. Mathematics, however, is still the underlying concept of programming languages. Just like a mathematical model, software always produces the same results given the same input.

9.1.1 How is Software Evaluated?

The question is how to evaluate Computer Science research? Traditionally, individual algorithms are compared in terms of running time performance or memory requirements. In software engineering, however, the focus shifts from developing efficient algorithms to constructing complete environments. At the same time, the value shifts from individual algorithms to designing large systems and effective user interfaces. For software engineers evaluation is therefore not always an easy task because it is often difficult to find appropriate similarities among related software engineering tools for a quantitative comparison. A traditional performance evaluation generally makes few useful statements, because the capability and interface of software tools is more important than their resource requirements.

The weight in the evaluation of a software engineering tool should be placed on whether a proposed system is actually functional and useable. Once it has been agreed upon that a computational method achieves a desired result, then a func-

tioning system must be the next most valuable goal. Also, once this goal has been achieved, the source code is the most complete and formal description available. Most research should therefore be conducted in the context of the source code and running software, maybe even the peer review. However, it seems that a persuasive demonstration is not readily accepted as a convincing evaluation. Even though software is a fundamentally new medium for research which naturally also requires new methods of evaluation.

It further seems difficult for researchers to peer review certain classes of software engineering research if they are exclusively presented in the form of a paper. The published results are often not easily reproduced without the accompanying software and the research is therefore difficult to verify. It seems that processors and software not only comprise a new arena of research and development but also a new media for publication. If peer review was conducted on functioning systems, then the evaluators could verify results with more confidence and vouch for correctness and completeness. This is especially true for the type of research that results in software systems such as $3D^{os}$. A functioning software system also inspires future research more vigorously.

The way in which the software engineering community tends to approach evaluation seems to underline this idea to some extent. A study that classified the papers submitted to the *International Conference on Software Engineering* in 2002 has found that a large percentage of the evaluations were exclusively based on demonstrating a software tool [11]. The same is often also the case at technology showcases or exhibitions such as CASCON and ASI. Clearly, maintaining and developing the source code must have been a priority and valued highly by the researchers. A persuasive demonstration requires a complete and correct system, including artifacts such as the user interface, which are not important in a theoretical model. In the same light, demonstrating a functioning version of $3D^{os}$ should rank highly

in the evaluation of the research presented here. The analysis along the cognitive dimensions in Section 5.3 should add further weight to the demonstration.

9.2 Discussion

In cases where software is used to simulate physical systems, the scientific method can still be readily applied for evaluation. Climatology, for example, often uses complex software to create models of the climate that scientists use to make predictions about the weather. The value of the simulation is measured by how closely the parameters of a model represent those of the physical world. In the case of climatology, the software is best evaluated by comparing its output to the actual weather. Other engineering disciplines also routinely rely on software-based simulations to conduct their research. Eventually, the simulations are always validated by comparing their output with the corresponding physical system.

A quantitative evaluation of $3D^{os}$ is similar, except that artificial systems are modelled rather than natural ones. In this case, the quality of the tool is related to how realistically it can simulate a mechatronic system. This quality is in fact quantifiable by measuring the degree at which the model deviates from its physical counterpart. The accuracy of logical as well as physical processes can be measured by using different types of control points. A physical control point, for example, can verify that boxes move at the correct velocity. A logical control point can verify that they follow the same path through the system and that they exit with the same color.

Unfortunately, this type of evaluation was not possible during the course of this dissertation, because a physical system of the appropriate scale was not available. $3D^{os}$ is designed for large-scale applications which are governed by vastly distributed control systems. It would therefore require a mechatronic system at the

scale of a factory to collect meaningful data. Yet, this type of scientific evaluation is an important area of future research and will certainly help to define areas for improvement. The following sections present qualitative reasons towards why and how the research presented in this dissertation improves upon the state of the art of Petri Nets, mechatronic system simulation and user interface design.

9.2.1 Petri Nets

Petri Nets alone lack a suitable representation from which to derive accurate values for many of the parameters that define a mechatronic control system. Instead, random variables are often used to approximate timing and sensor telemetry. By integrating virtual reality technology with the Petri Net language, many of the variables can be derived from an accurate physical representation of a system. More accurate Petri Nets than before can therefore be constructed, leading to more confidence in the results of analytical methods.

For example, in the factory model of Chapter 7, the operation that pushes a new box onto the platform in front of a cell and the one that extends the pneumatic rod are associated with the firing durations of the transition. Inappropriate durations can cause a collision between a box and the rod. Without a geometric description, it is difficult to determine if the chosen times are actually realistic. With Geometry-driven Petri Nets, a simple visual inspection of the 3D model can uncover a collision and allow designers to determine more correct durations as shown in Section 7.3. Since the firing durations do not violate physical constraints, they are more likely to be accurate and result in a more reliable cycle time analysis as well.

A side effect of representing Petri Net constructs as Scene Tree nodes is that each one also has a location in 3D space. This information provides useful parameters for constructing control logic in the inscriptions. For example, a transition which is a child of a rigid body will always have that same location as the body. Also, the

Petri Net constructs could implement the render action and become visible in the virtual environment; adding a new level of understanding through a 3D layout of the graph structure.

9.2.2 Mechatronic System Simulation

A defining characteristic of Geometry-driven Petri Nets is that sensor telemetry from the virtual environment is used to simulate the feedback loop between a control system and its surroundings. This property stands in contrast to current industrial simulation tools such as Delmia, which can only animate a 3D model according to a manufacturing process. Although a process model is still required to schedule animations, the outcome of the simulation is always the same unless random variables are introduced. In Geometry-driven Petri Nets, the variables are replaced by sensors which derive realistic values from the virtual environment.

The dynamic behavior of the virtual environment also confronts designers with problems that more closely reflect the challenges of developing a real system. This fact is illustrated by the solution for the dynamics-animated production line shown in Chapter 7, where a physically viable solution had to be found in order to insert and extract boxes using the left and right stops. More complex logic also had to be designed in order to control the additional mechanical components. Overall, the simulation reflects the final solution more closely than a corresponding system developed using Delmia. The behavior is more realistic and ultimately yields more accurate analyses.

$3D^{os}$ can also be used to model autonomous mechatronic systems such as vehicles that navigate natural terrain by continuously sensing the topology. Without the notion of sensors these types of system cannot be simulated, and pure animation tools are ineffective in this problem domain. $3D^{os}$ however, is capable of simulating both, repetitive assembly systems and autonomous transportation systems in

the same simulation environment. It would require little additional development to transport the boxes that exit the production line in Chapter 7 to another processing step using autonomous vehicles. In this case, two separate mechatronics simulation techniques would coexist in the same environment.

Yet another important advantage is using Petri Nets as a modelling tool for 3D mechatronics simulations. Section 3.2.2 illustrates how Petri Nets can express the most important concurrent programming artifacts such as concurrency and synchronization using exclusively a formal graph structure. A complex processing system can therefore be abstracted more clearly than using a procedural programming language alone or even when using State Machine or State Charts. State Machines can not express concurrency at all, while State Charts can only express concurrency but not resource flow.

9.3 Summary

This chapter has provided arguments towards the advantages of Geometry-driven Petri Nets. It was argued that integrating a virtual environment with Petri Nets is an effective mechatronic simulation method which produces more accurate Petri models that will ultimately lead to more reliable analysis. It was also argued that a persuasive demonstration of a software engineering tool should be ranked highly in the evaluation of research. Ultimately, the best measure of the quality of a software engineering tool is time. If it is used in the future then it is useful. However, in order to be used at all, it must be functioning and provide a good interface in addition to employing effective computational methodologies.

Chapter 10

Future Research

"Experiments and not conjecture are certainly the best basis to determine the performance of a system and simulation is the preferred method of experimentation." [1]

$3D^{os}$ is a general framework in which to implement and evaluate new ideas without having to reconstruct basic infrastructure. This chapter discusses several topics for future research, including new applications of Geometry-driven Petri Nets and improvements to the development environment. The range of mechatronic systems that can currently be simulated is foremost limited by the types and quality of the supported sensors. An important goal is therefore to improve the virtual reality component by modelling a larger variety of artifacts with a higher degree of detail. Another goal is to increase the cognitive support of the development environment. While the underlying theory of software engineering tools is essential to their usefulness, so is the manner in which they are presented. It is therefore always important to develop effective user interfaces in conjunction with any functional improvement.

10.1 Character Animation

Many types of manufacturing systems still incorporate human labor, while others are designed to respond to human or animal behavior. In both cases, methods are needed to simulate intelligent bio-mechanical agents. An important step in this direction is to animate skeletal systems for locomotion and posturing. So far, inter-

active kinematic algorithms or motion-capture are used to synthesize fixed animation sequences and motion-blending to smooth out transitions [77] [78]. While this approach is sufficient for animating the scenes of a motion picture, it is inadequate for dynamically changing environments. For example, an animation sequence of a character climbing stairs is unaffected by any unexpected obstacles.

Adaptive animation methods attempt to model articulated characters using physical-based animation [29] and computer learning techniques [30]. In this case, an animator defines the parameters of an environment rather than a single scene, and then reinforces the behavior of a character until it has learned to navigate and act autonomously. Interesting recent approaches include evolving neural networks [25] or layering different behavior controllers [79]. Both algorithms monitor the state of the skeleton and repetitively compute appropriate reaction forces. The animation process is therefore independent from the surroundings and will adapt to changes in the environment.

While the methods proposed by Faloutsos can already be supported using the capabilities of $3D^{os}$, it seems that future research should focus on evolving neural networks using genetic algorithms [80] [81]. Nature has already shown that a combination of learning and evolution methods can synthesize a plethora of locomotion and planning systems. An evolving control system also requires no explicit logic and can continue to learn and adapt. A combination of the Open Dynamics engine and virtual reality have already been used in conjunction with evolutionary programming when Tanev et al. evolved snake-like locomotion [82].

It would be interesting to investigate how to integrate the state of the art in neural network software and hardware with $3D^{os}$. A new software component will be required that integrates a Scene Tree with neural networks similar to how Petri Nets are integrated now. The virtual environment is an effective context for evaluating fitness, and the hybrid modelling language is a useful tool for executing the training

processes. Trained neural networks can be stored as new Scene Tree nodes and activated by the transitions of the Petri Net. However, it is still worthwhile to support key-frame animation sequences to model articulated characters [83].

10.2 Rigid Body Dynamics

Naturally, rigid body dynamics is an important component in the design of many mechatronic systems. Sufficiently large numbers of rigid bodies can even be used to approximate liquids. However, compounding the temporal discrepancies mentioned in Chapter 2, is the problem of how to represent artifacts in terms of their volumes rather than their surface features. Most dynamics engines represent simple shapes using primitives such as cubes, spheres and cylinders, and complex ones using triangles meshes. However, detecting collision and computing penetration depths is unreliable using triangle meshes. More recent approaches use convex hulls as the only primitive to represent all types of shapes.

Alliez et al. have recently proposed a method that tiles a 3D shape into isotropic tetrahedra using three-dimensional Delaunay triangulations to minimizes the radius of maximum containment for each tetrahedron [68]. Just like triangles tile two-dimensional space, tetrahedra tile three-dimensional space without leaving gaps. The algorithm has also produced tiling of natural shapes and mechanical parts while accurately retaining surface features. Closer to the surface, the tiles simply become smaller until the contours are met. Every object can therefore be modelled as a collection of smaller volumes, each of which occupies a minimal region of space. Tetrahedra also seem ideal candidates to represent rigid bodies, because they consist of simple shapes and centers of gravity. It is likely that there are equally simple ways of calculating collision and penetration depth. Realistic effects such as material breaking up during collision can then be simulated by removing rigid connections

among tetrahedra according to the strength of the reaction forces.

It would be worthwhile to investigate how to integrate Alliez's tiling algorithm into $3D^{os}$. The LCP solver of the Open Dynamics Engine is shape-independent and can be re-used to compute the reaction forces. However, data structures are needed to relate the strongly connected tetrahedra within a single rigid body with little computational overhead.

10.3 Plant growth

Another area of research is to model plant growth in order to simulate vegetation which effects the mechatronic systems that operate in natural environments. In general, vegetation influences friction and introduces obstacles, but in more specialized cases such as agriculture, mechatronic systems must observe crops and respond by planting, harvesting and watering.

The most common method to model plant growth is L-Systems [84]. Initially, an L-System is seeded by a string of symbols representing components such as branches, leaves, or buds. Through successive iterations, the symbols are continuously replaced according to predefined production rules. Different initial strings generate specimens and different production rules generate species including grasses, flowers and trees. A string is translated into a three-dimensional representation by correlating each symbol with a graphic primitive. Dynamic plant growth, which responds to obstacles, has also been demonstrated by adapting the production rules according to telemetry from the virtual environment [85].

Integrating L-Systems with $3D^{os}$ requires a parser that rewrites the production string. Just recently, Petri Nets have been used for the first time as a tool to specify parts of the rewriting process [86]. The authors have motivated their work by stating the simplicity and comprehensibility of the Petri Net language. It also remains to

integrate the continuously changing shapes of an L-System with rigid body dynamics algorithms.

10.4 Communication Infrastructure

Communication infrastructure has to model how sound or radio signals travel. In a natural environment the corresponding waves lose strength with increasing distance, and their range is effected by natural obstacles. One method for simulating this effect is to maintain proximity among producers and receivers, and to propagate a signal through nearest neighbors. Constraint Delaunay triangulations maintain proximity relationships efficiently while taking into consideration obstacles such as walls and terrain [87]. More importantly, the triangulations can be computed efficiently using computer graphics hardware [88]. Knowledge about proximity is also useful in constraint based motion planning of mechanical actuators [89].

10.5 Hybrid Modelling Language

There are still useful Petri Net extensions and analysis methods which are not yet part of $3D^{os}$. The quality of many Petri Net design patterns has already been documented, and this knowledge should be integrated in the development environment as a design guide. For example, users should be able to select patterns or templates while constructing simulations.

A priority is to support hybrid Petri Nets, which model continuous control flow explicitly using an additional set of constructs [90] [41]. Continuous places hold *fluids* instead of discrete tokens and continuous transitions distribute the fluid flow over the firing duration using user-defined flow-control functions. Hybrid Petri Nets combine discrete-event and continuous computation into a uniform hybrid modelling language with constructs explicitly designed to model fluid dynamics in terms of ordinary first-

order differential equations [91]. Most of the infrastructure and graphical widgets required to support the additional language constructs already exist in $3D^{os}$ and little additional development is necessary.

It should also be investigated how to improve the efficiency of the Python interpreter. Whenever a script is called, an execution frame is constructed which contains the call stack including all of the information required to interpret a script. In the case of $3D^{os}$, each Petri Net construct has its own frame. It seems possible to construct frames only once when the Petri Net is instantiated and then reuse them every time a script is called. In the model of Chapter 7, 2.4% of the computation was spent constructing frames. This number will increase significantly in applications which use in-occurrence inscriptions extensively. Before it becomes a performance bottleneck, a solution to this problem should be investigated.

During the latest revision of $3D^{os}$, the Petri Net constructs have also become Scene Tree nodes in order to simplify data management and storage. As a result, every Petri Net construct can also occupy a region in 3D space. It should be investigated to what extent occlusion culling can be used to simplify the computation within the Petri Net. At first glance, it seems that the control logic and dynamics computation must always be performed for the entire environment, regardless of whether it is visible or not. However, it may be possible to omit or simplify certain computation within *distant* constructs, and thereby improve runtime performance.

10.6 Autonomic Computing

In mid-October 2001, IBM released a manifesto observing that the main obstacle to further progress in the IT industry is a looming software complexity crisis [92]. The complexity of computing systems is approaching the limits of human comprehension in terms of managing software and computing devices. The goal is therefore to

develop autonomic computing systems that manage, configure and heal themselves without human intervention. The concept of *Autonomic Computing* seeks inspiration form self-governing social, economic or biological systems which learn and optimize with little or no centralized control.

Autonomous mechatronic systems can learn much form how social insects operate and display intelligence without implicit representation [93]. Complex behavior emerges from a collection of autonomous agents, such as ants, which have limited functionality and follow simple rules. However, since agents operate independently in a dynamic environment, their behavior is difficult to predict analytically. A virtual prototyping environment would therefore be a useful research and development tool in which to experiment with different configurations and behaviors. Since autonomous systems are also sensor-driven, most existing simulation tools cannot model them effectively. Since $3D^{os}$ has been specifically designed to express sensors, it will likely be a useful tool for developing autonomic mechatronic systems.

Weyns et al. have recently presented an architecture for an transportation system which has been developed in conjunction with Egemin, a commercial warehousing system vendor. The case study consists of a fleet of vehicles that distributes incoming goods to manufacturing cells and storage locations [37]. The vehicles navigate autonomously through a laser guidance system or by following magnets and cables embedded in the floor. In a traditional architecture, a centralized planning component computes schedules and transmits transportation assignments to each vehicle. In an autonomic architecture, vehicles have to find assignments themselves, according to local conditions and priorities. One advantage is that in a truly autonomic system, vehicles can simply be added and removed without any reconfiguration. It would be worthwhile to investigate to what extent modelling the transportation systems using $3D^{os}$ can benefit the development process.

10.7 Control Synthesis

A simulation of a control system is software that models other software. It is only logical to assume that as a simulation becomes more realistic, it will eventually contain the logic required to control its physical counterpart. The process of extracting the *real* control system from the simulation is called *control synthesis*. The process is similar to how some engineering design tools generate continuous control instructions (CNC) from geometric descriptions. Control synthesis has the advantage of reusing control logic which has been validated in a simulation. Theoretically, reliable systems can be synthesized before any physical components have been constructed.

Asynchronous control synthesis is also referred to as off-line programming, where control logic is extracted from the same model that defines the simulation. IGRIP for example, exports CNC instruction sets from the model of a work cell. The first step in supporting this feature is an object library containing the geometric description of common industrial devices. Most manufacturers already supply 3D models in different data formats, and it remains to develop converters for $3D^{os}$. Interfaces and algorithms to model jointed mechanical actuators are also required in order to record the activities of a work cell effectively. Much inspiration for the design of this functionality can be gained from existing tools such as IGRIP.

Synchronous control synthesis is also called simulation-based control in which the software that computes the simulation also controls the physical system at the same time. The challenges of this approach are to synchronize the sensors and actuators in the simulation with their physical counterparts. The advantage is that the same software acts as the control system, user interface and analysis tool. Simulation-based control opens the door for unprecedented design and control processes, in which the state of the simulation and that of the physical system are seamlessly interchangeable. For example, a physical device can receive input from a

simulation in order to respond to components which have not even been constructed yet.

Synchronous control synthesis is only possible when a simulation closely models the parameters of a physical system. Proof-of-concept has already been demonstrated by tools like Gazebo and Webots. Considering that $3D^{os}$ maintains a realistic representation of the logical and structural components, it may not be too difficult to achieve simulation-based control as well. The first step is to develop a Scene Tree node library which encapsulates common analog and digital I/O devices. The color interfaces of places, transitions and arcs then have to be modified to resemble Programmable Logical Controller (PLC) programming interfaces. PLCs are standardized embedded processors used to execute control logic at the assembler level in terms of I/O registers and memory. A Petri Net would then execute logic that controls hardware at the lowest possible level of abstraction. A simple hardware abstraction layer can then switch I/O between the simulation and the embedded systems [76].

10.8 Molecular Manufacturing

An emerging application area for simulation technology is micro or nano-scale manufacturing. Most of the fundamental science in this domain has already been conducted and it now remains an engineering challenge to construct a molecular manufacturing system. There already exist devices such as nano-scale motors [94], conveyors [95] and valves [96]. At this point only simple components can be manufactured, but at some level of complexity, control systems similar to the ones used in macro-scale manufacturing will likely be required to assemble more complex components [97]. At the same time, it is especially important to validate nano-scale systems thoroughly before they are released into the environment. Petri Nets have already

been used to model and analyze molecular processes, and the language will likely be of further use in this domain [47, 48].

It would be interesting to model micro or nano-electromechanical systems such as the molecular assembler described by Phoenix et al. [98]. One challenge will certainly be to distribute the simulation over many processors and divide $3D^{os}$ into separate development and runtime environments. Also, molecular systems are influenced by molecular effects such as Brownian motion. Research is required to determine what affects are important and how they can be simulated [99].

10.9 Summary

This chapter introduced topics of future research motivated by improving and adopting $3D^{os}$. The most important objective is to monitor emerging research and development, and to integrate new technology as it becomes available. It also seems that the computational resources required for general virtual environments are immense and that computer hardware must be leveraged extensively. Many of the methods recommended in this section are motivated by their potential for effective hardware support. L-Systems, neural networks and Delaunay triangulations for example, all have a relatively simple structure and dynamic behavior. They are therefore ideal candidates for dedicated hardware.

So far, virtual reality technology has also proven useful as a new media for art, or as a teaching and training tool [100]. While the ideas expressed in this chapter are motivated by mechatronics, there are many other applications for a virtual reality authoring tool.

Chapter 11

Conclusions

"The same law that enslaves men can also set them free. - King Arthur"

At the most fundamental level, this thesis has introduced a method that supports the spiral model of software development in the context of mechatronic systems design [101]. The essence of this process is short cycles between development and evaluation that provide frequent feedback about the quality of a design. Not unlike natural evolution, the spiral model leads to satisfactory solutions though continuous selection and refinement. Figure 11.1 depicts the development process which is simply an outward spiral of continuously repeating analysis, design, implementation and testing phases. Each turn around the spiral builds a base for the next phase of development.

The advantage of the spiral model is that experiences gained through a previous cycle influences the decisions made during the next cycle. Ineffective solutions are detected early during the development and therefore have less negative impact. Also, since a system has to execute more frequently, developers are encouraged to focus more on functionality rather than architecture. Time consuming formal design is de-emphasized and more resources are available to explore a larger solution space.

The spiral model is especially effective for designing software, because a failed attempt rarely costs resources other than developers' time. Therein lays a fundamental difference between the development of software and most other forms of engineering design, where materials are wasted and safety is compromised with a

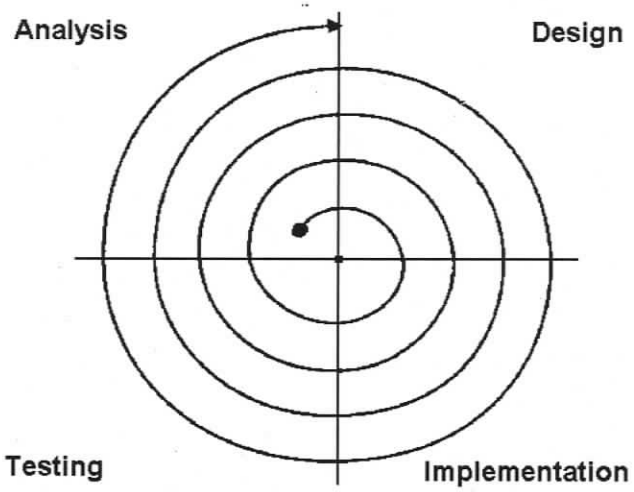

Figure 11.1: The Spiral Model of Software Development

trial and error development approach. The spiral model advocates a software development process which leverages the ease by which source code can be re-factored. One initially chooses the most obvious and simple solution to a problem, and then only motivates changes through evidence collected during analysis and testing. The subsequent re-factoring process will then lead to a more solid foundation for the next development cycle.

In order to apply the spiral model effectively in the context of mechatronic control software, expensive and often hazardous physical prototypes are required in order to generate realistic input. The challenge of this dissertation was to create sensor telemetry artificially and simulate the feedback loop between the control logic and physical structure of mechatronic systems using software. This goal has been achieved by integrating virtual reality technology with a hybrid modelling language to the extent of the capabilities of $3D^{os}$. The development environment can already be used to model sizeable distributed mechatronic systems in terms of actuators that

respond to physical stimuli and sensors that produce realistic values. Geometry-driven Petri Nets therefore support the spiral model by providing readily available and less costly virtual prototypes for testing and analysis in the place of physical ones.

This thesis has also demonstrated an application of Geometry-driven Petri Nets using an example that reflects a realistic manufacturing process including actuator control and resource tracking. Chapter 7 suggests how $3D^{os}$ supports the implementation, analysis and testing phases of the spiral model. The iconic modelling constructs and the structural properties of Petri Nets effectively support the development of large hierarchical and distributed hybrid control systems. The behavioral properties and the graphical display of the virtual environment, in addition, support many forms of analysis. Chapter 7 also shows how a cycle time analysis can suggest bottlenecks.

11.1 Contributions

This dissertation first and foremost contributes a new software engineering method for designing adaptive mechatronic control systems using a combination of a hybrid modelling language and virtual reality technology. Some components of this method are defined using mathematical formalism and others using the C++ programming language. A design pattern for how to integrate the necessary computational concepts and how to manage their interaction is documented throughout this dissertation as a reference for future research.

This dissertation also contributes the $3D^{os}$ development environment for Geometry-driven Petri Nets. Diverse user interface widgets and interaction models, as well as software architectures and algorithms have been specifically designed to integrate Petri Nets and virtual reality technology. The environment also supports novel

ways of editing and executing the Petri Net language, including an interface that applies actual colors to identify the different types of tokens and variables. In addition, a type system that supports arbitrary nesting of Petri Net pages was also developed.

This dissertation also contributes a technique for applying Geometry-driven Petri Nets in the context of mechatronic system simulation. Chapter 7 outlines how to construct simulations, and introduces various design patterns. It is also shown how to leverage the combination of Petri Nets and virtual reality technology to construct more accurate process models and compute the cycles times of a system.

Finally, this dissertation contributes a new generation in the evolution of Petri Nets. Traditionally, the formalism does not incorporate a geometric description of the system being modelled, and variables representing sensors and actuators are either approximated or randomized. Geometry-driven Petri Nets instead bind many of the parameters to a virtual environment containing a physically correct representation of the system. By validating a process model visually and by cross-referencing the state of the Petri Net with that of a virtual environment more correct control systems can be synthesized.

11.2 Final Thoughts

Through software, mathematics has evolved from an analysis tool to a synthesis tool. No longer is the tedious process of computing symbolisms limited by human cognitive powers. Software can now compute entire systems of mathematical models fast enough to simulate, and thereby imitate a real thing to the level of its observable interface [12]. Computer games, for example, already model certain types of interactive environments to the point that our auditory and visual senses are stimulated convincingly. This thesis has focused on constructing virtual mechatronic systems.

By definition, true virtual reality must represent everything, including the types of systems that can be described using Petri Nets.

In general, there is still a lack of authoring tools designed to construct and execute large and diverse virtual environments. While $3D^{os}$ began as a tool to demonstrate Geometry-driven Petri Nets, its focus is now shifting towards a general virtual reality engine driven by the philosophy that more accurate models of the physical world will produce more useful prototypes.

Bibliography

[1] Robert Diamond, James O. Henriksen, C. Dennis Pegden, Tony Waller, Charles R. Harrell, William B. Nordgren, Matthew W. Rohrer, and Averill M. Law. Future of simulation software: the current and future status of simulation software (panel). In *Proceedings of the Winter Simulation Conference*, pages 1633–1640, 2002.

[2] Daniel Jackson and Martin Rinard. Software analysis: a Roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 133–145, New York, NY, USA, 2000. ACM Press.

[3] Daniel Williams, Daniel Finke, D. J. Medeiros, and Mark Traband. Discrete simulation development for a proposed shipyard steel processing facility. In *Proceedings of the Winter Simulation Conference*, pages 882–887, Washington, DC, USA, 2001. IEEE Computer Society.

[4] Deogratias Kibira and Chuck McLean. Manufacturing modeling methods: virtual reality simulation of a mechanical assembly production line. In *Proceedings of the Winter Simulation Conference*, pages 1130–1137, 2002.

[5] Jeffrey Kephart and David Chess. The Vision of Autonomic Computing. *IEEE Computer Journal*, 36(1):41–50, 2003.

[6] Pattie Maes. Modeling adaptive autonomous agents. *Artificial Life, I*, (1&2)(9), 1994.

[7] Panos Antsaklis and Xenofon Koutsoukos. On hybrid control of complex systems: A Survey. In *Proceedings of the International Conference on Automation of Mixed Processes*, pages 1–8, 1998.

[8] G. A. Kaminka, M. M. Veloso, S. Schaffer, C. Sollitto, R. Adobbati, A. N. Marshal, A. Scholer, and S. Tejada. GameBots: The ever-challenging multi-agent research test-bed. *ACM Transactions on Computer Graphics*, January 2002.

[9] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2149–2154, Sendai, Japan, 9 2004.

[10] Olivier Michel. Webots: Symbiosis Between Virtual and Real Mobile Robots. In *Proceedings of the International Conference on Virtual Worlds*, pages 254–263. Springer-Verlag, 1998.

[11] Mary Shaw. Writing good software engineering research papers: minitutorial. In *ICSE '03: Proceedings of the 25th IEEE International Conference on Software Engineering*, pages 726–736, Washington, DC, USA, 2003. IEEE Computer Society.

[12] Herbert Simon. *The Science of the Artificial.* MIT Press, Cambridge, MA, USA, 1996.

[13] Alfred Aho and John Hopcroft. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1974.

[14] Hanan Samet. The Quadtree and Related Hierarchical Data Structures. *ACM Computer Surveys*, 16(2):187–260, 1984.

[15] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive Rendering with Coherent Ray Tracing. In A. Chalmers and T.-M. Rhyne, editors, *EG 2001 Proceedings*, volume 20(3), pages 153–164. Blackwell Publishing, 2001.

[16] Shachar Fleishman, Daniel Cohen-Or, and Marc Alexa. Progressive point set surfaces. *ACM Transactions on Computer Graphics*, 22(4):997–1011, 2003.

[17] Frank Losasso and Hugues Hoppe. Geometry clipmaps: terrain rendering using nested regular grids. *ACM Transactions on Computer Graphics*, 23(3):769–776, 2004.

[18] Carsten Dachsbacher and Marc Stamminger. Rendering procedural terrain by geometry image warping. In *Rendering Techniques*, pages 103–110, 2004.

[19] J. Blow. Terrain rendering at high levels of detail. In *Game Developers Conference*, 2000.

[20] Renato Pajarola. Large scale terrain visualization using the restricted quadtree triangulation. In David Ebert, Hans Hagen, and Holly Rushmeier, editors, *IEEE Visualization Journal*, pages 19–26, 1998.

[21] Peter Lindstrom, David Koller, William Ribarsky, Larry Hodges, Nick Faust, and Gregory Turner. Real-time, continuous level of detail rendering of height fields. In *Proceedings of the Conference on Computer Graphics and interactive techniques*, pages 109–118, 1996.

[22] www.remotesensing.org. Geospatial Data Abstraction Library.

[23] www.usgs.gov. United States Geological Survey.

[24] Mike Paterson and Frances Yao. Efficient binary space partition for hidden-surface removal and solid modelling. *Discrete Computational Geometry*, 5(5):485–503, 1990.

[25] Radek Grzeszczuk, Demetri Terzopoulos, and Geoffrey Hinton. NeuroAnimator: fast neural network emulation and control of physics-based models. In *SIGGRAPH '98: Proceedings of the 25th annual ACM Conference on Computer graphics and interactive techniques*, pages 9–20, New York, NY, USA, 1998. ACM Press.

[26] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The algorithmic beauty of plants*. Springer-Verlag, New York, NY, USA, 1996.

[27] O. C. Zienkiewicz and R. L. Taylor. *The Finite Element Method*. Butterworth-Heinemann, fifth edition, 2000.

[28] T.M. Liggett. *Interacting Particle Systems*. Springer Verlag, 1985.

[29] Joseph Laszlo, Michiel van de Panne, and Eugene Fiume. Interactive control for physically-based animation. In *SIGGRAPH '00: Proceedings of the 27th annual ACM Conference on Computer graphics and interactive techniques*, pages 201–208, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[30] Jonathan Dinerstein and Parris Egbert. Fast multi-level adaptation for interactive autonomous characters. *ACM Transaction of Computer Graphics*, 24(2):262–288, 2005.

[31] Per Lötstedt. Numerical simulation of time-dependent contact friction problems in rigid body mechanics. *SIAM Journal of Scientific Statistical Computing*, 5(2):370–393, 1984.

[32] Matthew Mason. Numerical Simulation of Time-Dependent Contact and Friction Problems in Rigid Body Mechanics: A Review. In Oussama Khatib, John J. Craig, and Tomas Lozano-Perez, editors, *The Robotics Review 1*, pages 311–313. MIT Press, Cambridge, MA, 1989.

[33] David Baraff. Analytical methods for dynamic simulation of non-penetrating rigid bodies. In *SIGGRAPH '89: Proceedings of the 16th annual ACM Conference on Computer graphics and interactive techniques*, pages 223–232, New York, NY, USA, 1989. ACM Press.

[34] Danny Kaufman, Timothy Edmunds, and Dinesh Pai. Fast frictional dynamics for rigid bodies. *ACM Transactions on Computer Graphics*, 24(3):946–956, 2005.

[35] www.ode.org. The Open Dynamics Engine.

[36] John Rieffel and Jordan Pollack. Automated assembly as situated development: using artificial ontogenies to evolve buildable 3-D objects. In *GECCO '05: Proceedings of the 2005 ACM Conference on Genetic and evolutionary computation*, pages 99–106, New York, NY, USA, 2005. ACM Press.

[37] Danny Weyns, Kurt Schelfthout, and Tom Holvoet. Architectural design of a distributed application with autonomic quality requirements. In *DEAS '05: Proceedings of the 2005 workshop on Design and evolution of autonomic application software*, pages 1–7, New York, NY, USA, 2005. ACM Press.

[38] Josie Wernecke. *The Inventor Mentor: Programming Object-Oriented 3d Graphics with Open Inventor, Release 2.* Addison-Wesley Longman Publishing Co., 1993.

[39] Jan-Erik Strömberg, Simin Nadjm-Tehrani, and Jan L. Top. Switched Bond Graphs as Front-End to Formal Verification of Hybrid Systems. In *Hybrid Systems*, pages 282–293, 1995.

[40] P. Mosterman and G. Biswas. Behavior generation using model switching: A hybrid bond graph modeling technique. In *Proceedings of the Society for Computer Simulation*, volume 27, pages 177–182, 1995.

[41] Hassane Alla and René David. Continuous and Hybrid Petri Nets. *Journal of Circuits, Systems, and Computers*, 8(1):159–188, 1998.

[42] R. A., C. Courcoubetis, T. A. Henzinger, and P. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. *Hybrid Systems*, pages 209–229, 1992.

[43] Carl Adam Petri. Fundamentals of a Theory of Asynchronous Information Flow. In *Proceedings of IFIP Congress*, volume 62, pages 386–390, 1963.

[44] Lars Michael Kristensen and Kurt Jensen. Specification and validation of an edge router discovery protocol for mobile ad hoc networks. In *SoftSpez Final Report*, pages 248–269, 2004.

[45] H. Demmou M. Paludetto J. Porras G. Moncelet, S. Christensen. Analyzing a Mechatronic System with Coloured Petri Nets. In *International Journal on Software Tools for Technology Transfer*, volume 2, pages 160–167, 1999.

[46] S. Miyano. How to model and simulate biological pathways with petri nets. In *Proceedings of the International Conference on Application and Theory of Petri Nets*, 2004.

[47] J. Barjis and I. Barjis. Formalization of the Protein Production by Means of Petri Nets. In *Proceedings of the International Conference on Information Intelligence and Systems*, pages 4–9.

[48] Leo Ojala, Olli-Matti Penttinen, and Elina Parviainen. Modeling and Analysis of Margolus Quantum Cellular Automata Using Net-Theoretical Methods. In *Proceedings of Applications and Theory of Petri Nets 2004: 25th International Conference, ICATPN 2004, Bologna, Italy, June 21-25, 2004 — Volume 3099 of Lecture Notes in Computer Science / Cortadella, Reisig (Eds.)*, pages 331–350. Springer-Verlag, September 2004.

[49] Tadao Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–579, 1989.

[50] Kurt Jensen. *Coloured Petri Nets: Basic Concepts*, volume 1. Springer Verlag, 1992.

[51] Kurt Jensen. *Coloured Petri Nets: Analysis Methods*, volume 2. Springer Verlag, 1994.

[52] Kurt Jensen. *Coloured Petri Nets: Practical Use*, volume 3. Springer Verlag, 1997.

[53] C. Ramchandani. *Analysis of asynchronous concurrent systems by timed Petri Nets.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1974.

[54] B. Berthomieu and M. Diaz. Modelling and verification of time-dependent systems using time petri nets. page 17, 1991.

[55] Wilfried Brauer, Robert Gold, and Walter Vogler. A Survey of Behaviour and Equivalence Preserving Refinements of Petri Nets. In *Lecture Notes in*

Computer Science; Advances in Petri Nets 1990, volume 483, pages 1–46, Berlin, Germany, 1991. Springer-Verlag.

[56] Rainer Fehling. A Concept of Hierarchical Petri Nets with Building Blocks. In *Proceedings of the International Conference on Applications and Theory of Petri Nets*, pages 148–168. Springer-Verlag, 1993.

[57] W. M. Zuberek and I. Bluemke. Hierarchies of Place/Transitions Refinements in Petri Nets. In *Proceedings of the Conference on Emerging on Technologies and Factory Automation*, pages 355–360, 1996.

[58] Jörn Janneck and Robert Esser. Higher-order Petri net modeling—techniques and applications. In *Workshop on Software Engineering and Formal Methods*, 2002.

[59] T. R. G. Green and M. Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. volume 7, pages 131–174, 1996.

[60] Vlatka Hlupic. Simulation software: an Operational Research Society survey of academic and industrial users. In *Proceedings of the Winter Simulation Conference*, pages 1676–1683, 2000.

[61] Jakob Nielsen and Rolf Molich. Heuristic evaluation of user interfaces. In *CHI '90: Proceedings of the SIGCHI ACM Conference on Human factors in computing systems*, pages 249–256, New York, NY, USA, 1990. ACM Press.

[62] E. Yucesan C.H. Chen J.L Snowdon and J.M. Charnes. An evaluation and selection methodology for discrete-event simulation software. In *Proceedings of the Winter Simulation Conference*, pages 67–75, 2002.

[63] Melody Ivory and Marti Hearst. The state of the art in automating usability evaluation of user interfaces. *ACM Computer Surveys*, 33(4):470–516, 2001.

[64] Michel Beaudouin-Lafon. Designing interaction, not interfaces. In *AVI '04: Proceedings of the ACM Conference on Advanced visual interfaces*, pages 15–22, New York, NY, USA, 2004. ACM Press.

[65] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, and R. Little. *Documenting Software Architectures.* Addison-Wesleay, 2003.

[66] www.rational.com. Rational Rose.

[67] Scott Miller and Dennis Pegden. Manufacturing simulation: introduction to manufacturing simulation. In *Proceedings of the Winter Simulation Conference*, pages 63–66, 2000.

[68] Pierre Alliez, David Cohen-Steiner, Mariette Yvinec, and Mathieu Desbrun. Variational tetrahedral meshing. *ACM Transactions on Computer Graphics*, 24(3):617–625, 2005.

[69] Wlodzimierz M. Zuberek. Timed Petri Nets in Modelling and Evaluation of Multiprocessor Systems. In *ICPP*, pages 695–698, 1987.

[70] Jens Linneberg Rasmussen and Mejar Singh. Designing a Security System by Means of Coloured Petri Nets. In *Application and Theory of Petri Nets*, pages 400–419, 1996.

[71] Gilles Moncelet, Søren Christensen, Hamid Demmou, Mario Paludetto, and José Porras. Analysing a Mechatronic System with Coloured Petri Nets. *Software Tools for Technology Transfer*, 2(2):160–167, 1998.

[72] Steven Gordon and Jonathan Billington. Analysing a Missile Simulator with Coloured Petri Nets. *Software Tools for Technology Transfer*, 2(2):144–159, 1998.

[73] Michel Beaudouin-Lafon, Wendy E. Mackay, Peter Andersen, Paul Janecek, Mads Jensen, Henry Michael Lassen, Kasper Lund, Kjeld Hoyer Mortensen, Stephanie Munck, Anne V. Ratzer, Katrine Ravn, Soren Christensen, and Kurt Jensen. CPN/tools: A post-WIMP interface for editing and simulating coloured petri nets. In *ICATPN*, pages 71–80, 2001.

[74] www.delmia.com. The Delmia tool suite. www.delmia.com.

[75] Frank Cheng. A methodology for Developing Robotic Workcell Simulation Models. In *Proceedings of the Winter Simulation Conference*, 8 2000.

[76] Brian P. Gerkey, Richard T. Vaughan, and Andrew Howard. The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems. In *Proceedings of the International Conference on Advanced Robotics*, pages 317–323, Coimbra, Portugal, Jul 2003.

[77] Michael Girard. Interactive design of 3-D computer-animated legged animal motion. In *SI3D '86: Proceedings of the 1986 workshop on Interactive 3D graphics*, pages 131–150, New York, NY, USA, 1987. ACM Press.

[78] Erika Chuang and Christoph Bregler. Mood swings: expressive speech animation. *ACM Transactions on Computer Graphics*, 24(2):331–347, 2005.

[79] Petros Faloutsos, Michiel van de Panne, and Demetri Terzopoulos. Composable controllers for physics-based character animation. In *SIGGRAPH '01: Proceedings of the 28th annual ACM Conference on Computer graphics and interactive techniques*, pages 251–260, New York, NY, USA, 2001. ACM Press.

[80] David Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[81] J. R. Koza. *Genetic Programming: On the Programming of Computers by Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.

[82] Ivan Tanev. Learned mutation strategies in genetic programming for evolution and adaptation of simulated snakebot. In *GECCO '05: Proceedings of the 2005 ACM Conference on Genetic and evolutionary computation*, pages 687–694, New York, NY, USA, 2005. ACM Press.

[83] Alberto Barbosa Raposo, Léo Pini Magalhães, and Alessandro de Lima Bicho. Control of Articulated Figures Animations Using Petri Nets. In *XIV Brazilian Symposium on Computer Graphics and Image Processing*, pages 200–207, 2001.

[84] Aristid Lindenmayer. Mathematical model for cellular interaction in development, Parts I and II. *Journal of Theoretical Biology*, 18:280–315, 1968.

[85] Przemyslaw Prusinkiewicz. Modeling of spatial structure and development of plants. *Scientia Horticulturae*, 74:113–149, 1998.

[86] William Remphrey Przemyslaw Prusinkiewicz. Characterization of architectural tree models using L-Systems and Petri Nets. In *4th International Symposium on the Tree*, pages 177–186, 2003.

[87] Franz Aurenhammer. Voronoi diagrams a survey of a fundamental geometric data structure. *ACM Computer Surveys*, 23(3):345–405, 1991.

[88] Kenneth Hoff, John Keyser, Ming Lin, Dinesh Manocha, and Tim Culver. Fast computation of generalized Voronoi diagrams using graphics hardware. In *SIGGRAPH '99: Proceedings of the 26th annual ACM Conference on Computer graphics and interactive techniques*, pages 277–286, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[89] Maxim Garber and Ming Lin. Constraint-based motion planning for virtual prototyping. In *SMA '02: Proceedings of the seventh ACM symposium on Solid modeling and applications*, pages 257–264, New York, NY, USA, 2002. ACM Press.

[90] Ralf Wieting. Hybrid high-level nets. In *Proceedings of the Winter Simulation Conference*, pages 848–855, New York, NY, USA, 1996. ACM Press.

[91] Fabio Balduzzi, Alessandro Giua, and Carla Seatzu. Modeling and simulation of manufacturing systems with first-order hybrid Petri Nets. *International Journal of Production Research*, 39(2):255–282, 2001.

[92] www.ibm.com. Autonomic Computing: IBM's Perspective on the State of Information Technology.

[93] Rodney Brooks. Intelligence without representation. Number 47 in Artificial Intelligence, pages 139–159. 1991.

[94] G. Bachand and C. Montemagno. Constructing Organic/Inorganic NEMS Devices Powered by Biomolecular Motors. *Biomedical Microdevices*, 2(3):179–184, 2000.

[95] R. O. Ritchie B. C. Regan, S. Aloni and U. Dahmen. Carbon nanotubes as nanoscale mass conveyors. *Nature*, 428:924 – 927, 2004.

[96] L. Kong, Q. Pan, B. Cui, M. Li, and Chou. Magnetotransport and domain structures in nanoscale NiFe/Cu/Co spin valve. *Journal of Applied Physics*, 85:5492–5494, April 1999.

[97] Chemical Industry Vision2020 Technology Partnership. Chemical Industry R&D Roadmap for Nanomaterials by Design: From Fundamentals to Function.

[98] Chris Phoenix. Design of a primitive Nanofactory. *Journal of Evolution and technology.*

[99] D. Bindel J.V. Clark, W. Kao, E. Zhu, A. Kuo, N. Zhou, J. Nie, J. Demmel, Z. Bai, S. Govindjee, K.S.J. Pister, M. Gu, and A. Agogino. Addressing the needs of complex MEMS design. In *Proceedings of MEMS '02*, pages 204–209, Piscataway, NJ, USA, 2002.

[100] Max North, Joseph Sessum, and Alex Zakalev. Immersive visualization tool for pedagogical practices of computer science concepts: a pilot study. *Consortium for Computing Sciences in Colleges*, 19(3):207–215, 2004.

[101] Barry Boehm. A spiral model of software development and enhancement. *SIGSOFT Software Engineering Notes*, 11(4):14–24, 1986.

Appendix A

Class Listing

Table A.1: Classes of module Geo

Fields
GeoBox3f
GeoColor
GeoCylinder
GeoGlobal
GeoLine
GeoMatrix
GeoPlane
GeoQuaternion
GeoSphere
GeoString
GeoVec2f
GeoVec3f

Table A.2: Classes of module Scn

Nodes	*Manipulators*	*Misc.*
ScnNode	ScnManipulator	ScnVolume
ScnSpace	ScnManipulatorTransform	ScnContact
ScnLight	ScnManipulatorBox	ScnNodeData
ScnMaterial	ScnManipulatorJoint	ScnType
ScnCamera	ScnManipulatorTab	ScnGlobal
ScnCameraStatic	ScnManipulatorGradient	
ScnCameraExamine	ScnManipulatorLight	
ScnGeometry	ScnManipulatorOutlineBox	
ScnCube	ScnManipulatorOutlineCoord	
ScnCone	ScnManipulatorCamera	
ScnCylinder		
ScnPlane	ScnPart	
ScnPolygonMesh	ScnPartRay	
ScnTerrain	ScnPartRotatorSphere	
ScnTrimesh	ScnPartRotatorBox	
ScnSphere	ScnPartTranslatePlane	
ScnGradientField	ScnPartAxis	
ScnGroup	ScnPartRange	
ScnWorld	ScnPartCone	
ScnTransform	ScnPartPoint	
ScnJoint	ScnPartDirection	
ScnRoot	ScnPartTabPlane	
ScnBody		
ScnAvatar		
ScnFog		

Table A.3: Classes of module Scn

Fields	*Sensors*	*Action*
ScnField	ScnSensor	ScnAction
ScnSFInt	ScnSensorCollision	ScnActionVolume
ScnSFEnum	ScnCollisionCylinder	ScnActionRender
ScnSFFloat	ScnCollisionSensor	ScnActionUpdate
ScnSFBool	ScnCollisionSphere	ScnActionTransform
ScnSFString	ScnCollisionCone	ScnActionCreate
ScnSFVec3f	ScnCollisionCube	ScnActionDestroy
ScnSFVec2f	ScnSensorKey	ScnActionEvent
ScnSFMatrix	ScnSensorGradient	ScnActionCallback
ScnSFColor		
ScnSFRef		
ScnMFInt		
ScnMFFloat		
ScnMFVec3f		
ScnMFVec2f		
ScnMFFunction		
ScnMFRef		

Table A.4: Classes of module Man

Manipulators
ManManipulator
ManManipulatorTransform
ManManipulatorBox
ManManipulatorJoint
ManManipulatorTab
ManManipulatorGradient
ManManipulatorLight
ManManipulatorOutlineBox
ManManipulatorOutlineCoord
ManManipulatorCamera

Table A.5: Classes of module Cmd

Instance	
CmdArc	CmdArcColor
CmdTransition	CmdArcControlColor
CmdPlaceInterface	CmdArcDataColor
CmdLink	CmdArcSensorColor
CmdPlace	CmdPlaceColor
CmdPage	CmdPlaceControlColor
CmdNet	CmdPlaceDataColor
CmdGlobal	CmdPlaceSensorColor
CmdQueue	
CmdReference	
CmdFunction	

Table A.6: Classes of module Win

graph	*PetriNet*
WinComponent	WinNode
WinGlobal	WinPlace
WinList	WinTransition
WinView	WinLink
WinOrthoZoomWindow	WinArc
WinOrthoPosWindow	WinInstance
WinWindow	WinText
	WinPage
	WinReference

Table A.7: Classes of module Rtn

Runtime	
RtnNode	RtnPlaceColor
RtnTransition	RtnPlaceControlColor
RtnPlace	RtnPlaceDataColor
RtnLink	RtnPlaceSensorColor
RtnArc	RtnArcColor
RtnReference	RtnArcControlColor
RtnPage	RtnArcDataColor
RtnObject	RtnArcSensorColor
RtnGlobal	RtnTransitionColor
RtnSelection	RtnTransitionDataColor
RtnTypeInfo	RtnTransitionFunctionColor
RtnActionMark	RtnTransitionSensorColor
RtnActionClear	
RtnActioninit	
RtnTraversal	
RtnSetTokenTraversal	
RtnClrTokenTraversal	

Table A.8: Classes of module App

Dialogs	*Controls*	*Frame*
DlgNode	CtlField	FrmGlobal
DlgMaterial	CtlSFInt	FrmMain
DlgLight	CtlSFFloat	FrmProject
DlgBody	CtlSFBool	FrmMaterial
DlgTransform	CtlSFEnum	FrmPage
DlgCube	CtlSFVec3f	FrmType
DlgSphere	CtlSFMatrix	
DlgCylinder	CtlSFColor	
DlgTerrain	CtlSFColor	
DlgTrimesh	CtlContactData	
DlgFog	CtlGrid	
DlgCamera	CtlArcGrid	
DlgAvatar	CtlPlaceGrid	
DlgGradient	CtlTransitionGrid	
DlgWorld	CtlHandlers	
DlgArc	CtlOpenGl	
DlgPlace	CtlOutput	
DlgTransition	CtlPetriView	
DlgOverview	CtlWorldView	
DlgToolbarGradient		

Appendix B

Kernel Source Code

This appendix contains the source code for the Petri Net Kernel of $3D^{os}$ in the order shown in table B.1:

Table B.1: Kernel File Listing

File
CmdTransition.h
CmdTransition.cpp
CmdPlace.h
CmdPlace.cpp
CmdArc.h
CmdArc.cpp

```
class CmdTransition : public ScnNode
{
    public:

        CmdTransition(CmdNet& iCmdNet,
                      CmdPage& iPage,
                      ScnSFFunction& iPrefireFunction,
                      ScnSFFunction& iPostfireFunction,
                      ScnSFFunction& iInfireFunction);

        enum EState
        {
            kENABLED,
            kDISABLED,
            kFIRE
        };

        ScnNode* produce(int iColor);
        void consume(int iColor, ScnNode* iContent);

        int mFirecount;
        DWORD mDelay;
        DWORD mDuration;
        float mProbability;
        CmdQueueLink<CmdTransition> mLink;

        CmdPage& mPage;

    protected:

        void refresh();

        void addArc(CmdArc& iArc);
        void remArc(CmdArc& iArc);

        void enable();
        void disable();

        COLContainer<CmdArc*> mOutgoingArcs;
        COLContainer<CmdArc*> mIncomingArcs;

        BITMap<ScnNode*> mTokenSet;
        BITMap<ScnSensor*> mVariableSet;

    protected:

        CmdNet& mCmdNet;

    protected:
```

```
        void beginOccurence(int iStamp);
        void endOccurence(int iStamp);
        bool doOccurence(int iStamp);

        ScnSFFunction& mPrefireFunction;
        ScnSFFunction& mPostfireFunction;
        ScnSFFunction& mInfireFunction;

    private:

        int mEnabled;
        int mStamp;

        EState mState;
};
```

```
CmdTransition::CmdTransition(CmdNet& iCmdNet,
                             CmdPage& iPage,
                             ScnSFFunction& iPrefireFunction,
                             ScnSFFunction& iPostfireFunction,
                             ScnSFFunction& iInfireFunction)
: ScnNode()

, mCmdNet(iCmdNet)
, mPage(iPage)
, mEnabled(0)
, mOutgoingArcs(5)
, mIncomingArcs(5)
, mDelay(0.0)
, mDuration(0.0)
, mProbability(1.0)
, mPrefireFunction(iPrefireFunction)
, mPostfireFunction(iPostfireFunction)
, mInfireFunction(iInfireFunction)
, mLink(*this)
, mStamp(0)
, mFirecount(0)
, mState(kENABLED)
, mTokenSet(0)
, mVariableSet(0)
{
    mCmdNet.addReadyTransition(mLink);
}

void CmdTransition::addArc(CmdArc& iArc)
{
    if (iArc.getState() == CmdArc::kENABLED)
    {
        mEnabled++;
    }

    switch (iArc.mDirection)
    {
    case kToPlace:
        mOutgoingArcs.put(&iArc);
        break;
    case kToTransition:
        mIncomingArcs.put(&iArc);
        break;
    }

    refresh();
}

void CmdTransition::remArc(CmdArc& iArc)
```

```
{
    if (iArc.getState() == CmdArc::kENABLED)
    {
        mEnabled--;
    }

    switch (iArc.mDirection)
    {
    case kToPlace:
        mOutgoingArcs.rem(&iArc);
        break;
    case kToTransition:
        mIncomingArcs.rem(&iArc);
        break;
    }

    refresh();
}

void CmdTransition::enable()
{
    mEnabled++;
    refresh();
}

void CmdTransition::disable()
{
    mEnabled--;
    refresh();
}

void CmdTransition::refresh()
{
    if (mEnabled == (mOutgoingArcs.size() + mIncomingArcs.size()))
    {
        if (mState == kDISABLED)
        {
            mState = kENABLED;
            mCmdNet.addReadyTransition(mLink);
        }
    }
    else
    {
        if (mState == kENABLED)
        {
            mState = kDISABLED;
            mCmdNet.remReadyTransition(mLink);
        }
    }
```

```
}

ScnNode* CmdTransition::produce(int iColor)
{
    if (mTokenSet.getBit(iColor))
    {
        return mTokenSet.clrContent(iColor);
    }
    else
    {
        return 0;
    }
};

void CmdTransition::consume(int iColor, ScnNode* iContent)
{
    mTokenSet.setContent(iColor, iContent);
};

void CmdTransition::beginOccurence(int iStamp)
{
    mStamp = iStamp;
    mState = kFIRE;

    CmdArc** lIter = mIncomingArcs.begin();
    while (lIter != mIncomingArcs.end())
    {
        (*lIter++)->remove();
    }

    CmdArc** lIter = mOutgoingArcs.begin();
    while (lIter != mOutgoingArcs.end())
    {
        (*lIter++)->reserve();
    }

    if (mPrefireFunction.mCode)
    {
        PyEval_EvalCode(mPrefireFunction.mCode, ScnGlobal::sDictionary, mPyDict);
    }

    DWORD lTime = mDuration - mDuration*(1.0-mProbability)*rand()/(RAND_MAX*1.0);
    mPyObject.mEndTime = T + lTime;
    mFirecount++;
}

void CmdTransition::endOccurence(int iStamp)
{
    if (mPostfireFunction.mCode)
```

```
    {
        PyEval_EvalCode(mPostfireFunction.mCode, ScnGlobal::sDictionary, mPyDict);
    }

    CmdArc** lIter = mOutgoingArcs.begin();
    while (lIter != mOutgoingArcs.end())
    {
        (*lIter++)->commit();
    }

    if (mEnabled == (mOutgoingArcs.size() + mIncomingArcs.size()))
    {
        mState = kENABLED;
        mCmdNet.addReadyTransition(mLink);
    }
    else
    {
        mState = kDISABLED;
    }
}

bool CmdTransition::doOccurence(int iStamp)
{
    if (mInfireFunction.mCode)
    {
        PyEval_EvalCode(mInfireFunction.mCode, ScnGlobal::sDictionary, mPyDict);
    }

    return mPyObject.mEndTime < T;
}
```

```
class CmdPlace : public ScnNode
{
    public:

            CmdPlace(CmdPage& iInstance);

        void update(CmdPlaceControlColor& iColor);
        void update(CmdPlaceDataColor& iColor);
        void update(CmdPlaceSensorColor& iColor);

        virtual void addArc(CmdArcInterface& iArc);
        virtual void remArc(CmdArcInterface& iArc);

        CmdPage& mPage;

    protected:

            COLContainer<CmdArcInterface*> mOutgoingArcs;
            COLContainer<CmdArcInterface*> mIncomingArcs;
};
```

```
CmdPlace::CmdPlace(CmdPage& iPage)
: ScnNode()
, mOutgoingArcs(5)
, mIncomingArcs(5)
, mPage(iPage)
{
}

void CmdPlace::addArc(CmdArcInterface& iArc)
{
    switch (iArc.mDirection)
    {
    case kToTransition:
        mOutgoingArcs.put(&iArc);
        break;
    case kToPlace:
        mIncomingArcs.put(&iArc);
        break;
    }
}


void CmdPlace::remArc(CmdArcInterface& iArc)
{
    switch (iArc.mDirection)
    {
    case kToTransition:
        mOutgoingArcs.rem(&iArc);
        break;
    case kToPlace:
        mIncomingArcs.rem(&iArc);
        break;
    }
}

void CmdPlace::update(CmdPlaceControlColor& iColor)
{
    CmdArcInterface** lArc = mOutgoingArcs.begin();
    while (lArc != mOutgoingArcs.end())
    {
        (*lArc++)->update(iColor);
    }

    CmdArcInterface** lArc = mIncomingArcs.begin();
    while (lArc != mIncomingArcs.end())
    {
        (*lArc++)->update(iColor);
    }
}
```

```
void CmdPlace::update(CmdPlaceDataColor& iColor)
{
    CmdArcInterface** lArc = mOutgoingArcs.begin();
    while (lArc != mOutgoingArcs.end())
    {
        (*lArc++)->update(iColor);
    }

    CmdArcInterface** lArc = mIncomingArcs.begin();
    while (lArc != mIncomingArcs.end())
    {
        (*lArc++)->update(iColor);
    }
}

void CmdPlace::update(CmdPlaceSensorColor& iColor)
{
    CmdArcInterface** lArc = mOutgoingArcs.begin();
    while (lArc != mOutgoingArcs.end())
    {
        (*lArc++)->update(iColor);
    }

    CmdArcInterface** lArc = mIncomingArcs.begin();
    while (lArc != mIncomingArcs.end())
    {
        (*lArc++)->update(iColor);
    }
}
```

```
class CmdArc : public CmdArcInterface
{
    public:

        CmdArc(CmdNet& iCmdNet,
               CmdPage& iPage,
               CmdPlaceInterface& iPlace,
               CmdTransition& iTransition);
        CmdArc(CmdNet& iCmdNet,
               CmdPage& iPage,
               CmdTransition& iTransition,
               CmdPlaceInterface& iPlace);

        void update(CmdPlaceDataColor& iColor);
        void update(CmdPlaceControlColor& iColor);
        void update(CmdPlaceSensorColor& iColor);

        void update(CmdArcDataColor& iColor);
        void update(CmdArcControlColor& iColor);
        void update(CmdArcSensorColor& iColor);

        enum EState
        {
            kENABLED,
            kDISABLED,
            kFIRE
        };

        CmdTransition& mTransition;
        CmdPlaceInterface& mPlace;

    protected:

        BITMap<CmdArcColor*> mColors;

        CmdNet& mCmdNet;
        EState mState;
        bool mPending;
        BITSet mDisabled;

        void remove();
        void reserve();
        void commit();

        void update();
};
```

```
CmdArc::CmdArc(CmdNet& iCmdNet,
               CmdPage& iPage,
               CmdPlaceInterface& iPlace,
               CmdTransition& iTransition)
: CmdArcInterface(iPage, kToTransition)
, mColors(0)
, mTransition(iTransition)
, mPlace (iPlace)
, mCmdNet(iCmdNet)
, mState (kENABLED)
{
    mTransition.addArc(*this);
    mPlace.addArc(*this);
}

CmdArc::CmdArc(CmdNet& iCmdNet,
               CmdPage& iPage,
               CmdTransition& iTransition,
               CmdPlaceInterface& iPlace)
: CmdArcInterface(iPage, kToPlace)
, mColors(0)
, mTransition(iTransition)
, mPlace (iPlace)
, mCmdNet(iCmdNet)
, mState (kENABLED)
{
    mTransition.addArc(*this);
    mPlace.addArc(*this);
}

CmdArc::~CmdArc()
{
    mTransition.remArc(*this);
    mPlace.remArc(*this);
}

void CmdArc::remove()
{
    mColors.begin();
    while (! mColors.end())
    {
        int lIndex = mColors.next();

        if (dynamic_cast<CmdArcDataColor*>(mColors.getContent(lIndex)))
        {
            CmdArcDataColor* lArcColor = mColors.getContent(lIndex);
            CmdPlaceDataColor* lPlaceColor = mPlace.getColor(lIndex);
            lPlaceColor->remove(*lArcColor);
        }
```

```
        else if (dynamic_cast<CmdArcControlColor*>(mColors.getContent(lIndex)))
        {
            CmdArcControlColor* lArcColor = mColors.getContent(lIndex);
            CmdPlaceControlColor* lPlaceColor = mPlace.getColor(lIndex);
            lPlaceColor->remove(*lArcColor);
        }
    }
}

void CmdArc::reserve()
{
    mColors.begin();
    while (! mColors.end())
    {
        int lIndex = mColors.next();
        if (dynamic_cast<CmdArcDataColor*>(mColors.getContent(lIndex)))
        {
            CmdArcDataColor* lArcColor = mColors.getContent(lIndex);
            CmdPlaceDataColor* lPlaceColor = mPlace.getColor(lIndex);
            lPlaceColor->reserve(*lArcColor);
        }
        else if (dynamic_cast<CmdArcControlColor*>(mColors.getContent(lIndex)))
        {
            CmdArcControlColor* lArcColor = mColors.getContent(lIndex);
            CmdPlaceControlColor* lPlaceColor = mPlace.getColor(lIndex);
            lPlaceColor->reserve(*lArcColor);
        }
    }
}

void CmdArc::commit()
{
    mColors.begin();
    while (! mColors.end())
    {
        int lIndex = mColors.next();
        if (dynamic_cast<CmdArcDataColor*>(mColors.getContent(lIndex)))
        {
            CmdArcDataColor* lArcColor = mColors.getContent(lIndex);
            CmdPlaceDataColor* lPlaceColor = mPlace.getColor(lIndex);
            lPlaceColor->insert(*lArcColor);
        }
        else if (dynamic_cast<CmdArcControlColor*>(mColors.getContent(lIndex)))
        {
            CmdArcControlColor* lArcColor = mColors.getContent(lIndex);
            CmdPlaceControlColor* lPlaceColor = mPlace.getColor(lIndex);
            lPlaceColor->insert(*lArcColor);
        }
    }
```

```
}

void CmdArc::update(CmdArcDataColor& iColor)
{
    CmdPlaceDataColor* lColor = mPlace.getColor(iColor.mIndex);

    if (lColor->enables(iColor, mDirection))
    {
        mDisabled.clrBit(iColor.mIndex);
    }
    else
    {
        mDisabled.setBit(iColor.mIndex);
    }

    update();
}

void CmdArc::update(CmdArcControlColor& iColor)
{
    CmdPlaceControlColor* lColor = mPlace.getColor(iColor.mIndex);

    if (lColor->enables(iColor, mDirection))
    {
        mDisabled.clrBit(iColor.mIndex);
    }
    else
    {
        mDisabled.setBit(iColor.mIndex);
    }

    update();
}

void CmdArc::update(CmdArcSensorColor& iColor)
{
    CmdPlaceSensorColor* lColor = mPlace.getColor(iColor.mIndex);

    if (lColor)
    {
        if (lColor->enables(iColor, mDirection))
        {
            mDisabled.clrBit(iColor.mIndex);
        }
        else
        {
            mDisabled.setBit(iColor.mIndex);
        }
    }
```

```
    update();
}

void CmdArc::update(CmdPlaceDataColor& iColor)
{
    CmdArcDataColor* lColor = mColors.getContent(iColor.mIndex);

    if (lColor)
    {
        if (iColor.enables(*lColor, mDirection))
        {
            mDisabled.clrBit(iColor.mIndex);
        }
        else
        {
            mDisabled.setBit(iColor.mIndex);
        }
        update();
    }
}

void CmdArc::update(CmdPlaceControlColor& iColor)
{
    CmdArcControlColor* lColor = mColors.getContent(iColor.mIndex);

    if (lColor)
    {
        if (lColor && iColor.enables(*lColor, mDirection))
        {
            mDisabled.clrBit(iColor.mIndex);
        }
        else
        {
            mDisabled.setBit(iColor.mIndex);
        }
        update();
    }
}

void CmdArc::update(CmdPlaceSensorColor& iColor)
{
    CmdArcSensorColor* lColor = mColors.getContent(iColor.mIndex);

    if (lColor)
    {
        if (lColor && iColor.enables(*lColor, mDirection))
        {
            mDisabled.clrBit(iColor.mIndex);
        }
```

```
        else
        {
            mDisabled.setBit(iColor.mIndex);
        }
        update();
    }
}

void CmdArc::update()
{
    if (! mDisabled.getMask())
    {
        if (mState == kDISABLED)
        {
            mState = kENABLED;
            mTransition.enable();
        }
    }
    else
    {
        if (mState == kENABLED)
        {
            mState = kDISABLED;
            mTransition.disable();
        }
    }
}
```