Virtual prototyping of automated manufacturing systems with Geometry-driven Petri nets

Jens H. Weber-Jahnke a,*, Jochen Stier b

a Department of Computer Science, University of Victoria, PO Box 3055, Victoria V8W3P6, BC, Canada
b Geist3D Software Labs, Victoria, BC, Canada

A R T I C L E   I N F O

Article history:
Received 13 December 2008
Accepted 17 June 2009

Keywords:
Robotics systems
Virtual prototyping
Geometry-driven Petri nets
Simulation
Early testing

A B S T R A C T

The design process of automated manufacturing systems typically involves physical prototypes to validate the interactions between hardware and software components. However, physical prototyping is expensive and time consuming, which often leads to insufficient opportunities for testing early during the development cycle. Our objective is to improve this situation by providing a method to develop realistic prototypes using virtual reality technology that can be applied during earlier development stages. Our approach combines a virtual reality engine capable of enacting the laws of rigid body physics with a new hybrid software modelling language to control the simulated hardware using virtual sensors and actuators as they would be present in a physical prototype. The new modelling language is called Geometry-driven Petri nets (GPN) and combines a class of timed, high-level Petri nets with data structures used in state-of-the-art VR environments. This article describes the new GPN approach, applies it to a case study of an automated manufacturing line, and compares it with related approaches.

© 2009 Elsevier Ltd. All rights reserved.

1. Introduction

The design of automated manufacturing systems (AMS) encompasses control software development activities as well as hardware development activities. Integrating these activities effectively has been a long-term research challenge. Verification and validation of AMS have traditionally relied largely on physical prototyping. This process is illustrated by the shaded elements in Fig. 1. Physical prototyping is costly and time consuming and a lack of early availability of physical prototypes often leads to insufficient opportunities for testing the overall system until late in the development life cycle [1]. However, the cost of correcting defects at a late stage of development can be orders of magnitude higher than if they had been discovered earlier [2]. Costly iteration may occur as indicated with black arrows in Fig. 1.

There have been efforts to provide methods and tools that facilitate the detection of software and hardware defects earlier in the life cycle, i.e., at the design stage. These methods often encompass some forms of virtual prototypes and their analysis (cf. dashed element in Fig. 1). On the hardware side, computer-aided design (CAD) tools have been equipped with ways to create animated three-dimensional (3D) simulations of the system in action. These simulations are typically based on scripted processes and 3D graphics rendering engines, e.g., [3,4]. On the software side, computer-aided software engineering (CASE) tools have been equipped with hybrid modelling languages, which can model discrete processes controlling continuous functions, e.g., [5]. Both approaches have their trade-offs. On the one hand, the scripts running 3D simulations mainly serve visualization purposes, but fail to consider the reactive properties of the AMS control software, including sensor-based actions and actuator-based manipulation of the physical environment. On the other hand, hybrid process models require designers to estimate and encode many abstract assumptions about the physical environment (e.g., the duration of a movement) that may later turn out to be unrealistic. Wrong assumptions lead to subsequent problems with the construction of the AMS [6]. Moreover, the need to encode assumptions about physical properties adds to the complexity of hybrid models.

Our objective is to combine the strengths of both approaches, integrating realistic virtual simulations of the physical environment with a hybrid language to model the control software, in order to reduce the number of abstract assumptions that have to be modelled. This method provides a more realistic virtual prototyping & testing of AMS, as indicated by the dashed element in Fig. 1. We have developed a virtual reality (VR) engine capable of realistically enacting the laws of physics and a hybrid modelling language controlling the physical devices using (virtual) sensors and (virtual) actuators as they would be present in a physical prototype. The developed hybrid modelling language is called Geometry-driven Petri nets (GPN) and combines a form of timed, high-level

* Corresponding author. Tel.: +1 250 721 8797; fax: +1 250 853 3168.
E-mail addresses: jens@uvic.ca (J.H. Weber-Jahnke), jochen@geist3d.org (J. Stier).

0010-4485/$ – see front matter © 2009 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cad.2009.06.012
Petri nets with Scene Trees, a data structure used in current VR tools. GPNs adopt the idea of resource-oriented modelling [7] by adding geometric objects and their behaviours as resources. Resource-oriented Petri nets (ROPN) have been shown to be more compact and less complex compared to the traditional process-oriented modelling methodology, however they have been criticized for their lack of intuitiveness. Wu attributes this lack of intuitiveness of ROPNs to the need to use token colours [8]. Coloured Petri nets are less intuitive because token colours commonly have an abstract nature. Our GPN approach assigns concrete physical (geometric) meaning to token colours and, thus, improves this situation. Another benefit of the resource-oriented decomposition of software control models according to geometric objects is that it provides designers with a notion of composable and reusable mechatronic components that encapsulate physical hardware as well as software control features. This is a significant improvement over the current practice of componentizing hardware models separately from software models [9]. Our approach was implemented in an integrated development tool called Geist3D [10]. We have evaluated our approach using several case studies, including AMS and other automated mechatronic systems. Geist3D conforms to standards and can import models from other CAD and Virtual Reality design tools. Currently supported import formats include the Virtual Reality Modelling Language (VRML) and the Autodesk .3ds file format, with further import filters under development, e.g., for SolidWorks. Geist3D has been made available on sourceforge under an open source license and we invite others to use and evolve it.

The rest of this paper is structured as follows. Section 2 starts with an introduction to hybrid modelling languages, after which Section 3 introduces Geometry-driven Petri nets. Section 4 describes the physics engine used in our simulation environment. We apply our approach to a case study of a automated manufacturing line in Section 5. Section 6 contains an evaluation of our approach. Section 7 provides further references to related work and Section 8 offers conclusions.

2. Design and simulation of automated manufacturing software

Manufacturing control softwares are driven by sensory telemetry about the physical world and use actuators to manipulate objects in the physical world. Fig. 2 shows this general control loop of such a system [11]. Typical approaches to simulating automated manufacturing systems (AMS) do not enact this control loop, but execute scripted stochastic processes. This approach bares the danger of scripting assumptions that may turn out to be unrealistic in the real world. Therefore, our approach is to truly design and simulate the control loop in Fig. 2. This requires a method capable of modelling the physical world to drive the sensor telemetry and to execute hybrid control logic at the same time. Sensor telemetry causes state changes in the software process, which in turn alters the state of the mechanical components via actuators. In order to test the control software, it is necessary to maintain a correct cause-and-effect relationship between the actuators and the values recorded by the sensors. Our approach artificially recreates the feedback loop between a control software and the environment — a loop that is inherently present in physical prototypes due to natural processes.

2.1. Hybrid modelling languages

Hybrid modelling languages are used to design interdependent discrete and continuous processes [5]. Good examples are automated manufacturing processes which assemble products from components. At a high level of abstraction, the process consists of sequences of discrete steps which often have to be synchronized. At a more detailed level, continuous control processes in the manufacturing plant consist of timed tasks such as transportation, welding or milling. Hybrid modelling languages combine a language for continuous functions with the constructs of a discrete-event language.

Hybrid modelling languages have emerged from formalisms that were originally designed to express purely discrete processes or purely continuous processes, respectively. Examples for discrete modelling languages are based on state machines [12]. Petri nets [13] of graph replacement systems [14]. These languages have been further evolved to describe timed discrete-event systems in which state transitions occur over periods of time. Time periods generally model continuous processes in the physical world leading from one state to the next. In addition to time, Petri nets and state machines have also been extended using continuous flow along the edges, as in hybrid Petri nets [15] and Hybrid Automation [16].

Another class of hybrid modelling languages is based on the continuous modelling theory of Bond Graphs [17]. Bond Graphs are 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 express the continuous flow in electrical, mechanical, thermal or hydraulic systems. For example, an electric 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.

Hybrid modelling languages like the ones mentioned above require designers to state assumptions about the electro-mechanical systems behaviour in the physical world, e.g., the time it takes to move from A to B. Specifying these assumptions correctly requires expertise and effort, and they add complexity to the overall software model. This is where our approach is different. Rather than attempting to code assumptions about properties and behaviour of the physical world, we infer these properties by simulating the behaviour of the physical world in a virtual reality environment. Hardware models are available as a result of the computer-aided design process and can be readily imported from CAD tools. If sensors and actuators are embedded in these models, they can effectively be integrated with software control models for the purpose.
of designing and testing AMS control software. We present this approach in detail in the next section.

3. Geometry-driven Petri nets

3.1. Foundation: Coloured Petri nets

The hybrid modelling language developed for our approach is called Geometry-driven Petri nets (GPN) and has been developed on the foundation of Coloured Petri Nets (CPN) [18]. A CPN is a directed graph consisting of two different node types called places and transitions. Under the right conditions, a transition can occur and consume tokens from its incoming places and deposit new tokens on the outgoing places. Tokens can have different types, also referred to as colours. Places have limited capacity for carrying tokens of specific colours. Arcs in the CPN graph have weights that define how many tokens of any colour are consumed respectively produced when a transition occurs. The marking of the CPN refers to the set of tokens currently residing on all the places. As transitions occur they subsequently change the marking from one state to the next.

Formally, the signature of a CPN is defined by a tuple \((\Sigma, P, T, A, c, k, e, m_0)\), with

- \(\Sigma\): a set of types called a colour set,
- \(P\): a finite set of places,
- \(T\): a finite set of transitions,
- \(A\): a finite set of arcs \(A \subseteq (P \times T) \cup (T \times P)\),
- \(c\): a colour function \(c : P \cup T \mapsto \varphi(\Sigma)\), with \(\varphi(\ldots)\) denoting the powerset of its argument,
- \(k\): a place capacity function \(k : P \mapsto M(\Sigma)\), with \(M(\ldots)\) denoting the multiset of its argument,
- \(e\): an arc weight function \(e : A \mapsto \mathbb{N}\),
- \(m_0\): the initial marking \(m_0 : P \mapsto M(\Sigma)\), and \(\forall p \in P : m_0(p) < k(p)\).

The fact that transitions occur in parallel makes CPNs a good choice for modelling concurrent processes. Tokens can be used for synchronization. They can also be used to express the flow of resources in processes. CPN have a formal mathematical basis which supports a rich set of analyses.

3.2. Extending and applying CPN to AMS processes

Many researchers have used extended forms of CPNs for designing and analyzing mechatronic system processes, e.g., [19,20,8,5,7]. Typical extensions include temporal, stochastic, as well as continuous properties [5]. Other research has investigated different decomposition paradigms for designing AMS processes with CPNs. Two principal decomposition paradigms are the process-oriented design and the resource-oriented design, respectively [8]. Resource-oriented decompositions have several advantages, as they have been shown to be more compact and help to avoid potential deadlocks in equivalent process-oriented networks [8]. At the same time, they are perceived less intuitive and harder to understand than their process-oriented counterparts, because they rely on the use of abstract token colours.

The notion of Geometry-driven Petri nets (GPN) introduced in this article adopts a resource-oriented decomposition paradigm and uses several CPN extensions introduced by others. The original contribution of GPNs consist of their tight integration with physical (geometrical) objects in a virtual reality simulation of the physical world. This integration not only makes the use of resource-oriented CPNs more intuitive (i.e., abstract token colours are replaced by concrete physical objects), but also provides a way to realistically simulate the control loop presented in Fig. 2. We have chosen to represent physical objects in CPNs with Scene Trees, a data structure that has useful properties for VR simulations. The next section provides a brief introduction to Scene Trees, followed by an explanation of how Scene Tree objects integrate with the CPN structure.

3.3. Representing objects with Scene Trees

A Scene Tree is a data structure that organizes the information about 3D objects simulated in a computer system. In such a tree, a hierarchy of differently typed nodes relates artifacts such as the geometric description of surfaces and material properties, as well as visual attributes such as colours and textures. The rendering process generates an image from the Scene Tree by traversing it. 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.

In this article, the concept of a Scene Tree can sufficiently be introduced by giving an example rather than discussing a formal definition, which can be found in [21]. Fig. 3 shows a Scene Tree of a salt 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 salt 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 salt inside.

The example above demonstrates how a Scene Tree reflects structural dependencies among objects. The salt dispenser resting on the chair is represented by a node which is a descendant 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 dispenser is changed. This indicates how actuators can be realized: they change transformation nodes or rewrite entire parts of the Scene Tree structure.

Sensors can be added as additional tree nodes encapsulating a tree traversal that computes the sensor value. We have implemented a collection of reusable sensors, e.g., for collision detection, proximity and path finding. The latter one relies on gradient fields to model paths guiding autonomous robots through space [22]. We will discuss an example for using gradient sensors later in this paper.

3.4. GPNs: Integrating Scene Trees with CPNs

We have established so far that the GPN approach considers control process models (CPN) as well as physical models (Scene
Tree). We will now explain how these models interact. Basically, in a GPN, physical structures (sub-trees) of a Scene Tree can appear as tokens in the Petri net, while sensors appear as transitions.

3.4.1. Actuators: How the control process influences the VR simulation

The CPN implements actuators by modifying Scene Tree structures using inscriptions. Inscriptions are actions that are triggered upon certain conditions. They can read and change any property value of this object token, i.e., Scene Tree element. The Scene Tree data structure has a catalogue of possible properties types, e.g., dynamic properties such as velocity, direction, friction, masses and visual properties such as colour and texture. Moreover, the Scene Tree data structure has a set of transformational operations, e.g., attach and detach, which can be used to create and dissolve joints between physical objects.

We distinguish between place inscriptions and three types of transition inscriptions, namely pre-fire inscriptions, in-fire inscriptions and post-fire inscriptions. As the name indicates, place inscriptions are associated with places in the CPN and triggered upon appearance of an object token of a particular type at that place. Pre-fire and post-fire inscriptions are associated with transitions. They are executed right before and after a transition occurs, respectively. In-fire inscriptions are executed repeatedly according to specified sampling intervals while a transition is occurring.

Consider the example of a robotic arm in Fig. 4. The left side shows a visualization of the hardware model contained in the Scene Tree, while the right side visualizes the CPN that controls its function. Note, that in this Petri net visualization, boxes represent transitions and circles represent places. Filled circles indicate places carrying tokens and filled boxes indicate enabled transitions. The direction of arcs is indicated by thick endpoints of the connecting lines.

The effector of the arm is positioned by using inscriptions to change the angular velocities around the joint axes. The effector is designed to pickup boxes using vacuum suction. This action is actuated in the Scene Tree by adding a temporary 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 destination during delivery, e.g., to a conveyor belt.

The right side of Fig. 4 displays the CPN 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 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.

The Petri net visualization in Fig. 4, a screen shot from our tool, does not discriminate between different tokens in order to simplify the layout and avoid clutter. A more detailed view, called colour set, is available when a particular place or transition is selected in the tool. Fig. 5 displays the colour sets of transitions deliver and drop. Each transition consumes and produces the tokens representing the robotic arm (vertical joint, horizontal joint, effector) and a box, which is going to be handled by the robot. The colour set is organized in a 2D array in order to uniquely identify tokens in the inscriptions. The slots b0, b1 and b2 are typed slots, i.e., they contain tokens of particular type (vertical, horizontal, effector). 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. This allows for different types of boxes to be part of the manufacturing process.

Table 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 inscription of transition drop performs the reverse operation. The function attach creates a fixed joint between the two bodies and the function detach destroys the joint. Inscription syntax is based on a purely functional subset of Python. This means that loops and iterations are not allowed in inscriptions in order to keep the analytical properties of CPNs.

3.4.2. Sensors: How the VR simulation influences the control process

Sensors appear as special objects in the Scene Tree. In contrast to regular Scene Tree objects, sensor objects do not appear as tokens in the CPN but as guarded transitions. The occurrence of guarded transitions is conditional on external events, i.e., it is based on the sensor telemetry. For example, consider the robot arm in Fig. 4 being deployed in a conveyor belt setting. In this case, boxes would travel on the conveyor belt to the robot arm to be picked up
and delivered. In this case, the transition connecting place \textit{buffer} and place \textit{P2} may be guarded by a proximity sensor at the “pickup position” of the robot.

### 3.5. Timed transitions

Timed transition occurrences were originally introduced by Ramchandani in order to describe and analyse timed dynamical systems more effectively [23]. In GPNs, the durations are used to execute discrete time step algorithms. 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 is reserved on the outgoing places so that enough space remains in order to deposit tokens at the end of the occurrence interval.

Consider the transition between places \textit{buffer} and \textit{P2} in Fig. 4 as an example application for a timed transition. In the previous section, we assumed that a conveyor belt would transport boxes to the pickup position. In that case, the transition between \textit{buffer} and \textit{P2} would be guarded by a sensor detecting the arrival of a box. In an alternative setup, we may have human labourers taking boxes from the buffer and placing them on the machine pickup position. In this case, the transition between \textit{buffer} and \textit{P2} is not guarded by a sensor but it is timed with a specified duration, in order to capture the delay it would take a human to service the machine.

An alternative extension to Petri nets could apply a notion of time to places instead of transitions [24]. In this case, tokens \textit{submerge} in a place and remain hidden to the outgoing arcs until they \textit{emerge} again at the end of the duration. Both approaches are largely equivalent and we decided to implement to most common of them, namely timed transitions.

### 3.6. Concurrency conflicts

Concurrency conflicts arise when multiple transitions are enabled to occur but their occurrence would be mutually exclusive because they are competing for the same resource, i.e., they are about to consume the same token from incoming places or allocate the same capacity in outgoing places. Such conflicts need to be resolved in a Petri-net-based simulation environment. Common approaches to resolve such conflicts are either deterministic (e.g., based on assigned priorities) or non-deterministic (e.g., based on assigned probabilities). In practice, both approaches are needed. We have implemented a hybrid approach that allows GPN developers to assign priorities as well as probabilities. A conflict between enabled transitions is resolved by first selecting those transitions with maximum priority. If several such transitions exist, a random number is generated and the conflict is resolved based on their assigned probabilities.

### 3.7. A unified component concept

In order to facilitate reuse and manage complexity in Industry-scale systems, it is important to support a notion of components and component composition. Ideally, this concept should encapsulate and unify control software and physical hardware aspects. For example, a model of a robot arm may be reused in its entirety, including hardware and control software. We realize this objective in GPNs by introducing the concept of \textit{pages}. A page encapsulates a CPN (software model) and a Scene Tree substructure. The page has an interface made up of selected elements of the encapsulated CPN. This idea is based on the general notion of hierarchical Petri nets [25]. Place refinement and transition refinement are alternative forms of hierarchical Petri nets, where pages publish only places or transitions at their interfaces [26]. Our GPN approach is based on place refinement.

The CPN within a page looks like a normal Petri net, except for the addition of \textit{references} attached to selected places (cf. right side of Fig. 6). From the outside, a page appears as a large rectangle surrounded by one \textit{link}, or hole, for each reference as depicted on the left of Fig. 6. 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. Consequently, a GPN page is defined as a tuple \((N, S, R)\), where \(N\) is a CPN, \(S\) is a Scene Tree structure and \(R\) is an interface that consists of a subset of the places in \(N\).

The introduction of pages requires several changes to the formal definitions of the Petri net structure, since pages can be reused in GPN models in lieu of “regular” places.

Reusing a page is simply done by copying an existing one and pasting it to a new design. This type of reuse method is often referred to as \textit{template}-based reuse. Compared to \textit{type}-based reuse, template-based methods are simpler to implement in a tool and also simpler to understand from a user’s point of view. They are used in most CAD tools today. A drawback of template-based reuse is that it does not support verification of consistency relationships between reused components.

### 4. Making it real: The physics engine

Mechanical systems naturally depend on properties such as friction, masses and forces. The laws of physics are important factors when it comes to a realistic simulation and test of automated manufacturing systems. Control logic often has to monitor physical quantities. Physics-based animation libraries – also called \textit{physics engines} – are now emerging as software components that simulate rigid body dynamics in terms of geometries, masses, kinematic relationships and friction. Most engines support different types of mechanical joints, including hinges, ball bearings, linear and universal joints.

The majority of the physics engines are based on discrete time step algorithms. 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. 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.

Our approach integrates the Open Dynamics Engine (ODE), which is freely available under an Open Source license [27]. 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 [28]. The dynamics engine is integrated via geometric node types in the Scene Tree, e.g., 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. A joint implicitly
connects its parent and child nodes. When a joint is encountered, then optional linear or rotational forces can be applied. As a result, the positions and orientations of the bodies is being updated in each time step. Surface attributes of objects can be defined to influence frictional forces.

5. Case study: Production line

We use this section to demonstrate the GPN approach to model and test control software for a material handling system designed to paint boxes. This case study was chosen because it displays many characteristics of an industrial manufacturing system [29], while still being simple enough to be covered in this article. The manufacturing process requires different concurrent tasks in order to control numerous devices and track the flow of resources.

The model shown in Fig. 7 originated from the Technical University of Munich (TUM), Germany. We imported the TUM physical model into the scene graph using the Virtual Reality Modelling Language (VRML), a format that is supported by our tool, Geist3D. The original TUM model had been animated using key-frame sequences. Our physics engine provides a more realistic simulation by interpreting rigid body dynamics and forces instead of pre-programmed animation sequences. In order to use it, we need to add some further hardware properties to the TUM model, including mechanical joints, masses and coefficients of friction.

The system contains two painting stations, three conveyors and a robotic arm. The arm 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. 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 colour 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 the software control model is to implement an efficient production process, 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 GPN 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.

5.1. Painting station

5.1.1. Physical hardware

Designing control software to handle the boxes in front of the painting station using a physics-based simulation is more realistic (and challenging) than designing a model that abstracts from the laws of physics. Using rigid body dynamics, the boxes may move in unanticipated ways 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 have 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 Fig. 8. When a stop is lowered, it is level with 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.

The simulation further revealed 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 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.

5.1.2. Control software

The control software model (CPN) of the painting station is shown in Fig. 9. Place P2 contains five resource tokens 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 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.

Place P1 contains a token representation of the box arriving at the colour-sensitive collision sensor that is a part of the configuration 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.

5.2. Production line

We will now show how the concept of GPN pages is used for component-based reuse and composition of larger applications. Fig. 10 displays the CPN that models the entire production line depicted in Fig. 7. It contains two instantiations of the painting station page, 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.

6. Evaluation

6.1. Reactive systems

The case study discussed in the Section 5 exhibits many properties of a typical AMS. We use it to demonstrate the capability of our approach to design reactive simulations of AMS, i.e., simulations in which sensors gather telemetry from an interpretation of a virtual physical world, and in which actuators influence this virtual reality. Our ability to enact this control loop (cf. Fig. 2) is the main novelty of our work compared to related approaches, which use scripted simulations or non-reactive process models. The benefit of being able to design and prototype reactive AMS becomes even more evident if we consider labourer participation and intervention in the manufacturing process. Bzymek simulates the design of such a manufacturing system from the aerospace industry, including labourers interacting with eighteen partially automated machines [3]. The design and simulation system used for this study is Delmia/Quest, a leading commercial product. Labourers can be modelled in Quest for visualization purposes and to collect statistical variables such as the average distance travelled by them. Bzymek reports problems with the collection of meaningful labourer statistics and perform all their simulations without them, suggesting that simulated labourers should be used only for visualization purposes only [3]. Still, Bzymek recognizes that the design of the plant layout and the machining processes impacts the potential for accidents and injuries. The design of AMS typically includes safeguards for accidents such as proximity sensors that detect the presence of humans in dangerous areas. Such designs cannot be simulated with Quest and similar existing systems, since they require reactive behaviour.

The design and simulation of reactive system behaviour becomes even more important with so-called next-generation production paradigms, such as holonic manufacturing systems (HMS) [30] and reconfigurable manufacturing systems (RMS) [31]. A holon is an autonomously acting building block of a manufacturing system used for transforming, transporting, storing and validating objects. Holons have a software/information part as well as a physical processing part. Virtual prototyping of HMS has been challenging because traditional AMS simulation tools do not model reactive agents. In contrast, the concept of GPN pages is capable of encapsulating both parts (software and hardware) of a holonic...
component and our simulation environment can be used to analyse their collaborative behaviour.

An important property of RMS is the ability to adapt and evolve, i.e., to be self-reorganizing. A simple example where such self-organizing behaviour is needed is the adaptation of the routes taken by transport holons due to changing layouts of the machine setup in the manufacturing plant. Researchers have suggested biologically-inspired algorithms as potential solutions, for example the intelligent behaviour exhibited by ant colonies [31]. Real ants lay down pheromones directing each other to resources while exploring their environment. With GPNs and Geist3D, it becomes easy to simulate the behaviour for transport holons. Pheromone levels can simply be represented by another property associated with Scene Tree objects (analogously to colour, texture etc). In the transport example, we specifically consider pheromone levels at sections of the plant floor. Transport holons are equipped with two types of sensors, a collision sensor and a gradient sensor (cf. Fig. 12). The collision sensor detects blocked routes and prevents collisions while the gradient sensor detects the direction of the highest pheromone levels in the local proximity of the holon. Holons leave pheromone markers on floor sections they used. Pheromone levels at sections unused by transport holons decrease over time. Consequently, an initial transport routing layout (and pheromone trail) as indicated on the right side of Fig. 12 will be reconfigured automatically when passage ways are blocked due to evolution in the floor plan of the manufacturing system.

6.2. Cognitivesupport

Design languages may become cluttered and lack cognitive support when they are applied to complex, large-scale applications. In particular, non-hierarchical Petri nets may quickly become complex with larger applications. The hierarchical nature of GPNs provides an adequate abstraction mechanism to hide complexity and build complex applications. This is shown by Fig. 10 demonstrating how designers can compose larger-scale systems while abstracting from the complexities of individual work cells, as presented in Fig. 9. While hierarchical Petri nets with pages and ports have been applied before, an original contribution of our approach consists in the fact that GPN pages not only encapsulate Petri nets but also physical objects (cf. Section 3.7). This means that whenever a predefined page is reused, not only the controlling Petri
net but also the associated physical structures get reused as a single, functional component that can be used to compose even more complex components. While it can be argued that a configuration management system (CMS) can be used to keep track of dependencies between control processes and the corresponding physical hardware, CMS operate as repositories and the notion of a configuration as an encapsulated abstraction typically does not permeate the entire design methodology. Consequently, we argue that GPNs provide better cognitive support by exhibiting a higher abstract gradient, according to the well-known Cognitive Dimension framework for usability evaluations [32]. Green and Petre define an abstraction as “a grouping of elements to be treated as one entity, whether just for convenience or to change the conceptual structure”. The need for abstraction is supported by other studies published. For example, Bzyl and petre report the “very generic Visual Basic-type layout” user interface for designing simulation in Delmia/Quest as highly confusing [3]: “As one is creating hundreds of parts and sub-parts it is extremely easy to miss or overlook one of the countless input variables. The overlooking of a single variable can result in problems ranging from simulation failure to inaccurate results which can be very frustrating.”

Another criteria in the Cognitive Dimension evaluation framework is called Closeness of Mapping. Green and Petre state that program design “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” [32]. We argue that the integration of VR simulations of realistic physical processes with Petri nets controlling these processes provides a closer mapping to the problem world of designing AMS than current approaches that are either fully Petri net based or that are fully based on scripted VR animations.

6.3. Scalability

The scalability of our approach depends on the size and connectivity of the CPN structure as well as the artifacts of the VR environment. The artifacts in the VR environment are triangulated. In practice, the effort spent on processing the triangles for rendering and collision detection eclipses the effort spent on executing the Petri net. Therefore, the runtime performance of the simulation is practically bounded by the complexity of the virtual reality model alone. Geist3D uses depth buffering to render a 3D scene. Each rendering step must first sort all triangles in a scene before they can be painted and sorting n elements takes O(n log n) time. The resulting worst case complexity of O(n log n) indicates that the approach is scalable. Further optimizations can be applied to limit the set of triangles to be sorted, such as occlusion culling and space partitioning. 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. Other rendering techniques that provide a more realistic shading may be less scalable, e.g., ray tracing.

7. Related work

An important characteristic of our approach is that sensor telemetry from the virtual environment is used to simulate the feedback loop between an embedded control software and its physical environment. Valkenaers and Holvoet have highlighted the importance of considering the environment as a "first-class abstraction" during the design of manufacturing control systems [6]. This property stands in contrast to current industrial simulation tools such as the widely used Delmia tool suite [3,33], which can animate scripted visualizations of the environment, but do not simulate true physical processes triggering reactive behaviour in the control software. Like Geist3D, Delmia uses a Scene Tree data structure. It includes a graphic simulation language (GSL) and command line interpreter (CLI) to define animation scripts and automate editing functions. Although a process model is still required to schedule animations, this process does not consider any sensory inputs.

Several other approaches apply VR and physics engines to simulate sensor-driven mechatronic systems. However, they lack a high-level modelling software language and a cohesive component concept. Two examples are Webots and Gazebo. Webots is a commercial 3D simulation tool for mobile robots [34]. Webots includes a predefined set of sensors, actuators and robots. The main difference to our approach is that software is programmed using a C/C++ API rather than being modelled with CPNs. There is no unified component concept encapsulating software and hardware, and only predefined robots can be used. Gazebo is a simulation toolkit developed by the Robotics Research Lab at the University of Southern California [35]. Similar to Webots, Gazebo offers a low-level programming API for software development rather than a hybrid, component-oriented modelling language.

Compared to low-level programming or scripting, the advantage of using hybrid modelling languages is their raised level of abstraction when it comes to dealing with complex systems and the fact that they are mathematically analysable. Our approach is based to a large degree on Petri net theory. CPNs have proven effective in modelling and verifying many types of processing systems [7,8,5]. Our approach of integrating a model of the physical hardware and environment with a CPN has benefits as well as drawbacks. On the plus side, adding the physical dimension provides for a more realistic simulation and test opportunity of the models under development. On the other hand, it limits the analysability of the software model, since the physical simulation environment has to be taken into account. To see this, let us consider the example of a throughput analysis based on cycle times of the painting stations operating in the production line in our case study (cf. Section 5). In a CPN consisting of k conflict-free cyclic sub-nets, the overall cycle time τ is given by the maximum time of (τ₁, τ₂, ..., τₖ), where τᵢ is the cycle time of each sub-net, which is computed by summing the firing durations of the transitions. This metric is easy to compute as long as no sensors are involved. However, as soon as we use a simulation that relies on the sensor/actuator control loop involving an interpretation of the physical environment, this metric is harder to compute analytically and, in general, must be determined using simulations.

We have argued that our approach to virtual prototyping AMS using a realistic enactment of the hardware/software control loop helps to reduce the cost of developing manufacturing systems. This argument is based on earlier research on the life cycle and cost of modular assembly systems, e.g., Heilala points out the value of component-based simulation software with 3D capabilities for the design and configuration of AMS [36].

Our work is related to research on Integration Prototyping (IP) [1], which attempts to combine the benefits of physical prototyping (realism, environment) with the strengths of virtual prototyping (visualization, UI configurability). Jung presents an architecture and middleware to connect virtual prototype components with physical prototype set ups. They demonstrate their approach with a simple setup using Lego technology. The difference to both approaches is that IP adds realism to VP by connecting virtual interfaces to physical prototypes, while we extend the notion of virtual prototyping with a simulation of the physical environment. Both approaches can be combined in an AMS development life cycle transitioning from virtual prototyping using a physics engine to integrated prototyping using real physical components.
8. Conclusions

The main contribution of this research is a tighter integration of hybrid software modelling languages with physical hardware models being simulated in a physics-driven virtual reality environment. Traditional software modelling languages do not incorporate a geometric description of the hardware and variables representing sensors and actuators are either approximated or randomized. Our approach provides an opportunity to rapid prototyping and realistic testing of robotic applications early in the development cycle. It makes Boehm’s vision of the Spiral Model of software development more affordable in the context of robust software development [37]. The Spiral Model has traditionally been applied mainly to pure software projects, whereas mechatronic systems have still largely been developed using a waterfall-oriented approach. This is because prototype construction of systems involving hardware in physical environments is expensive.

We furthermore introduce a unifying component concept that encapsulates software as well as hardware, and provides for reusability and scalable compositions. The approach has been implemented in an integrated development environment (Geist3D) and released into open source.1 Our current tool does not include the ability to generate production code from the software model, since we did not want to focus on a particular target platform (language/API). Nevertheless, automatic and customizable code generation from Petri net models has been well studied in the literature [38]. We consider this feature a future extension.

As discussed in the previous section, using a physics-based engine to drive the development of robotic control software based on hardware geometry and rigid body dynamics adds realism and testability, but it also inhibits application of some analytical tools that have traditionally been used to formally analyse software models. Winning back some of these analysis capabilities is an interesting and important direction for future research.

References


1 http://sourceforge.net/projects/geist3d/.