Towards Service-Based Flexible Production Control Systems and their Modular Modeling and Simulation*

Holger Giese and Ulrich A. Nickel
University of Paderborn
Warburger Straße 100
D-33098 Paderborn
Germany
[hg|duke]@uni-paderborn.de

Abstract: Modeling of modern production plants often requires that the system provides means to cope with frequent changes in topology and equipment and can easily be adapted to new or changing requirements. For validation in form of simulation, however, usually a complete specification of both, the production control software and the physical elements of the manufacturing plant is required. We therefore proposed to use a service-based architectural approach to build the control software using more rigorous separation by means of well-defined interfaces following the software component paradigm. We present an extension of ROOM that further facilitates service-based design and permits the independent validation of components for such a design style. We show how the combination of both concepts permits the compositional validation of the system and thus enables early design validation even for flexible systems. The presented approach further reduces the validation overhead imposed by design evolution as long as local component properties are considered and component interfaces are stable.

1 Introduction

Today’s production plants are often characterized by the facility of manufacturing individual goods with small lot sizes. There are many product variants, which means that one has to employ flexible manufacturing systems which can easily be adopted to the new requirements. The available production equipment may also change temporarily due to downtimes and durable when the production capacity has to be adjusted somehow. Current production systems face therefore three major problems: (1) Production control software needs to become decentralized to increase their availability. It is not acceptable, that a failure of a single central production control computer or program causes hours of downtime for the whole production line. (2) Production control software becomes more and more complex. In contrast to the traditional mass production, today’s market forces demand smaller lot sizes and a more flexible mixture of different products manufactured in parallel on one production line. (3) The production control software architecture is required to support a flexible extension mechanism to adjust a system rapidly when additional equipment is available or equipment has to be removed.

* This work has been supported by the German National Science Foundation (DFG) grant GA 456/7 ISILEIT as part of the SPP 1064
These problems can be addressed by flexible production control software that employs autonomous operating software agents and a service-based overall software architecture. Some of these production agents will control specific parts of the overall production system like a single manufacturing cell or a transport robot. Other production agents will take the responsibility for manufacturing certain kinds of goods. Such flexible autonomous production agents need knowledge of manufacturing plans for different goods and of their surrounding world, e.g. the layout of the factory or the availability of manufacturing cells. In addition, such production agents have to coordinate their access to assembly lines with other competing agents.

Assume, that a new kind of good shall be produced. If one has to change the topology of a system, often big parts of the control software have to be specified anew. This causes long down times due to extensive tests. An approach would be to simulate the modified specification of the control software, beforehand. Such a direct simulation approach implies, that we have a complete specification of both, the production control software, and the physical elements of the manufacturing plant. In a large production system such changes may happen frequently, which in the worst case would cause an extensive adaptation of some parts of the specification. But, for the agent interaction the exact behavior of the agents is not relevant. It is only important, that the agents can communicate correct with each other via a particular protocol. Other details are not relevant and we therefore have to avoid such direct implementation dependencies and apply the traditional software engineering principles of separation and modularization. A first attempt towards the modular design and simulation of production control software has been developed in [GN01]. In this paper this work is extended towards support for more flexibility in form of service-based architectures.

The rest of the paper is organized as follows. In section 2 the used case study and techniques are described. The problems of missing modularization and separation are then discussed in section 3, where the concepts of component-based and service-based design are introduced. Some inherent restrictions of current contract-based specification approaches w.r.t. component behavior and simulation are also discussed. In section 4 we present an approach, that supports the required compositional component notion and partial model simulation. The resulting evaluation scenario in form of simulations is described in section 5 and the paper is finally concluded.

2 Modelling Flexible Production Control Systems

This paper uses the simulation of a simple production process as running example. This production process models a factory with various manufacturing places and with shuttles transporting goods from one manufacturing place to another. The example stems from the ISILEIT project funded by the German National Science Foundation (DFG). The goal of the project is the development of a formal and analyzable specification language for manufacturing processes. This specification language shall allow us to verify important system properties like lifeness and the absence of deadlocks. In addition, a code generator shall provide automatic code generation for the building blocks of a manufacturing process, namely shuttles, gates, storages, assembly lines, etc. The used flexible manufacturing system case study is realized with a track based material flow system, which transports the
goods to the different robots or working stations (see Figure 1). Note, that in the current employed case study the physical shuttles are not equipped with a computational device. Therefore, additional external control software on additional computation nodes that handles the shuttles is required.

Figure 2 shows a schematic overview of our case study. On this production line we produce bottle openers, which consist of several components. The system is specified in a way, that one can assign a production task to a shuttle, which means that one shuttle is responsible for the production of certain components. The first step in the working task is to move to station 1, where somebody equips the shuttle with the appropriate material. A display shows the worker, which pieces are needed. By pushing a button, the worker signals the material flow system, that the shuttle is completely equipped. The shuttle now moves to station 2, where the portal robot takes the material from the shuttle and hands it over to the rotator, where the required manufacturing step is performed. After that, the portal robot takes the assembled good from the rotator and puts it on the waiting shuttle. The shuttle now moves to the storage (station 4) where the good is stored. If the control station does not assign a new task to the shuttle, it will rotate on the main loop, until it gets a new task again. Note, that station 3 does not have any functionality at the moment. In the near future, it will connect this production line to a second one.

In our case study, we use Programmable Logic Controllers (PLC) to control the stations of the production line. For the specification of the behavior of such a controller, different approaches exist, which cover different abstraction levels. Sequential Function Charts for example describe the sequence of a PLC program as a state transition diagram, whereas Structural Text (ST) is a notation similar to PASCAL. The PLC programming standard
IEC61131-3 [Int93] tries to integrate these languages based on a common concept of data types, variables and program organization units.

The behavior of embedded system processes is also often specified either using SDL process diagrams [ITU99] or using statecharts [HG96]. Both notations basically model finite state automata which react on signals by executing some actions, sending signals, and changing to new states. Both languages have a well defined formal semantics and tool support for analysis and simulation and code generation is available [Dou98, AT98, RoR].

However, the discussed languages do not provide appropriate means for the specification of complex application specific object-structures as required by the described autonomous production agents.

Common object-oriented modeling languages, like UML, support the modeling of complex application specific object-structures. UML focuses on early phases of the software life-cycle like object-oriented analysis and object-oriented design, cf. [BRJ99]. Thus, UML behavior diagrams, like collaboration and sequence diagrams, usually model typical scenarios describing the desired functionality. Only UML state diagrams - an adjusted version of the original statecharts [HG96] - provides means to model reactive behavior. For the real-time and embedded system domain, ROOM [SGW94] and its successor UML-RT [SR98] provide the required more domain specific adjustments. See [RSRS99] for a proposal how to map the concepts of the ROOM approach to the UML.

Figure 2  Topology of our sample factory example

IEC61131-3 [Int93] tries to integrate these languages based on a common concept of data types, variables and program organization units.

The behavior of embedded system processes is also often specified either using SDL process diagrams [ITU99] or using statecharts [HG96]. Both notations basically model finite state automata which react on signals by executing some actions, sending signals, and changing to new states. Both languages have a well defined formal semantics and tool support for analysis and simulation and code generation is available [Dou98, AT98, RoR].

However, the discussed languages do not provide appropriate means for the specification of complex application specific object-structures as required by the described autonomous production agents.

Common object-oriented modeling languages, like UML, support the modeling of complex application specific object-structures. UML focuses on early phases of the software life-cycle like object-oriented analysis and object-oriented design, cf. [BRJ99]. Thus, UML behavior diagrams, like collaboration and sequence diagrams, usually model typical scenarios describing the desired functionality. Only UML state diagrams - an adjusted version of the original statecharts [HG96] - provides means to model reactive behavior. For the real-time and embedded system domain, ROOM [SGW94] and its successor UML-RT [SR98] provide the required more domain specific adjustments. See [RSRS99] for a proposal how to map the concepts of the ROOM approach to the UML.
3 Towards Flexible and Modular Design and Evaluation

Assume, that a new kind of good shall be produced. If one has to change the topology of the material flow system, e.g., often big parts of the control software have to be specified anew. This causes long down times due to extensive tests. Thus, in [NN01] we describe an approach of how we can simulate the modified specification of the control software, beforehand. The Fujaba CASE tool [Fujaba] can be used to generate executable Java code and to observe its execution using the Java reflection library. Production sequences can be visualized and analyzed. The simulation is based on a simulation kernel, which serves as a model for the physical components of the production system.

Such a simulation approach implies, that we have a complete specification of both, the production control software, and the physical elements of the manufacturing plant. Imagine, that we just change the set-up for the assembly line or the CNC-code of a robot. In a large production system this happens frequently. In the worst case, this would cause an extensive adaption of some parts of the specification. But, for material flow purposes, many details of the behavior of an assembly line are not relevant. It is only important, that a shuttle can communicate with the assembly line via a particular protocol or how long the assembly line takes to perform the next production step. Other details as specified in the statechart of the AssemblyLine class in Figure 3 are not relevant. We therefore have to avoid such implementation dependencies and apply the traditional software engineering principles of separation and modularization.

![Figure 3 Statechart of AssemblyLine](image)

The description of external phenomena like the AssemblyLine in form of a class is a problematic solution. Traditional class-based object-oriented design does not emphasize separation and thus more rigorous separation by means of interfaces is required when direct class dependencies should be avoided. This implicit treatment does not support the inde-
pendent deployment and composition of parts. The component paradigm [Szy98] therefore demands to consider the contractual relations more explicit to support the systematic exchange of parts. Instead of direct class relations, explicit contracts have to be used.

The concept of evaluation by simulation does also not scale up to complex systems, because the cognitive capacity of a human to keep track with the simulation results visualized by the tool is rather limited in practice. Thus, an overall system simulation is not directly applicable for evaluation of complex systems. A compositional component-based and systematic approach for modeling and checking properties is therefore needed, which permits to check system properties in a modular fashion within the original scenario of simulation-based early design evolution.

3.1 Component-Based Design

Building systems by using components is a well known concept from classical engineering disciplines. The reuse of such pre-defined and tested software components allows the engineer to construct systems on a very high abstraction level. This decreases time-to-market and improves the productivity. Thus, effort has been put on adopting these principles to software engineering. A successful example for the application of the component paradigm in the field of industrial process control can be found in [CL00].

In terms of software, a common notion for components does rarely exist [Cou99]. However, most definitions put emphasis on the independence between production and deployment, the composition by third-parties and the explicit specification of context dependencies in a contractual style. Szyperski [Szy98] defines a component as follows:

A Software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

Following this definition, component technology adopts the principles of object-orientation [RBP+91,Boo93,JCJO96,CAB+94] from the implementation in a programming language environment up to the run-time environment of the system. An application or system is decomposed into runtime elements, that can be build, analyzed, tested and maintained, independently. As Meyer and Mingins stated in [MM99] component technology is based on the solid ground of object technology in applying the encapsulation principle of object-orientation.

For the real-time and embedded system domain, ROOM [SGW94] and its successor UML-RT [SR98] propose a component concept (capsules) with explicit connections points (ports). A very general notion of a protocol is employed to specify general multi-party interactions. This includes binary protocols which are by far the most common ones. The ROOM port concept permits to restrict the possible interaction via a specific connection to a certain set of signals specified by a given protocol role in form of a statechart.

Besides the ports of the ROOM approach, in the literature a number of different notions for contracts have been proposed which can be classified by the following hierarchy [BJW99]: syntactical interface, behavior, synchronization and quality of service. While contracts in form of syntactical interfaces are supported by all typed programming languages and middleware platforms, full behavioral contracts, e.g. in form of pre- and post-conditions, are rarely used in practice. Synchronization contracts can describe the non uni-
form service availability, while also request scheduling solutions such as specific reader/writer policies are possible. Quality of service contracts further allow to describe the contract behavior w.r.t. time and throughput characteristics, but their platform dependent nature renders their consideration during design and simulation a complex task.

For the production control domain a more detailed notion for contracts than commonly used in programming languages is required. Therefore, a contract has to be a kind of boundary object that extends the UML-RT port concept rather than being simply a pure abstract concept such as an interface. We further restrict the rather general notion of protocols and protocol roles in ROOM to the binary case to exclude the complexity of multi-party interactions and their scheduling [JS96].

In Figure 4 a Factory subsystem developed in this manner is presented. The description employs the UML-RT concepts to denote the provided and used boundary contracts of the factory subsystem in form of ports. As a visual short-cut we use a black box for a main role and a white one for its counterpart. The explicit handling of the main protocol role and its complementary role is also simplified using a single <<contract>> stereotype to specify exactly one main protocol for the providing side and derive the usage protocol of the client side, implicitly.

In Figure 5 for the AssemblyLine class such an explicit contract CAssembly for the assembly line is presented. The whole subsystem of the factory example of Figure 1 can be redesigned in this manner employing the component and contract concepts. In the given factory example the task processing provided via the CTask contract is realized using the CSource, CAssembly, CTarget and CPlanRepository contract.

Following the component paradigm, instead of implicit abstract classes such as AssemblyLine, a contract CAssembly with a protocol is used to describe the component boundary (cf. Figure 5). Thus, some of the problems that arise when the detailed behavior of a component changes, can be avoided. The provided and used contracts of a component are an implicit description of all possible environments and thus result in a well-defined and complete test frame. In contrast, in the case of an explicit given class, only one specific test scenario is given which consists of the given surrounding classes and their current implementation. The abstraction and decoupling realized via the contracts can be further used to support the modular simulation.
3.2 Service-Based Architectures

As one of the major problems of current production systems, we already identified the need for a more flexible architecture concerning the adjustments due to changes in the system setup - even at run-time. On the software level the dynamic composition of components via services has recently received considerable attention in form of open service-oriented software architectures and web-services (cf. [NET00, ONE01]). Here, each application determines its context embedding by means of dynamic service lookup. This architectural style facilitates the integration of independently developed systems, using service contracts which include meta-data to guide their composition with third-party components in a plug & play manner at run-time.

However, service-oriented software architectures are not really a new concept. The basic principles have been standardized, e.g. in tina-c [CM95] in the telecommunication field or the ISO Open Distributed Processing (ODP) model [ISO95]. Well established middleware approaches such as DCOM [Cha96] or CORBA [Vin98] also support trader and name services for dynamic lookup of services. Newer approaches such as Jini [Sun00], Microsoft .NET [NET00] or Sun ONE [ONE01] further emphasize the service-oriented composition at run-time.

In the domain of production control systems the most prominent approach is OLE for Process Control (OPC) [IL02]. It extends the PC based Microsoft concepts towards the industry and automation domain providing a server browser for remote lookup. Sun’s Jini [Sun00] also address embedded devices. However, the main focus lies on small independent devices and the support of ad hoc networking.

To model the service-oriented aspects of a production control software, we have further extended the UML-RT approach. We use solid contract/port connections to describe fixed cooperations while service-based dynamic ones are visualized using a dashed line.

In Figure 6, the internal design of the factory software component is described using both, fixed and dynamic cooperations. Inside the factory control software, a number of shuttle agents are used to describe the autonomous processing required for each requested task. The underlying hardware restrictions (the shuttles itself have no programmable processing
units) further enforce a mapping where the shuttle agents virtually control their physical shuttle by communicating with the connected gates to achieve their goals. The required processing further enforces, that the shuttle agent cooperates temporarily with the material depot, assembly lines or output depot using the $C_{Source}$, $C_{Target}$ and $C_{Assembly}$ contracts.

The contract $C_{PlanRepository}$ is used only in occasional periods to optimize the performance of the shuttle agents by providing actual informations concerning changes in the floorplan of the factory. However, the autonomy of the agents ensures, that they will adjust to directly observed changes.

Figure 7 depicts the contract used to decouple the overall $FactoryControl$ and the different shuttles. Its basic protocol describes how a task $t$ to process is hand over and autonomously processed by the shuttle control software agent.

In contrast to the common static cooperation scenario in embedded systems the service-based architecture is employed here. Therefore, when the overall factory software control component $FactoryControl$ receives a new request to process a task it dynamically looks
up an available shuttle agent to process the task. Thus, when a shuttle is temporarily not available and therefore no longer registered in the service lookup, the overall control will automatically choose another shuttle, if available.

This is achieved using the contract types to identify reasonable matches between service providers (Shuttles) and service customers (FactoryControl). On the first sight, type equivalence seems to be a reasonable choice for matching. However, the further evolution of systems will then enforce major redesign activities when, for example, extended versions of a shuttle or shuttle software are employed. A more practical choice is inheritance-based subtyping as supported by most programming languages and middleware approaches. This choice assumes, that all further developed software artifacts are using the same unique class and contract hierarchies. If these assumptions are not fulfilled, an alternative approach are efficient notions for contract matching (subtyping) that exploit the type structure or meta-data (cf. [NET00, Sun00, Gie01a]). In this case, we propose to employ notion proposed in [Gie01a], while depending on the supported contract types other notions are also applicable.

4 Behavior and Composition

Due to the fact, that a component is subject to composition by third parties, it has to provide a clear definition of how it can be used. In the object-oriented approach, interfaces or abstract classes are employed to separate usage and implementation concerning the syntactical typing. For component-oriented programming and distributed systems the independent construction of each component further requires that a suitable service contract covering also semantic issues is provided. Otherwise, the required implementation-independent information for the composition by third-parties is not available.

Component systems differ from traditional software products, where a view restricted to the white-box composition and the sequential case fits most often. Instead, the third-party composition of components has to consider black-box composition and often takes place in a concurrent environment. Therefore, composition can result in reasonable synchronization problems, such as deadlocks, that cannot be avoided using specific implementation styles. To address this problem, such composed systems have to be tested or simulated. However, to get a reasonable result, usually a complete specification of the system is required.

Using the protocols of provided and used contracts, even the simulation of an incomplete specification becomes feasible. In a first step, the contract protocols can be used to build the most general possible component environment by generating arbitrary request sequences as guaranteed via the provided contracts and assuming the behavior guaranteed by the used contracts. A simulation can, however, cover only the possible system behavior, but fails for liveness aspects. When classes represent the component border in an implicit manner, the model simulation can assume progress for each single statechart, because it describes a realization which is executed. In contrast, to assume progress for a given set of provided and used contracts will also result in the obligation of connected components to serve them in an independent manner. In practice, however, provided and used behavior do often depend on each other and therefore progress and liveness properties can not be handled for each contract in isolation.
Another crucial question w.r.t. progress and the provided component contracts is, whether different clients are served in a fair manner. If no fairness is assumed, a set of client requests will only exclude the blocking of one of its members, when the direct or indirect synchronization of the clients itself exclude that one client can rule out any other one. Therefore, to implement fairness in an explicit manner based on a set of unfair operating components is a rather hopeless undertaking. The single components should instead guarantee to process the different client request in a fair manner, e.g., using a fair request queue.

The described contract concept as short-cut of the port and capsule extension thus permits to evaluate whether the component connections are used and provided in a protocol conform manner. However, the proposed protocols and contracts are not sufficient to achieve the intended modular form of designs which supports validation by simulation. The protocol restrictions do not address lifeness properties and therefore fail to describe the component environment as required.

To overcome this limitations for contract protocols, we extend the used notion of state-charts and further distinguish progress and quiescent transitions [Rei98] denoted by usual and dashed arcs, respectively. We further demand that progress is guaranteed by the contract provider and all possible provider events are never blocked by the clients. Therefore, a secure usage will only result in a situation where clients will wait for the answers or guaranteed state changes of the provider, whereas the provider can never be blocked by a client.

In Figure 8 quiescent transitions from the states waiting and producing leading back and forth to a state failure are used to also take the possible malfunction into account. These transitions might occur, but may also never occur. In contrast for the other progress transitions hold that one transition will occur if at least one is enabled. Note, that no history state is used and therefore the semantics of the statechart requires that in case of a failure the loaded material is always removed manually before the shuttles become operational again.

The progress guarantee for provided contracts results in arbitrary protocol conform usage by the test environment. For the used contracts the guaranteed progress is employed during
simulation and therefore only relevant cases of permanent blocking are observed. The boundary contracts \texttt{CTask}, \texttt{CSource}, \texttt{CTarget}, \texttt{CAssembly} and \texttt{CPlanRepository} have to be combined with the executable model of the factory example to obtain the intended test scenario. A task request may occur and the appropriate processing by the shuttles is initiated. The used contracts \texttt{CSource}, \texttt{CTarget}, \texttt{CAssembly} and \texttt{CPlanRepository} are requested as specified exploiting the progress property. The progress of a contract, however, cannot always be guaranteed by realizing the component independently of the component environment and the interplay with other components. The described test environment construction is therefore only valid in a strictly layered architecture.

If more general forms of architectures are considered the relation between provided and used contracts of components cannot be ignored. In contrast to safety properties it is problematic to ensure lifeness properties such as progress for arbitrary connected components. Therefore, commonly the whole external relevant component behavior described in form of processes such as formalized by CSP [Hoa85], CCS [Mil89] or LOTOS [ISO89] have to be considered.

An overwhelming variety of preorders and congruences for process refinement and abstraction have been proposed for these process algebraic approaches to address whether a given external component specification is realized correctly. The proposed relations, however, have to ensure substitutability [LW93] w.r.t. any possible process environment and therefore often result in very tight specification realization relations. The relations enforce that the specification has to reveal too many details of a realization, e.g., the complete buffering effects, and therefore do not provide the necessary degree of separation.

In a layered architecture with acyclic module usage relations, in contrast, a separate treatment of lifeness properties becomes feasible [LS94] by exploiting the acyclic nature of dependencies. We generalize this idea to support separation for progress properties even for non-layered structures employing explicit contract dependencies for a given set of provided and used contracts.

In [Gie00,Gie01b] even the combination with explicit partial external specifications covering a subset of the used or provided contracts of a component are presented. To specify such complex dependencies between multiple provided and used contracts so-called complex contracts describing the explicit behavior and synchronization can be employed. By including the traditional case of a complete external specification in form of a specification process the approach supports a whole spectrum of possible component descriptions varying w.r.t. the degree of abstraction and embedding restrictions.

For the factory example we consider only the simplest case to build the needed overall component behavior based on explicit specified contract dependencies. If a used contract is required to realize the behavior of a provided contract, we have to declare a dependency between them to make this regress explicit. We further have to restrict that for the composition of components the concatenated dependency relations remains acyclic. The specified dependencies therefore guide the possible component composition by demanding that a component embedding never results in a cyclic progress dependency [Gie01b].

The construction of the dependency relation for the elements of the \texttt{Factory} component can be easily determined considering which used contracts are required to ensure progress of the provided contract. Following this simple scheme in Figure 9 we can conclude that
the CT\text{ask} contract of the Factory\text{Control} requires the Shuttle contracts and thus depend on them. For the Shuttle holds that its provided behavior requires the CSource, C\text{Assembly}, CTarget and CGate contracts, while the C\text{PlanRepository} contract is required only sometimes to update the agent information concerning the floorplan of the factory. If it's temporarily not available this will not hinder the agent to fulfill its task. Therefore, no dependency has to be declared.

In Figure 10 the dependencies between the provided CT\text{ask} contract of the Factory component and the used contracts are specified. The construction of the dependency relation for the Factory component can be easily derived considering its internal design as depicted in Figure 9 using the following simple rule: The dependencies of a component are given by the concatenation of all internal port connections and the dependencies of the contained components. Thus the cooperation of the CSource, CTarget and C\text{Assembly} contracts are definitely required for providing the Task contract, while the C\text{PlanRepository} contract is only used in a sporadic manner to update the locally stored machine control programs.
5 Modular Evaluation

Current approaches to evaluate software systems and their architecture which take concurrency problems into account [MDEK95, LAK+95, GKC99] do not address an open dynamically composed component system that permits the plug & play for independent developed components.

The proposed approach, however, can evaluate the software components in a suitable test environment by using the additional information provided by the contract protocols and dependency relation. While the provided and used contract protocols are combined as described before, the progress of provided and used contracts of the test environment is non-deterministically controlled as specified by the contract dependency relation. When the test environment in a specific state is waiting for progress at the CTask contract, it has to provide progress for the CSource, CAssembly and CTarget contracts. The CPlanRepository contract, in contrast, needs not to be served in this state.

Besides the component internal dependencies between provided and used contracts, the overall component behavior of a factory is of interest. We propose to use a set of UML sequence diagrams to specify the necessary component behavior. Each time the prefix of one such specified trace is present, we further demand that the overall partial trace has to be conform with the one specified by the sequence diagrams. Otherwise, the realization of the component does not conform to that sequence diagram.

In Figure 11 the expected property for a single do request to a factory is specified. Using a sequence diagram and the scenario based techniques permit to consider this requirement in separation. The factory realization will process multiple do request in parallel, but for each single request the presented sequence diagram can be used as necessary behavioral property. A simulation scenario should therefore take track of the initiated do requests and whether the specified related deliver requests have occurred.

In Figure 12 the expected property for a single do request to a factory is specified. Using a sequence diagram and the scenario based techniques permit to consider this requirement in separation. The factory realization will process multiple do request in parallel, but for each single request the presented sequence diagram can be used as necessary behavioral property. A simulation scenario should therefore take track of the initiated do requests and whether the specified related deliver requests have occurred.
In Figure 12 the decomposition of the required property for a single do request to the elements of the factory is depicted. Splitting the sequence diagram a test scenario for the two relevant components FactoryControl and Shuttle is derived.

6 Conclusion and Future Work

In this paper a concept for the design of flexible production control system software that enable partial model evaluation in form of simulation has been presented. It has been discussed which extensions to traditional contract-based component separation are required to end up in a useful and appropriate service- and component-based architecture which supports partial simulation. The proposed technique to describe the component behavior and component environment does further provide the necessary restrictions to exclude unexpected component embeddings which invalidate the assumed dependencies for the provided and used contracts of a component. We plan to further extend the presented ideas to also address besides simulation model checking [CGP00]. Also contracts and requirements should include quality of service aspects such as worst case execution times or throughput guarantees. Also more elaborated tool support for the design and evaluation of complex real-time systems with this approach is planned.

References


