Accepted for: German-Tunisian Conference on Smart Systems and Devices (SSD)

Negotiation Scenarios between Autonomous Robot Cells in Manufacturing Automation: A Case Study

Torsten Heverhagen, Rudolf Tracht
University of Essen, Germany
FB 12, Industrial Automation
{Torsten.Heverhagen|Rudolf.Tracht}@uni-essen.de

Abstract

Today's industrial manufacturing systems have to face the problem of fast changes in demand of products and the product spectrum. One solution for getting a more flexible structure of production lines is the concept of autonomous and cooperative production units, which is taken from the idea of holonic manufacturing systems [1]. In such systems productions requests are negotiated between production units.
The specification and description of these negotiation is a complex and error-prone task. This paper introduces a negotiation protocol and its specification with the object oriented specification language UML-RT (Unified Modeling Language - Real Time). The paper shows how a scenario-based object oriented development process together with event-driven simulation can help in validating complex specifications at an early development stage.

1. Introduction and Motivation

At the University of Essen an assembly line case study is being developed, which consists of 3 autonomous, cooperative assembly robot cells, a part storage, a product storage, a quality control system, and a transport system. An outline of the case study is shown in Figure 1.
The transport system consists of a belt conveyor on which special pallets are mounted and a PLC. The PLC is not shown in Figure 1. The special pallets are prepared to take up parts and products for transportation. Products consist of parts. In our case study, one product is an electrical light switch for surface mounting. A product of this type, for example, consists of the parts switch box and switch button.
The part storage is located at the beginning of the production line. It consists of a storage area for parts and a palletizing robot. When an assembly robot needs parts, the transport system carries empty pallets to the part storage, tells the part storage to put the needed parts on the pallet and carries the pallet to the assembly robot cell.
The assembly robot cell consists of a part storage area, a product storage area, an assembly area, the robot, and an industrial PC (IPC). The IPC (not shown in Figure 1) manages and controls the assembly robot cell. The object oriented IPC-program is designed with UML-RT. The assembly robot is able to:
(1) negotiate production requests
(2) take parts from a pallet and put it on the part storage area,
(3) assemble the parts to a product, and
(4) put assembled products to a pallet or the product storage area.
When an assembled product should be carried to the quality control system, the IPC-program asks the transport system for an empty pallet and tells it to carry the product to the quality control system.
The quality control system consists of a vision system with camera and image processor. It is responsible for the decision if an assembled product should be carried to the product storage or to the rejects.
The product storage consists of a palletizing robot and a product storage area.
Each assembly robot cell has the same product spectrum. The number of assembly robot cells working simultaneously depends only on the quantity of demanded products. When this quantity exceeds the capacity of all existing assembly cells, it is also possible to extend the manufacturing system with new assembly robot cells or to raise the productivity of existing assembly robot cells with improved components.

In this paper we assume that the number of existing assembly robots is sufficient for the maximum demand of products. For the minimum demand only one assembly robot is needed. The decision of how many robot cells are working simultaneously is a result of negotiations between the autonomous robot cells. There is no higher instance which controls this decision.
The central idea of this negotiation is that assembly robots can dynamically enter and leave the production process whenever it is necessary.
In this paper we propose a negotiation protocol which fulfills the requirements of our case study. Another aspect of this paper is the use of the object oriented analysis and design language UML [2]. Most of the figures in this paper are UML diagrams. Though we explain UML syntax and concepts whenever used it is out of the range of this paper to give a comprehensive introduction into the UML. Some of the UML concepts and diagram types belong to a special UML extension which is called UML-RT [3]. We chose this extension because of its clear separation between interface and implementation of software components. The main concepts of UML-RT are introduced in section 3.

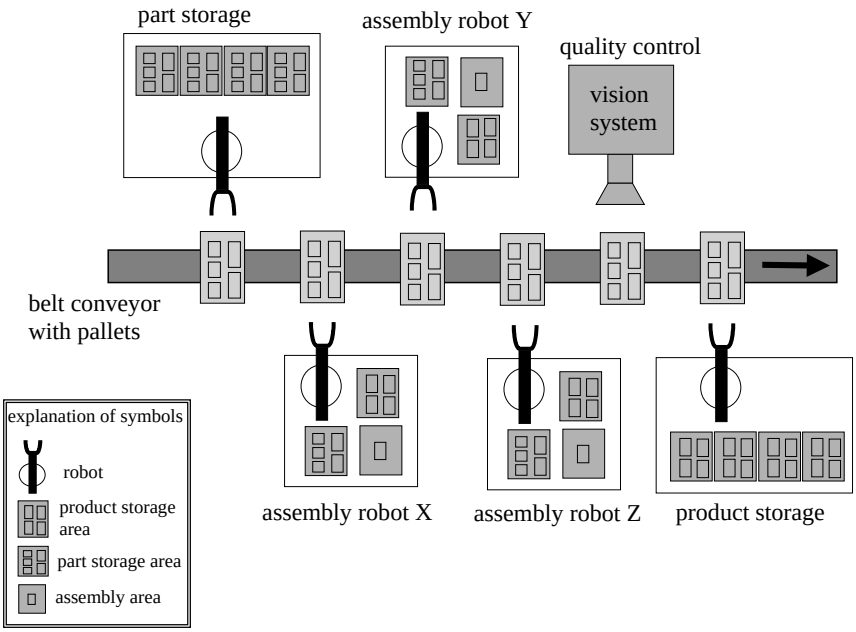

Figure 1. Outline of the assembly line case study

Before we start with describing our negotiation protocol we show how the requirements for our case study are modeled with *use cases* (section 1.1). The overall software architecture of our case study is given in section 1.2 as an *object diagram*. In section 2 we pick the negotiation out of our use cases and describe it more detailed with *sequence diagrams*. Section 3 introduces structural modeling techniques like *class diagrams* and *structure diagrams*. Section 4 uses *statecharts* for modeling behavior of software components. In section 5 the software components introduced in section 3 are discussed in more detail together with considerations taken from section 4. Sections 6 and 7 close the paper with an overview of how the transition to implementation is done and how sequence diagrams can be used to validate the software model against the given requirements.

1.1. Requirements

Requirements are captured in the *use case model*. It consists of *use cases* and *actors*. A graphical notation is given in Figure 2. Use cases are rendered as ellipses with solid lines. The name is displayed inside or below the use case. Actors are rendered as stick figures with the actors name below the figure.
Actors represent the users of the system to be developed or modeled. In our case study the system is the assembly line and one user is a production planning and scheduling system (PPS). The PPS is modeled as an actor called *PPSActor*. It sends production requests (orders) to the assembly line and receives status messages from the assembly line.
Use cases represent the functionality or services which the system provides for its users (actors). Our

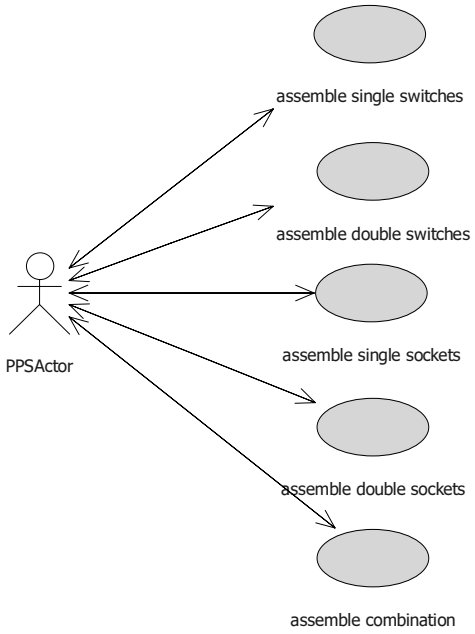

Figure 2. Use case diagram for the case study

assembly line provides the PPS with the functionality to assemble several products. For the production of different products different flows of actions has to be performed by the assembly line. This is the reason why we modeled one use case for each product.
The description of a use case can be done with natural language, formal structured text or pseudocode. The use case model is the foundation for the discussion with the customer about if the system contains all requirements of the customer. That's why it should be understandable and readable for the customer. It is the binding requirement specification for the further development of the system.
Each use case of the case study starts with an order (production request) from the PPS. The first action in each use case is the negotiation of the order.
In this paper we concentrate solely on the negotiation of orders and leave out the main part of the use cases: the assembling of products. But before we come to the negotiations we must introduce the overall software architecture of the system.

1.2. Software Architecture

While the *use case model* describes the overall behavior of the system in term of use cases, the *software architecture* describes the structure of the system in terms of software components, their interface and relationships.
For the purpose of this paper only short information about our software component structure is needed. In Figure 3 an object diagram is given, which shows the top level components and their relationships. The important real world objects of Figure 1 are represented by objects (rendered as rectangle) in Figure 3. Objects are instances of classes. The three assembly robots X, Y, and Z are instances of the class *AssemblyRobot.* They are called robotX, robotY, and robotZ.
The key idea of our software architecture is the use of a *mediator* for the linking of software components. Without a *mediator* we had to establish links between almost every object combination. This would lead to less reusable software components. The *Mediator* is a software design pattern which is described in [4]. It is responsible for the correct relaying of messages between system components.
Not for all objects of Figure 3 a class is specified. In the further discussion of this paper we only need to remember the assembly robots, the mediator and the PPS.

2. Negotiation Scenarios

For each use case exists a comprehensive description of the flow of actions which has to be performed to

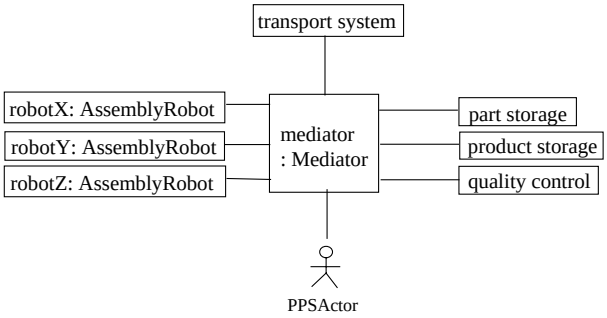

Figure 3. Object diagram for the software architecture

realize the task of a use case. This flow of actions can contain branches and loops. It should consider all possible situations into which the use case may run.
A concrete flow of actions performed under a concrete situation is called a scenario. Scenarios are very helpful in capturing requirements in early design stages.
The simplest scenario in one of our use cases takes place, when the PPS sends an order which is impossible to fulfill. A description of this scenario could look like the following:
"The PPS sends at time 11 p.m. an order to the assembly line about the production of 50 switches within 1 hour. The assembly robots are at this time work overloaded. Every assembly robot rejects the order. One assembly robot tells the PPS about the rejection of the order."
With this rejection the scenario is over. This was a complete scenario through the use case *assemble switches*. Even with this simple scenario several questions arise like: Which messages must be sent between the assembly robots? Which assembly robot sends the rejection to the PPS? If we remember, that assembly robots may enter and leave the production process, how does a robot know, how many participants are in the negotiation?
For considering the first question a special diagram is given in Figure 4, called *sequence diagram*. In sequence diagrams the messages dispatched between participating objects are shown with their time ordering.
The objects displayed in Figure 4 are known from the software architecture of Figure 3. Each column represents an object. The object names are given at the head of each column. The vertical dotted lines below the object names are time lines for each object. The time is increasing from top to down. Horizontal lines with single sided arrows are asynchronous messages. Rectangles at the end of message arrows show that the message is processed by the receiving object.
Conform to the software architecture every message is dispatched over the mediator. At first the order is submitted to each assembly robot with message *OrderSubmission*. Next the robots discuss about the

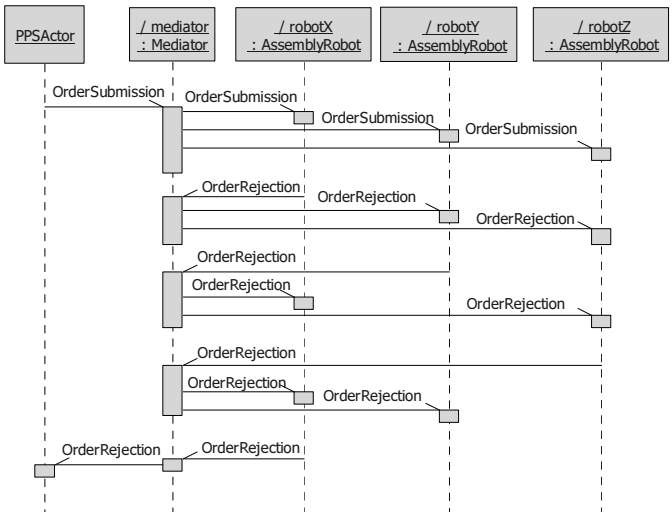

Figure 4. Sequence diagram for the simplest scenario

order. Because no one is able to assemble a switch at this time, every robot sends the message *OrderRejection* to each participant. At last *robotX* sends the *OrderRejection* to the PPS.

Why *robotX*? The decisions made during the negotiation are rule based. One rule is, that if all assembly robots rejected an order, the robot with the highest priority sends the decision to the PPS.

More about rules is given in section 4. Questions that arise from the consideration of scenarios must be answered and documented in the use case descriptions. For most systems it is not possible to describe all scenarios and sequence diagrams. In this section we describe one more scenario for the motivation of the following sections:

"The PPS sends at time 11 p.m. an order to the assembly line about the production of 50 switches within 1 hour. The assembly robots are at this time idle. RobotX sends an offer to robotY and robotZ to tell that it can fulfill the order completely. RobotY also sends an offer to robotX and robotZ. RobotZ is within one hour only able to assemble 30 switches. Therefore it sends an offer for cooperation to robotX and robotY. Because robotX has a higher priority than robotY it gets the order. RobotY and robotZ send an OrderRejection to robotX and each other. RobotX sends an acknowledgement for the order to the PPS."

This scenario is no complete flow through the use case *assemble switches*, because the actions for the assembling of the switches are left out.

The scenario shows, that assembly robots can cooperate in processing an order. One aim of the negotiation is that only a minimum number of assembly robots is used. Higher priority robots are preferred even if they have to cooperate.

The next two sections explain, how structure and behavior of the system is modeled to meet the requirements defined in the use case model.

3. Introducing the Negotiator Capsule

Classes are the most important concept of object orientation. They define the behavior and structure of their instances. Normally classes contain attributes and operations and have relationships with other classes. The UML contains mechanisms to extend or specialize classes. This mechanism is called *stereotyping*. A class that is extended or specialized is called a *stereotype*. Actors and use cases, for example, are stereotypes.

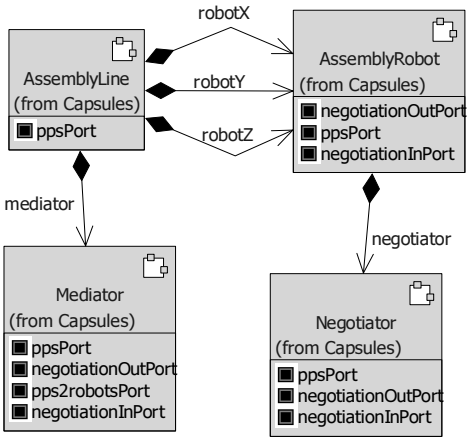

Figure 5. Class diagram for capsules

UML-RT provides three additional stereotypes: *capsules*, *ports*, and *protocols*.

A capsule is an active class. The difference to a normal class is the interface description. While normal classes describe their interfaces with operations (and perhaps attributes and relationships), capsules define ports for this reason.

A port is used by a capsule to send messages to ports of other capsules. A capsule can also contain other capsules. This is modeled by an aggregation relationship. Figure 5 shows four capsules (*AssemblyLine*, *AssemblyRobot*, *Mediator*, and *Negotiator*).

Capsule *AssemblyLine* represents the complete assembly line. It contains three instances (robotX, robotY, and robotZ) of capsule *AssemblyRobot* and one instance of capsule *Mediator* (*mediator*). The port *ppsPort* is used to send and receive messages to and from the PPS.

Capsule *AssemblyRobot* has three ports. Port *ppsPort* is connected over the mediator with the *ppsPort* of the assembly line. Connections of different ports are not shown in class diagrams, but in *structure diagrams* like in Figure 6, which is discussed later.

During negotiation an assembly robot has to send and receive messages simultaneously. For this reason we decided to define two ports for the negotiation,

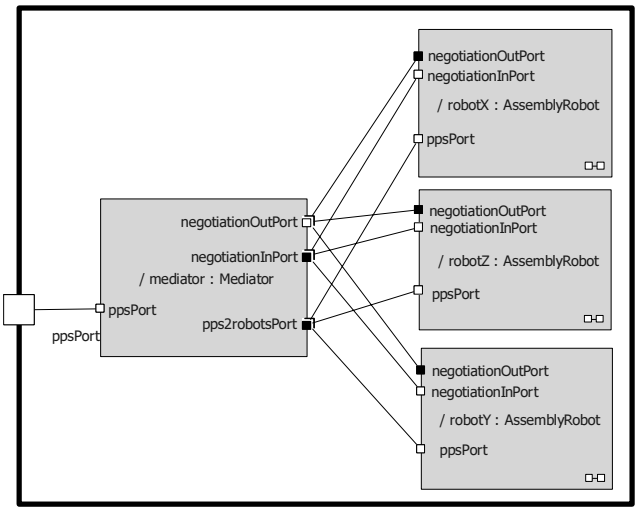

Figure 6. Structure diagram for the assembly line

one for outgoing and one for incoming messages, called *negotiationOutPort* and *negotiationInPort*.

Capsule *Mediator* contains the necessary ports for relaying messages from assembly robots to the PPS and between them.

The capsule *AssemblyRobot* is responsible for very different tasks: the negotiation, the management of accepted orders, and the assembling. In such cases it is a good idea to separate responsibilities into different capsules. So a capsule called *Negotiator* is modeled, to which the assembly robot delegates all responsibilities for the negotiation.

To show how ports are connected with other ports UML-RT introduces a new diagram type: the *structure diagram*. Figure 6 shows the structure of the capsule *AssemblyLine*. A structure diagram is an extended object diagram. Ports of instantiated capsules are connected with solid lines.

The specification of which messages can be sent through ports is done with *protocols*. Protocols define a set of incoming and outgoing messages. This messages are called *signals*. Figure 7 shows a protocol called PPS_Protocol, which contains two incoming and one outgoing signal. Signals *OrderRejection* and *OrderSubmission* are known from Figure 4. This protocol is mapped to all ports called *ppsPort* and *pps2robotsPort*. The other ports are mapped to the *NegotiationProtocol* which is more complex and discussed in section 5.

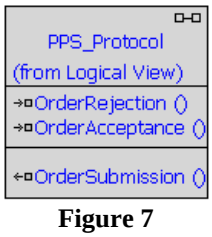

Figure 7

The next section explains how behavior of capsules is modeled. Because we only concentrate on the negotiation, we only have to consider the *Negotiator* capsule.

4. Modeling Behavior of the Negotiator

Until now we have described, how requirements and the structure of the software system are modeled.

Behavior of objects is modeled with statecharts. Different modeling tools for UML provide slightly different notations and capabilities for statecharts. We used the tool “Rational Rose RealTime” [5] for our models, in which the statecharts are similar to ROOM-charts [6], incorporate the programming language C++, and are executable.

Figure 8 shows the top level statechart of the capsule *Negotiator*. It consists of four states and the initial point. The first state *wait_for_order* is a wait state. The transition *receiveOrder* fires, if an *OrderSubmission* is received through port *ppsPort*.

State *collect_ackns* is for the question in section 2, “How many participants are in the negotiation”. Every assembly robot has to acknowledge the order

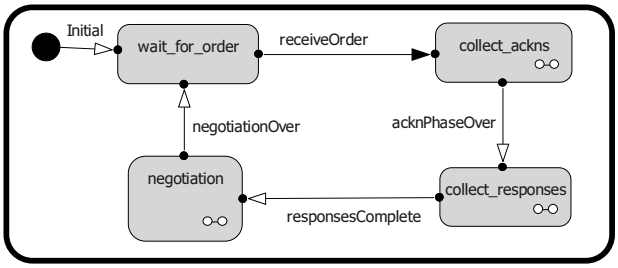

Figure 8. Top level statechart of the negotiator

to all other assembly robots. The transition *acknPhaseOver* must fire synchronously in all participating assembly robots. During state *collect_responses* all participants send their offer, rejection, or cooperation offer to each other. After every response is collected, the transition *responsesComplete* fires. This should also happen synchronously. During the *negotiation* state, the responses are compared and possible cooperations are established.

Statecharts are hierarchical. Figure 9 shows the sub-statechart of state negotiation. The rules necessary for decisions during negotiation are modeled as *choice points*. For example the choice point *only_rejections* evaluates to true, when the simple scenario of section 2 happens.

If we look at the second scenario of section 2, then the negotiation sub-statechart of assembly robot X would go from *only_rejections* over *False* to *did_I_reject*, over *False* to *criteria1*, over *takeFirstOffer* to *am_I_winner*, over *sendAcceptance* to *negotiationOver*.

To every transition an action can be attached. We modeled these actions as operations of the *Negotiator* capsule. During modeling the behavior, which we outlined in this section, the structure of our capsules

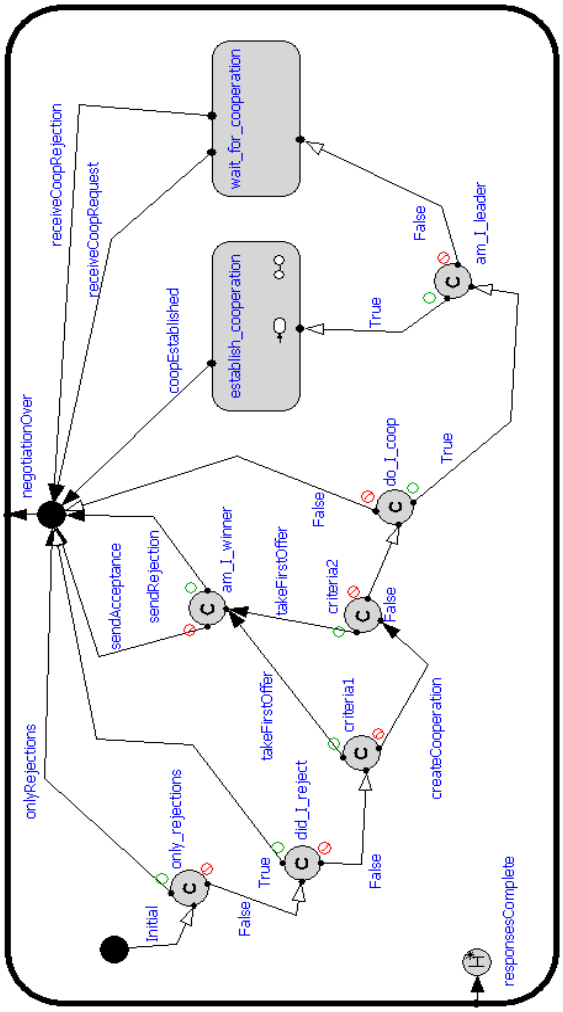

Figure 9. Sub-statechart of state negotiation

became also more detailed. For this aspect the next section is dedicated.

5. The Resulting Negotiator Capsule

Modeling the behavior of capsules also results in a more elaborated class structure. This means, that the capsule now has about 40 new operations, attributes, and helping classes like sorted collections or data classes associated with it. The *NegotiationProtocol* now contains 8 outgoing signals (Figure 10). Protocols can be conjugated, so that the signal direction is inverted. The port *negotiationInPort* is mapped to the conjugated *Negotiation-Protocol*. To every signal a data class can be attached. These data classes are normal classes and no capsules.

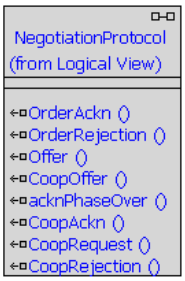

Figure 10

The negotiator also needs a port to the order management of the assembly robot. This is left out in the paper.

6. Implementation and Testing

UML offers *component* and *deployment diagrams* for documenting implementation specific things. For programming of operations, actions, and choice points we used C++. With this, the model can be compiled and run. During execution the statecharts of capsules are monitored. It is possible to record traces about signals sent and received by capsules. These traces can be transformed into *sequence diagrams*.

One possibility of validating the modeled behavior against requirements is to compare the monitored sequence diagrams with the "ideal" sequence diagrams, which are documented for the use case scenarios. An outline of this process is shown in Figure 11.

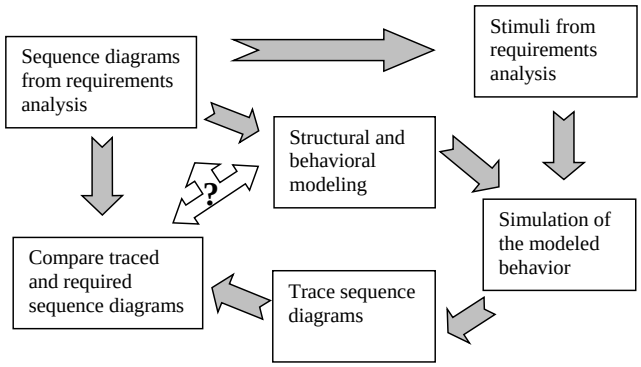

Figure 11. Validation activities

7. Summary

The development of negotiating assembly robots is an ongoing work in our department. Its complexity demanded the application of analysis and design techniques like UML. Especially extensions made by UML-RT improved the applicability of UML in the area of manufacturing automation. For the integration with existing PLC environments *Function Block Adapters* are proposed in [7].

8. References

[1] Holonic Manufacturing Systems http://hms.ifw.uni-hannover.de/

[2] G. Booch, J. Rumbaugh, I. Jacobson, *UML Users Guide*, Addison-Wesley, 1999.

[3] B. Selic, J. Rumbaugh, "Using UML for Complex Real-Time Systems", http://www.objectime.com/otl/technical/umlrt.html

[4] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns*, Addison Wesley, 1995.

[5] http://www.rational.com

[6] B. Selic, G. Gullekson, P.T. Ward, *Real-Time Object-Oriented Modeling*, Wiley, New York, 1994.

[7] T. Heverhagen, R. Tracht, "Integrating UML-RealTime and IEC 61131-3 with Function Block Adapters", IEEE Int. Symp. on Object Orient. Realtime Computing (ISORC2001), to appear.