T U M

I N S T I T U T F Ü R I N F O R M A T I K

Modelling System Families with Message Sequence Charts: A Case Study

Stefan Wagner, María Victoria Cengarle, Peter Graubmann

TUM-I0416
Oktober 04

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-10-I0416-0/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Modelling System Families with Message Sequence Charts: A Case Study

Stefan Wagner[1], María Victoria Cengarle[1], and Peter Graubmann[2]

[1] Technische Universität München, Institut für Informatik, Boltzmannstr. 3, D-85748 Garching
[wagnerst|cengarle]@in.tum.de

[2] Siemens AG, Corporate Technology, Otto-Hahn-Ring 6, D-81739 München
Peter.Graubmann@siemens.com

Abstract. A production system is used as case study of MSCs enriched with a connector construct intended to improve the design. The language of MSCs is further enlarged with the notion of variation points and variants in order to capture the evolutionary aspects of system family development. The design of the basic production system is based on earlier work and provides the initial core assets of the presented system family. Variations of the system are developed and incorporated into a family model that can be parameterised by the features for a specific product.

Keywords: Message Sequence Charts, MSC Connectors, Product Lines, System Families, System Family Evolution, Variation Points, UML Sequence Diagrams

Acronyms: MSC: Message Sequence Chart, HMSC: High-Level MSC, HTF: Holonic Transport Vehicle

This work was sponsored in part by the BMBF within the Eureka Σ! 2023 Programme, ITEA projects ip0004 *CAFÉ* and ip02009 *FAMILIES* and by the DFG project *InTime*.

Contents

1 Introduction

Software system families have established themselves as an important area in computer science research and industrial practice [15]. In this paper, we develop and try out concepts to model a system family using Message Sequence Charts (MSCs) in a case study. Special focus is on the applicability of MSC connectors [3,6,7] for the description of component interaction. We also evaluate the usage of MSC connectors as an abstraction means within standard MSCs. We further extend the standard MSC language with additional concepts that proved useful in the context of system family description and evolution. The approach is tailored for the MSC language but the concepts are equally applicable to the UML [23] as well.

A system family, also called product line or product family, is a *set of software systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way*. [15]

The specification of a holonic flow of material in a production system [2] is used as basis for the case study. In this production system, autonomous vehicles transport engine parts between machine tools where they are processed. In this context, the term "holonic" refers to the computer-aided and highly integrated control of this production system. It was selected because firstly it shows a high degree of communication, and secondly, the basic specification and design could be taken over from [1]. We intentionally resorted to an already existing specification in order to avoid ending up with a specification that was particularly designed to meet our expectations with respect to the MSC connector applications. This design is extended by several variants of both machine tool and transport vehicle mechanisms in order to turn the basic production system into a larger system family. All these variations are incorporated into a single parameterised model that allows a variant selection based on system features.

Contribution. The contribution of this paper is twofold. Firstly, we show the usefulness of MSC connectors in terms of conciseness. Secondly, variations and variation points are introduced into the MSC language and a first methodical approach for the transformation of feature sets into variation points and variants is presented and applied in a case study.

Outline. The report is divided into two main parts. Part I describes the concepts that are evaluated in the case study that is presented in Part II. We start sketching out our approach to modelling variants and variation points in Message Sequence Charts in Section 2; in Section 3 we summarise the MSC connector concepts and indicate the usage of variants in this construct. In Section 4, we briefly describe the basic production system as it is specified in [1,2]. Based on this specification, we design this basic production system with MSCs and MSC connectors (Section 5). The high-level view of the entire system is presented in Section 5.1; a refinement of the transport vehicles can be found in Section 5.2; the database of the transport vehicles is briefly refined in Section 5.3. In Section 6, we extend the basic production system and introduce several alternative mechanisms for the production process. The respective design changes let the basic production system evolve into a real system family as presented in Section 7. Section 7.1 presents the features of the system family in a structured way, Section 7.2 describes the connection between features and MSCs, and Section 7.3 shows the revised MSCs of the high-level view. Eventually, we present related work in Section 8; conclusion and outlook follow in Section 9.

Part I
Concepts

2 Modelling Variability

In this section we describe our approach to modelling system families with MSCs in general. The basis for the modelling is a feature model following [10] that is used to identify the variability in the system family. We define feature, variation point, and variant, and present a methodical approach for the transition from the feature model to MSCs with variation points and variants.

2.1 From Features to Variations

A *feature* is an essential aspect or characteristic of a system in a domain. From the developers' point of view features can be described as distinctively identifiable abilities of the system that must be implemented, tested, delivered, and maintained [22]. There exist methodical approaches such as FODA [10] and FORM [22] that can be used to organise several features into a tree of And/Or nodes in order to identify the commonalities and variabilities within the system. We assume that one of these approaches is used to build a feature model as starting point from which we derive variation points in MSCs with the method proposed below.

A feature can have several children in the tree that express different variations. Therefore, features cannot directly be used to describe the variability in the model. We follow [15,20] in the definition of a variation point: A *variation point* is a location within a use case of the system where a variation occurs. That variation indicated by the variation point is captured in one or more *variants* that describe the different possibilities of the variation. We understand use cases here in a very general sense so that they describe different possibilities of the usage of the system. Moreover we consider the use case to contain all layers of the system, e.g. low-level redundancy.

The main question concerning system family modelling is how to map features onto variation points and variants. The general relationship is that certain features are mapped to variants of variation points. Each feature, and consequently, the variant may be optional or alternative. A "mandatory feature" does not describe a variant[1], i.e., there is no variation and no need to map that feature to a variant. An optional feature maps to a single variant of a variation point. The approach for alternative features is similar. Each alternative becomes a variant of a common variation point. The name of the parent feature can also name the variation point if it has only one optional or one set of alternative features as children.

Hence we can describe a variation point by its name that might be directly derived from a feature. This results in the following structure for a variation point:

VariationPoint ::= *VariationPointName* (*VariantList*)

VariantList ::= *VariantName*, *VariantList*

| *VariantName*

We want to illustrate our approach with an example of a car wiper system: in our system, each car has a front wipe; some models may have rear wipes. Both the front and the rear wipe must be able to wipe steadily. The front wipe can optionally align its speed to the rain and car speed. Furthermore it can be

[1] In literature, the term "mandatory feature" is used nevertheless, mostly in the context of feature selection where "mandatory features" are useful in order to clearly determine the available choices.

a special type of wipe, either wiping in fixed intervals or starting automatically in case of rain. The rear wipe has an optional washing device. The corresponding feature model is depicted in Figure 1.

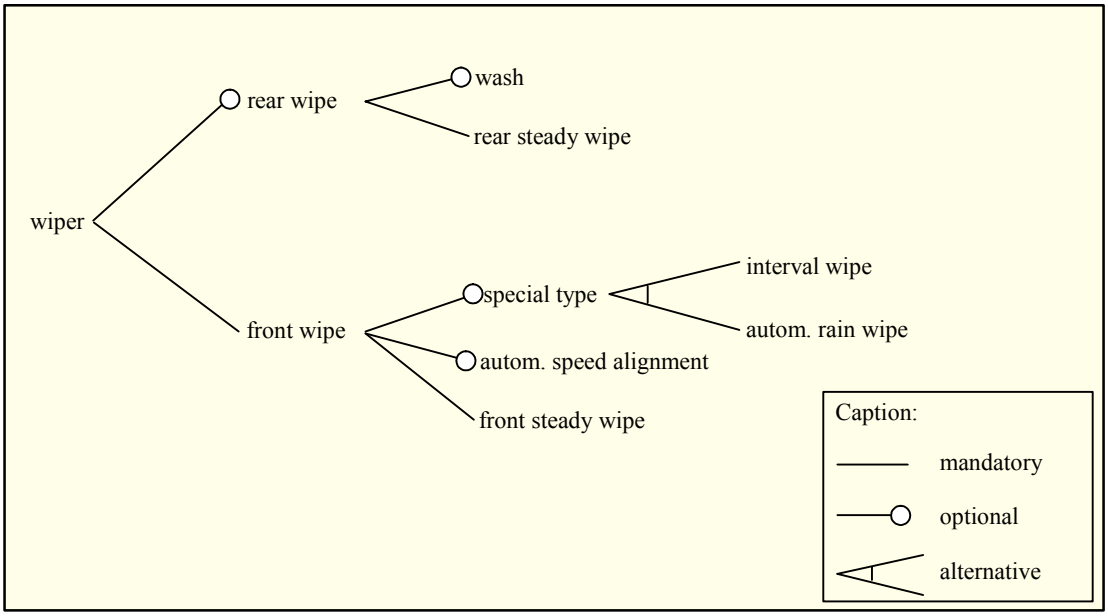


Figure 1 Example of a feature model for a car wiper

The feature model leads us to the following variation points and variants in the structure given above:

wiper (rear wipe)

rear wipe (wash)

front special (special type)

front autom. (autom. speed alignment)

special type (interval wipe, autom. rain wipe)

It can be seen that most variation points can use the name of their corresponding features. Only *front special* and *front autom.* had to be given new names. In this example there is also only the variation point *special type* that has two associated variants because there are only two alternative features.

2.2 Variability in Message Sequence Charts

For the MSCs we use the notion of *variant occurrences* and *variation point occurrences* derived from the variants and variation points. If a feature is chosen, it implies potentially one or more variants that have occurrences in several MSCs and connectors or the corresponding variation points appear in one or more HMSCs.

A variant occurrence can be described in the following attribute grammar based on [26]:

Interaction ::= *Basic*
| *CombinedFragment*

CombinedFragment ::= ...
| ***variant***(*VariantName ArgumentList*, *Interaction*)
{*allowedValues*(*VariantName.name*, *ArgumentList.list*}

ArgumentList ::= (*NonEmptyArgumentList*)
{*ArgumentList.list* := *NonEmptyArgumentList.list*}
| ε
{*ArgumentList.list* := nil}

NonEmptyArgumentList ::= *Argument*, *NonEmptyArgumentList*
{*NonEmptyArgumentList.list* :=
[(*Argument.name*, *Argument.value*) |
$NonEmptyArgumentList_1.list$]}
| *Argument*
{*NonEmptyArgumentList.list* :=
[(*Argument.name*, *Argument.value*)]}

Argument ::= *FormalParamenter* : *ActualParameter*
{*Argument.name* := *FormalParameter.name*;
Argument.value := *ActualParameter.value* }
| *FormalParameter*
{*Argument.name* := *FormalParameter.name*;
Argument.value := null}

MSCs are two-dimensional diagrams. Therefore the variant occurrences can appear in both dimensions. One way to change MSCs is to introduce or to remove instances – this obviously is a change in the horizontal dimension. Or we can modify the behaviour expressed in the MSC by adding or removing messages and actions, by changing conditions or by re-ordering the events on the instance axes (thereby of course keeping to the rules of the MSC language). All these changes can be seen as a modification of the partial order described by the MSC and modify the vertical dimension. These two dimensions for modelling variants are depicted in Figure 2.

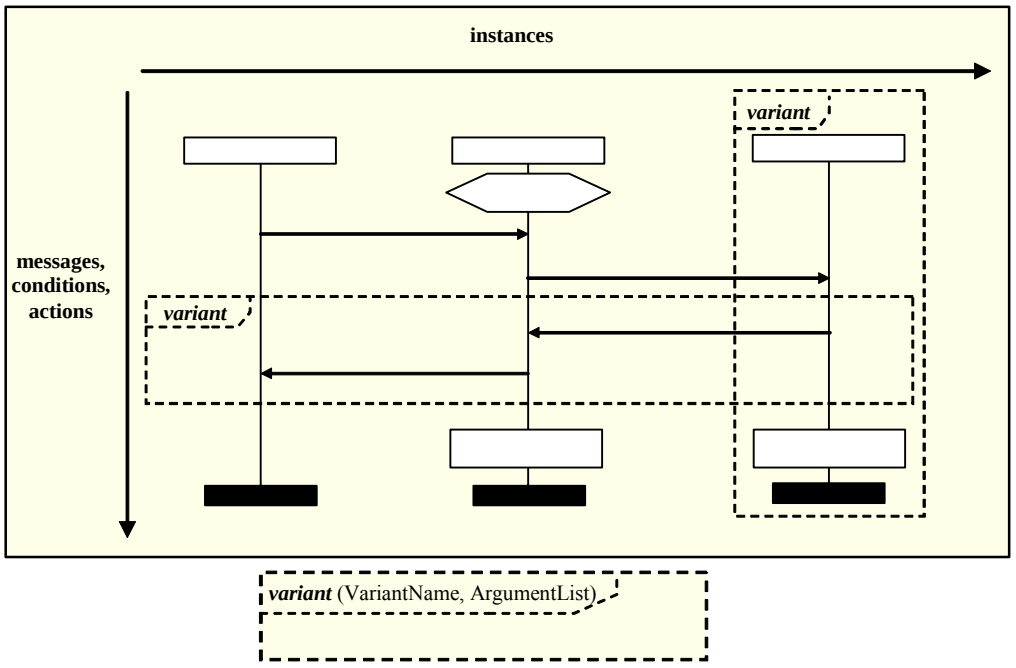


Figure 2 The two dimensions of variation in MSCs with the respective representations of the variant occurrence.

To express a change in the horizontal dimension, this is, for a modification of the occurrence of instances in a MSC, we do not have a language construct in the standard [8] yet. We therefore introduce a *repeat construct* for the MSC language that is capable of reduplicating instance axes. Syntactically, this construct is designed to conform to the well known loop construct with the sole difference that it is applied to instance axes instead of messages. It uses the keyword ***repeat*** followed by the repetition boundary indication (*n*,*m*) where *n* and *m* stand for (expressions of) natural numbers. This means that the operand – the instance axes together with the messages between them, contained in the dashed box that limits the repeat operator – will be duplicated at least *n* times and at most *m* times. Messages crossing the boundaries of the repeat operator will be duplicated accordingly; each instance of such a repeated message is understood to be sent to or received from a small co-region that is placed where the original message is attached to the instance outside the repeat operator, hence, the send or reception order of these duplicated messages is arbitrary (see Figure 3).

Similar to the loop construct, the repetition boundary may take the form (*n*,***inf***). This means that the repeat operand will be duplicated at least n times. Also, ***repeat*** <*n*> will be interpreted as ***repeat*** (*n*,*n*) and ***repeat*** (***inf***) may replace ***repeat*** (1,***inf***). ***repeat*** (0,*n*) or ***repeat*** (0) are acceptable; in the first case, the MSC part contained in the repeat operator may possibly be dropped out in a certain instantiation (this includes the elimination of the messages that cross the boundaries of the repeat construct, too); the second case just specifies that a certain MSC part does not occur: this is particularly relevant when optional variants are described within one MSC. A repeat construct where the first repetition bound is greater than the second one (i.e., $n > m$) is considered equivalent to a ***repeat*** (0) statement.

The repeat construct, originally thought for application in MSC connectors, proved also rather useful to structure and simplify the classical MSCs. In this case study, we only used the repetition of one instance axis a fixed number of times.[2]

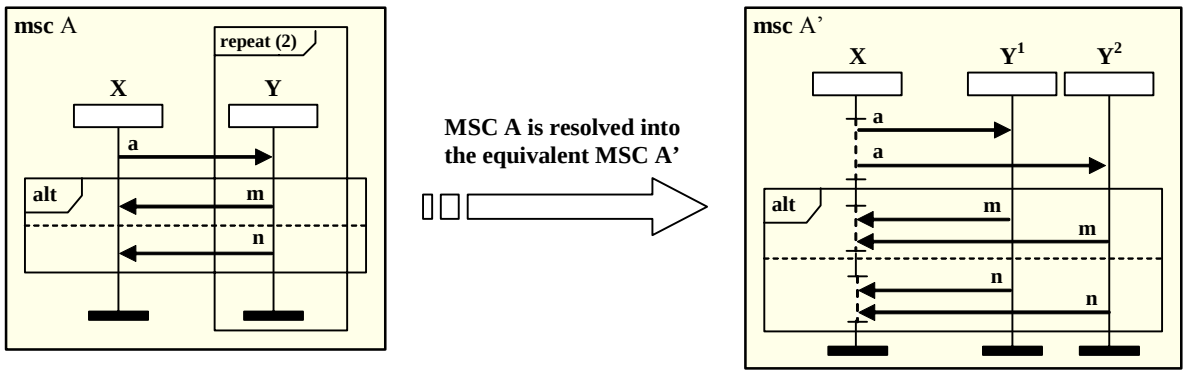


Figure 3 The repeat construct in MSC *A* and its resolution into MSC *A'*.

In case variations are too complex to be incorporated into a single MSC and actually describe strongly different sequences, it is also possible to describe variation points in HMSCs as shown in Figure 4. It is a combination of both dimensions on a higher level of abstraction and in essence similar to the description of variations in use case diagrams in [13,14].

[2] One might wonder whether a similar alternative construct with respect to instance axes might be useful; however, the classical alternative inline expression already does the job.

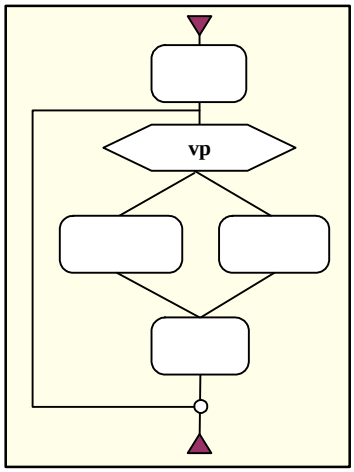


Figure 4 Variation points in HMSCs

Finally it is also possible to hide variants inside MSC connectors [3,6,7]. They use the same notational elements as the standard MSC language. The difference is however that the variations are not visible on the MSC level but are encapsulated in the connector. We introduce the MSC connectors in more detail in the next section.

We return now to the wiping system example from above. We cannot go into details here but want to illustrate the expression of variants in MSCs by an example MSC. In Figure 5 a possible sequence is shown for incorporating the special types of front wipers. The *FrontWiper* is either started by the *RainSensor* or by the *UserInterface*. In the *interval wipe* variant the length of the interval of the wiping might be changed. Finally the stop order can again originate either from the *RainSensor* or from the *UserInterface*.

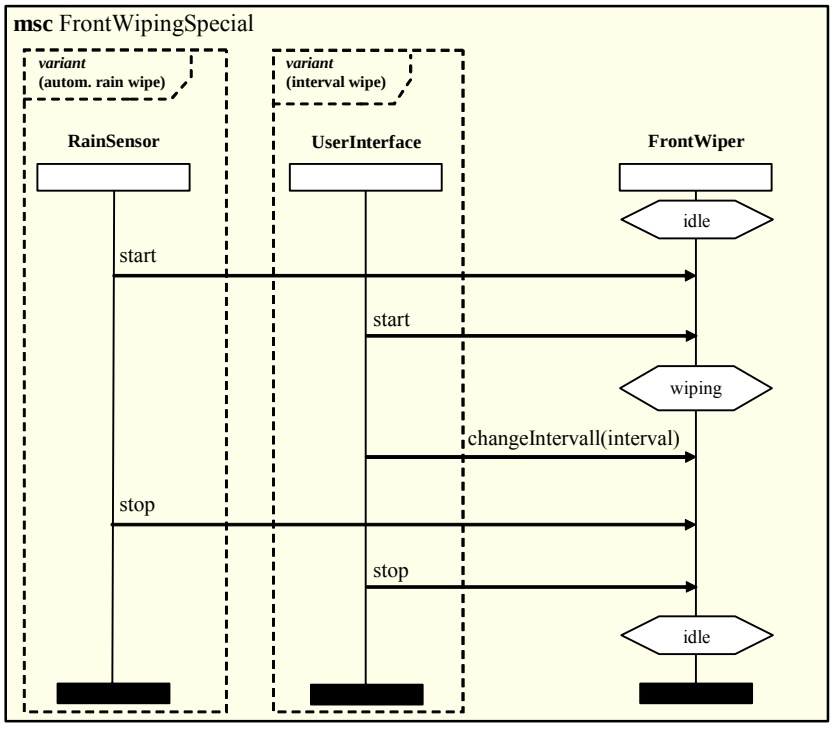


Figure 5 MSC FrontWipingSpecial as an example for variants in MSCs

3 MSC Connectors

We sketch the main principles and the extensions to the standard MSC language [8] with connectors; for a detailed description we refer to [4,5,7], further information can also be found in [3,6].

The reasons for the introduction of MSC connectors were twofold: On the one hand side, they are intended to provide a MSC language construct suitable for describing and reasoning about component interfaces and component interactions which allows abstraction to interactions in a similar way as the MSC reference construct or decomposed instances do for component behaviour. On the other hand, MSC connectors are meant to facilitate the message flow definition between those MSC constructs that currently use the gate concept. It turned out that both aims can be realized using the same language concept. In this case study, MSC connectors are used for both reasons, this means, to single out domain respectively application specific interactions between components (see the respective connector definitions in Figure 21 or Figure 33) as well as to abstract and generalize recurring interaction patterns between MSC constructs (see the connectors defined in Figure 13, Figure 14, or Figure 20).

MSC connectors represent the structure of an interaction between two (or more) MSC constructs which in this section will summarily be called components in order to facilitate the presentation. It should however be kept in mind that these "components" are either MSC components – usually described by entire MSCs, MSC references in HMSCs or decomposed instances – or arbitrary MSC constructs (like inline expressions or MSC references used to structure parts of the behaviour of a MSC). The common property is that components communicate with each other via their environment. Connectors match these environments and describe the details of the interaction structure.

Thus, the first of the MSC connector concepts is the extended environment of components: the standard MSC language only provides one monolithic environment. This is extended by introducing additionally *environmental instances* thus allowing partitioning of the overall environment in logically distinct and disjunctive "environments" that resemble the roles a component takes in its interactions. Environmental instances are indicated by the keyword ***env*** in the instance header (see Figure 6). It should be mentioned that environmental instances may be repeated as well.

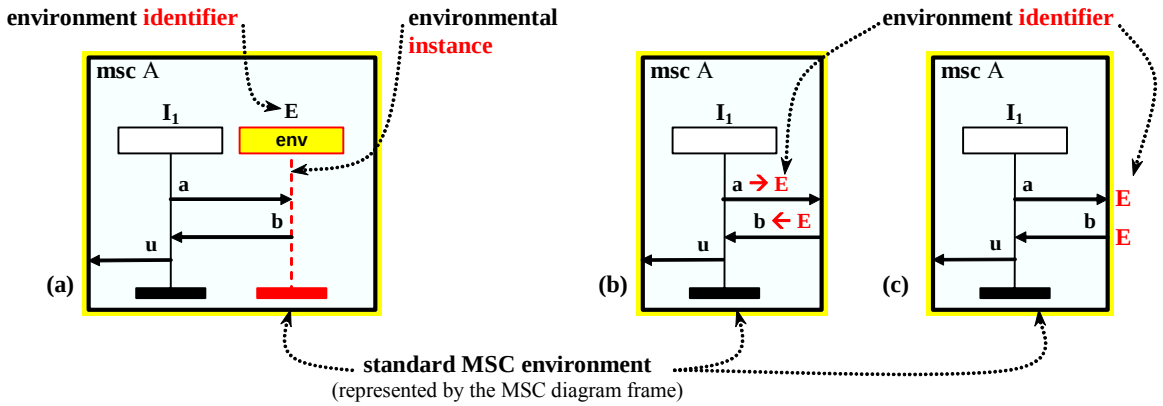


Figure 6 The structured environment and presentation variants: (a) Environmental instance *E*; (b) "Arrow mechanism"; (c) "Attached environment identifier".

Counterparts of the environmental instances in the MSC connectors are the so-called *external instances*. From a logically point of view, they represent component roles involved in the interaction: they describe the message exchange with the components which actually corresponds to component messages sent to or received from the environment. External instances are identified by the keyword ***ext*** in the instance header (see Figure 7).

The MSC connector definition resembles the MSC reference definition save that the keyword ***msc*** is replaced by the keyword ***con*** (see Figure 7). External instances are allowed only in connectors. Beside this, connectors are built by the same MSC constructs that are allowed in the MSC references, in particular, there may be *internal instances* (these are the usual instances which are neither external nor environmental) in order to allow message transformations. Connectors may also contain environmental instances.

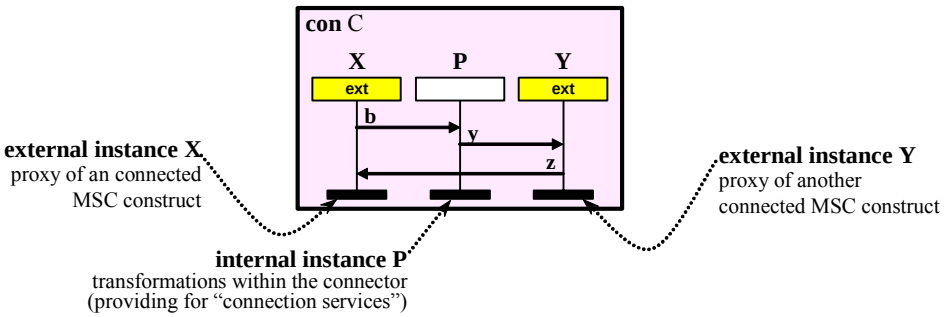


Figure 7 Definition of the MSC connector *C*.

The MSC connector application is syntactically expressed by a double lined arrow. Either one of the arrow heads touches the involved instance axis, or a tiny circle above the crossing of instance axis and connector symbol indicates that the respective instance is part of the interaction (see Figure 8). Using a MSC connector to connect components now requires identifying the component environments with the connector roles, this means, to associate external instances in the component MSC construct with external instances in the MSC connector. The mapping is not necessarily one-to-one: for instance, environments that describe roles distinguished by one component may be mapped onto one external instance because earlier the interaction was defined without this distinction but the new component has to be compatible with it nevertheless. This *external instance mapping* is described explicitly by listing the associated external and environmental instance names where the connector meets the connected component (see Figure 8b).[3]

Since MSC connectors are intended as abstraction means for interaction patterns, they cannot only refer to concrete message names. Thus, MSC connectors contain message variables which are bound to the concrete messages exchanged with the components via *message mapping*. The message mapping specifies exactly how message variables can be substituted by the names of the concrete messages. It is enclosed in square brackets and is placed below the connector symbol (see Figure 8a).

[3] The general definition of the external instance mapping reflects the need for a flexible connector application, but ensures the correctness of the association:
For each connector endpoint and the respective MSC construct, a non-empty subset of a partition of the external connector instances has to be mapped injectively into a partition of the environmental instances of the MSC construct. There is no message exchange among instances of the same partition element.

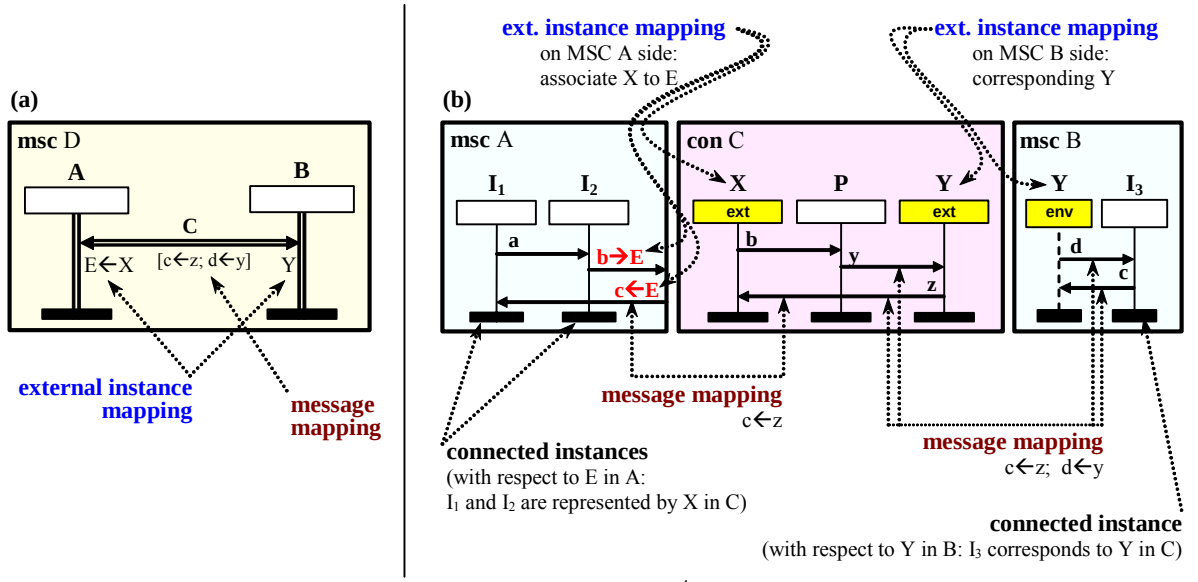


Figure 8 MSC connector *C* connecting the decomposed instances[4] *A* and *B*: (a) the connector application; (b) the respective definitions of *A*, *B* and *C*.

The behaviour defined by a connector application is the result of a merging of the behaviours specified by the MSC connector and the components. The *basic* merging procedure can be defined in the following manner:

> *Each connector in/out message event on an external connector instance has to be identified with a corresponding equally named in/out message event that is received from, respectively sent to one of the environmental instances in the connected MSC that are associated to the external connector instance through the external instance mapping.*
> *The message mapping has to be performed prior to the merging.*

For this merging procedure, it is assumed that messages with identical names also have to correspond with respect to their parameters.

As already stated above, besides the particular constructs for the connector concepts, all other standard MSC constructs can be used in a connector definition. This is also the case for the repeat construct which however has to conform to the number of repetitions required by the connected components; otherwise, it has to inherit its cardinality through the connector application. There is also no particular restriction to the language constructs for variation discussed in this paper (see Figure 9).

[4] In analogy to the graphical presentation of the connector, we propose to denote a decomposed instance with a double lined instance axis.

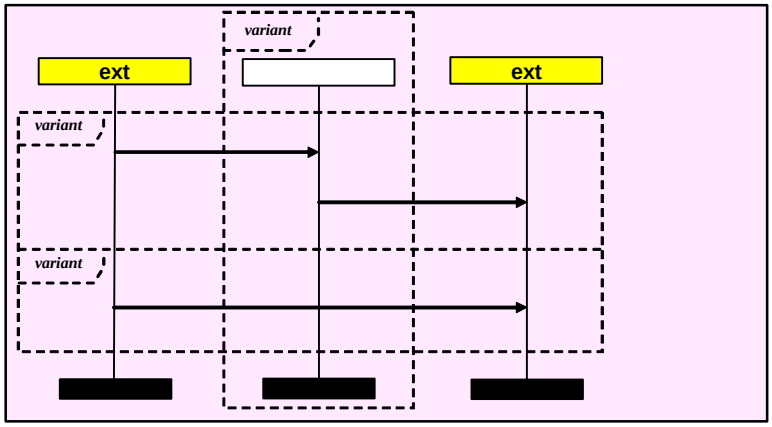


Figure 9 Variants in MSC connectors

Part II
Case Study

4 Description of the Basic Production System

Our case study is based on a specification [2] and a corresponding design [1] that describe a production system for engine parts. The aim of the original design was to explore the applicability of the UML-RT language [9] on the description of the system. We take the same example – without major modifications – as the initial asset for our system family, first, to explore the applicability of MSC connectors for the system family design purposes and then, to investigate into the representation of variation points within MSCs.

The engine parts, more generally called work pieces, are deburred and washed by the system. In the basic variant of the system, as described in [1, 2], three machine tools are used for deburring and washing the work pieces, and three autonomic transport vehicles, also called holons or, for short, HTFs[5], carry the work pieces between the machine tools. Furthermore, there are input and output storages, which buffer the work pieces on their way from and to the environment of the system. Of course, the holons move the work pieces also between storages and machine tools.

The basic scenario of the production system is the following: Each work piece processed by the system is taken from the input storage, treated by the first, second, and third machine tool in this order and finally delivered to the output storage. The machine tools have in addition to their workplace a buffer for two untreated work pieces.

All parts of the system synchronise by broadcast communication over a radio system. This makes the system extremely flexible. To keep the system to a neat size, we assume according to the specification that the communication is failure-free.

Each holon is only able to carry one work piece at a time. Furthermore, each holon has an internal database that contains a complete copy of the state of the working process. They use this database to make their decisions and keep it up to date by exchanging broadcast messages.

Figure 10 shows the major use cases of the system corresponding to the major scenarios described in more detail below (following [1]).

[5] This abbreviation comes from “holonic transport vehicle”; but since the original system was described in German, the characters stand for “Holonisches Transport Fahrzeug”. In order to keep the relation to the original study visible, we did not adapt the abbreviation to the English.

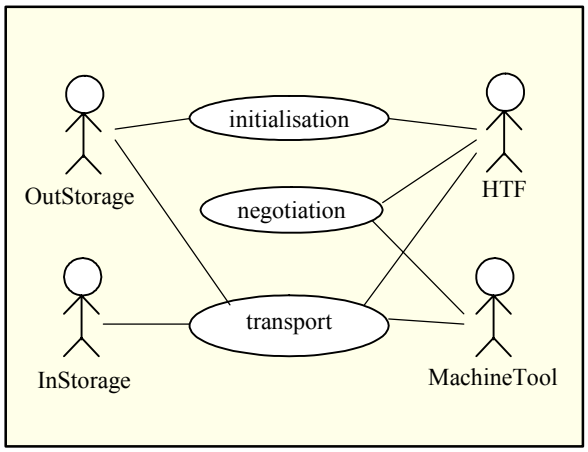


Figure 10 Use case diagram

A. Initialisation of the production:

1. One of the holons asks the output storage about the work plan, i.e. how many work pieces should be treated. It is not defined which holon starts the initialisation.
2. The output storage sends the work plan via broadcast to everybody.

B. Negotiation of jobs between a machine tool and the holons:

1. A machine tool or input storage posts a job to carry away a (treated) work piece and simultaneously sends its status.
2. The holons receive the message and update their database. They start to compute a bid.
3. The holon that has the first available bid sends its bid. The other holons listen and only send a bid if they can do better. If no holon sends a bid, the machine tool will post the job again after a certain time.
4. After a fixed time the machine tool ends the negotiation and the holon with the lowest bid is assigned to the job.

C. Transport of a work piece:

1. A holon requests the necessary resources for the transport in arbitrary order.
   - A holon requests a work piece from the input storage or a machine tool, respectively.
   - A holon requests a place from the output storage or another machine tool.
2. The release of the resources is reported. The following actions can occur in arbitrary order.
   - A machine tool or the input storage releases the work piece.
   - The output storage or the other machine tool releases a place.
3. The holon transports the work piece and acknowledges the successful work piece transfer via broadcast.

The class diagram in Figure 11 is based on [1]. It describes the static structure of the system with a focus on the classes used in later sections. Furthermore it defines a basic terminology for the further discussion of the system. We reduced this diagram to an overview of just the important classes in consideration. An *HTF*, also called holon, includes a single *Database*, transports *WorkPieces* and executes *Jobs*. The *Jobs* are defined in the production program *ProdProg* and involve two *Locations*.

A *Storage*, as well as a *MachineTool*, is at a specific *Location*. *InStorage* and *OutStorage* are both a special kind of *Storage*.

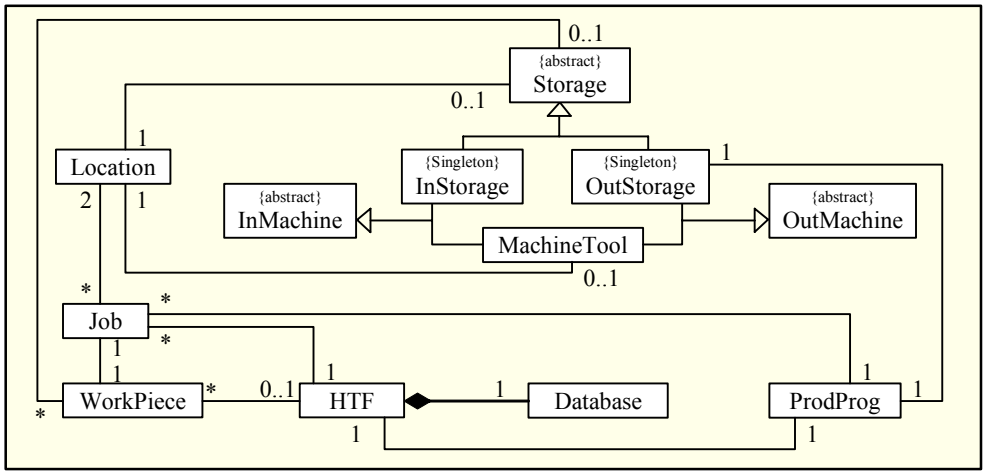


Figure 11 Class diagram

We define *InMachine* as the superclass of *MachineTool* and *InStorage*, and *OutMachine* as the superclass of *OutMachine* and *MachineTool*, because they often serve the same purpose in the MSCs. Considering implementation, the multiple inheritance structures can be problematic, in case of equally named methods or attributes in both superclasses. In this study we assume no naming collisions.

5 Basic System Design

The following MSCs describe the scenarios of the use cases from Section 4 in greater detail. First we develop a high-level view over the system, and then two refining iterations are performed. Each section is composed of an HMSC to show the interrelations of the MSCs, the used connectors, and some exemplary MSCs.

5.1 High-level View

The high-level view describes the interplay of the physical parts of the system, i.e. the communication of HTF, machine tools, input storage, and output storage. It must be noted that all these MSCs send their broadcast messages to the normal recipients as well as to the unspecified environment (i.e. the standard MSC environment, as defined in [8]). This simply means that there could be other components in the environment of the system that might be interested in these messages.

First the system starts the initialisation, and then a machine tool or the input storage posts a new order to negotiate. Afterwards the HTF that won the bidding process takes the work piece, moves it to the destination and releases the work piece. Finally the system waits for a new order to repeat the sequence. Figure 12 shows the HMSC describing the overall behaviour of the system.

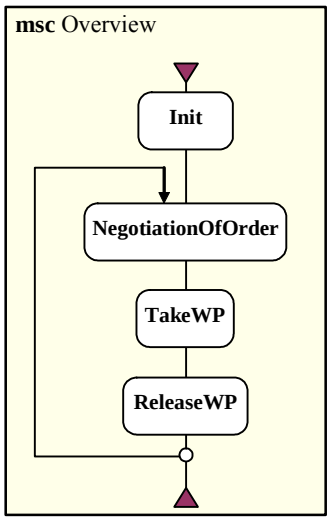


Figure 12 MSC Overview

The corresponding MSCs to the HMSC are depicted in the following but first the associated connectors are described. The connector *requestReply* in Figure 13 is simple but powerful. It represents two messages between two components. It means that one of the components requests information from the other and receives a reply. Hence the external instances are called *Req* (for Requester) and *Rep* (for Replier). The request/reply type of communication is common in distributed systems and especially in this production system. The well-known naming of this connector [12] makes it easy to understand when it is used in an MSC as can be seen in the following.

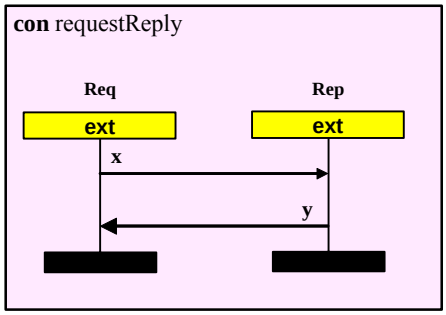


Figure 13 CON requestReply

Broadcast is an important concept that is hard to express in standards MSCs and UML [24] and is also typical for connectors [12]. The MSC connector in Figure 14 shows the simple and generic modelling of a broadcast (or multi-cast) communication which is often used in the system. The connector contains a single message that is sent to an arbitrary number of components. This is realised by the use of the parameterised repeat construct. The parameter must be set when using the connector. The external instances are called *Sender* and *Rec* (for Receiver).

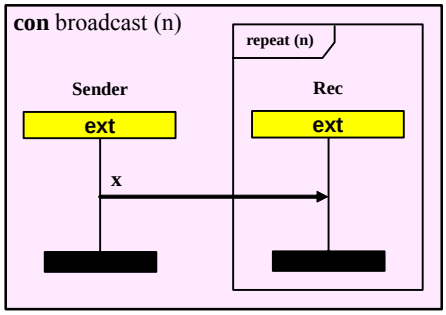


Figure 14 CON broadcast

An interesting issue for tool support is that a CASE tool could ensure the consistency between the number of repeated components inside the connector and the number of components connected with the connector when used in an MSC.

Having described the necessary connectors, we detail the scenarios in the following. The initialisation process of the system is described in the MSC in Figure 15. One HTF broadcasts a request for the production program to all participants. It is non-deterministic which HTF sends the message. The *OutStorage* answers by broadcasting the production program. The MSC uses the connector *broadcast* introduced in Figure 14 above. It can be seen that the connector allows a simple modelling of broadcasts. Only the number of recipients has to be passed as a parameter. Furthermore the repeat construct keeps the MSC small and concentrated on the essentials.

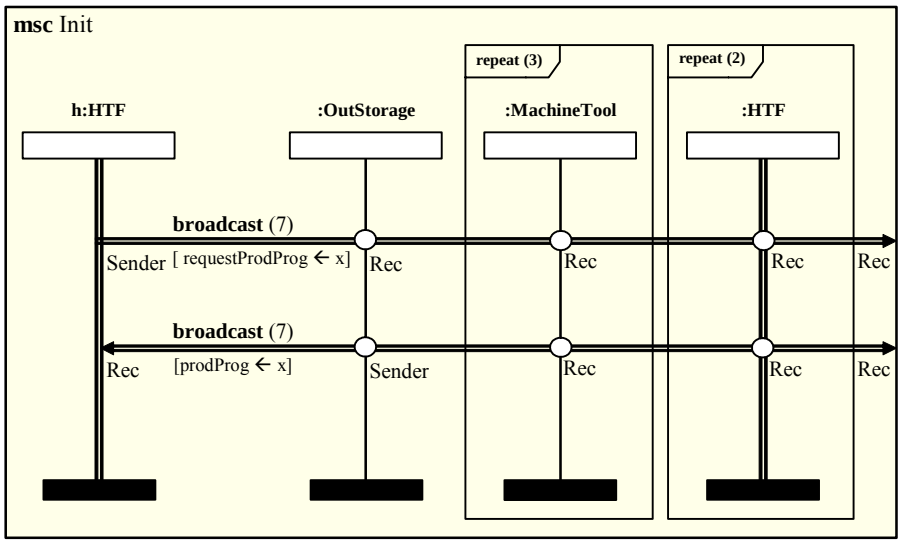


Figure 15 MSC Init

A simple example for the negotiation of an order is depicted in Figure 16. A *MachineTool* has an order to place (i.e. a new job to be done) and therefore broadcasts the job order to all HTFs. Every HTF creates that job and one of them broadcasts a bid for it. It is assumed that the broadcasting HTF is the one that computed the bid most quickly. The other HTFs would only broadcast a bid if theirs were better (i.e. lower) than the first bid. In this MSC we assume that no other HTF has computed a better bid. It is also important to notice that an HTF is not allowed to underbid its own bid. After a certain period, the *MachineTool* ends the negotiation with a broadcast and the HTF save the status of the job. This means that the winning HTF can start to transport the work piece.

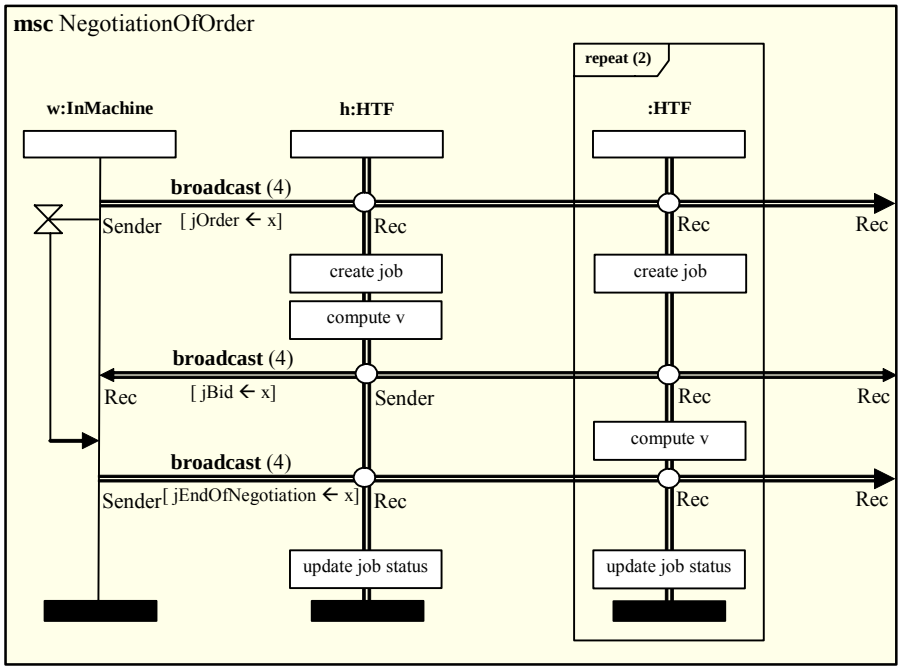


Figure 16 MSC NegotiationOfOrder

Figure 17 describes how a work piece is taken from the *InStorage* or a *MachineTool*. This implies that the HTF *h* won the negotiation for the transport of that work piece. First the responsible HTF *h* drives to the origin location. It requests the work piece from the *InMachine* (i.e. the *InStorage* or the *MachineTool*, see Figure 11) which answers by releasing the work piece. This is handled in a request/reply-like manner, thus we introduce the connector *requestReply* as described in Figure 13. The HTF broadcasts to all other HTF its transportation job and they update their job databases.

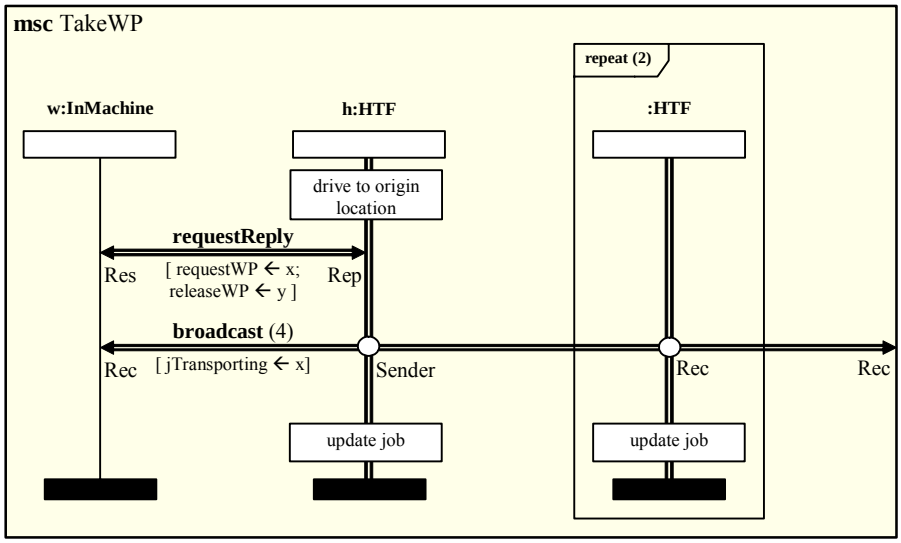


Figure 17 MSC TakeWP

The MSC for releasing a work piece is described in Figure 18. It is similar to and follows chronological after the MSC *TakeWP* in Figure 17. Here it is assumed that the HTF *h* is transporting a work piece. The transporting HTF drives to the destination location, requests a place for the work piece, which is released by *OutMachine* (which can be *OutStorage* or a *MachineTool*, see Figure 11). This is again communication in a request/response style and therefore the *requestReply* connector is reused. Afterwards the HTF broadcasts to all other HTF the completion of the job and they update their job databases again.

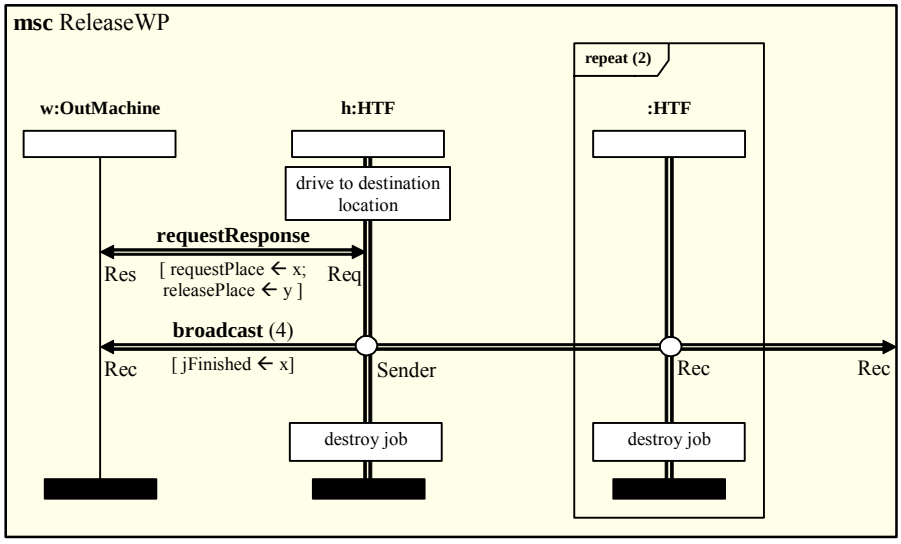


Figure 18 MSC ReleaseWP

5.2 First Iteration: Refining the HTF

In the first iteration, we concentrate on the behaviour of a single HTF because it contains most of the logic of the system, i.e. databases of the working process and handling of the negotiations. It is refined into the four components *Database*, *Disponent*, *SingleJobControl*, and *IOSystem*. The *Database* stores all the relevant incoming information and information about computed bids. The *Disponent* is responsible for handling the negotiation of orders, the *SingleJobControl* for executing a single job at a time and the *IOSystem* is responsible for the communication with the environment.

The HTF behaviour must provide its part of the system behaviour described in Section 5.1. It first initializes the process by receiving the production program, possibly by requesting it beforehand. Then four activities run in parallel: The HTF negotiates orders, waits for a job and executes it, listens for transporting messages and listens for finishing messages. In case it negotiates, the HTF can either be in the listener or originator role, that means it has a lower bid than is currently the best and sends it or just listens to the other bids, because its own bid is to high. If it has the lowest bid, the arrival of a new job will be noticed, and it will take the work piece, transport it to the destination and release it there. The HMSC in Figure 19 shows the life-cycle of an HTF.

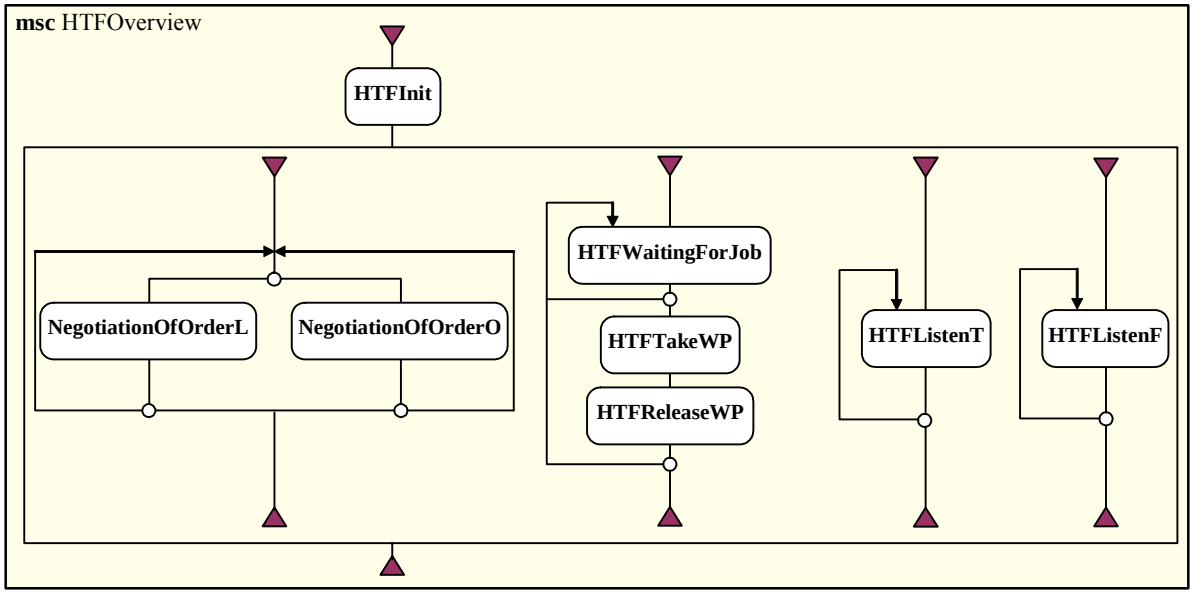


Figure 19 MSC HTFOverview

We again provide descriptions of the used connectors at first and then describe the MSC from *HTFOverview* in more detail. The connector in Figure 20 is actually quite similar to the broadcast connector in Figure 14, but it has an explicit instance between the external components that receive the message from the sender and forwards it to all the other receivers. The *forwarder* can be set as a parameter. In our case this will be the *IOSystem*.

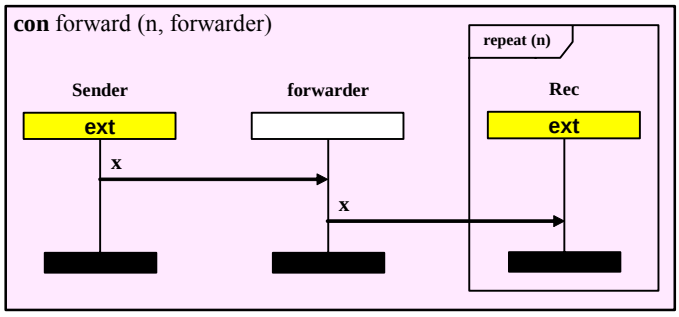


Figure 20 CON forward

The connector *getBids* depicted in Figure 21 shows in contrast to the more technically, message-oriented connectors, a more semantically interesting connector. It summarises the broadcasts of the bids during the negotiation. This connector shows that it is not only possible to group a technical issue, such as request/reply, to a connector, but also more application-oriented parts of an MSC. The connector hides the whole forwarding mechanism and the loop over the incoming bids, but transports that knowledge by using the name *getBids* to the reader of the MSC that uses the connector. It is parameterised with the minimal times the forward connector should be looped, i.e. the minimal amount of bids we want.

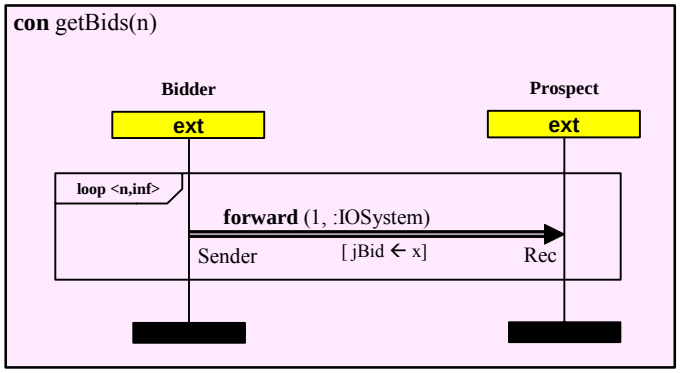


Figure 21 CON getBids

We start with the initialisation from the point of view of an HTF. The corresponding MSC is shown in Figure 22. First the request for the production program is broadcasted. This part is optional because only one of the HTF does the request and therefore not each HTF sends this message. In our design it is non-deterministic which HTF requests the program. In any case the database of the HTF receives the broadcast of the production program from the output storage. The database stores that production program and afterwards all components of the HTF are idle.

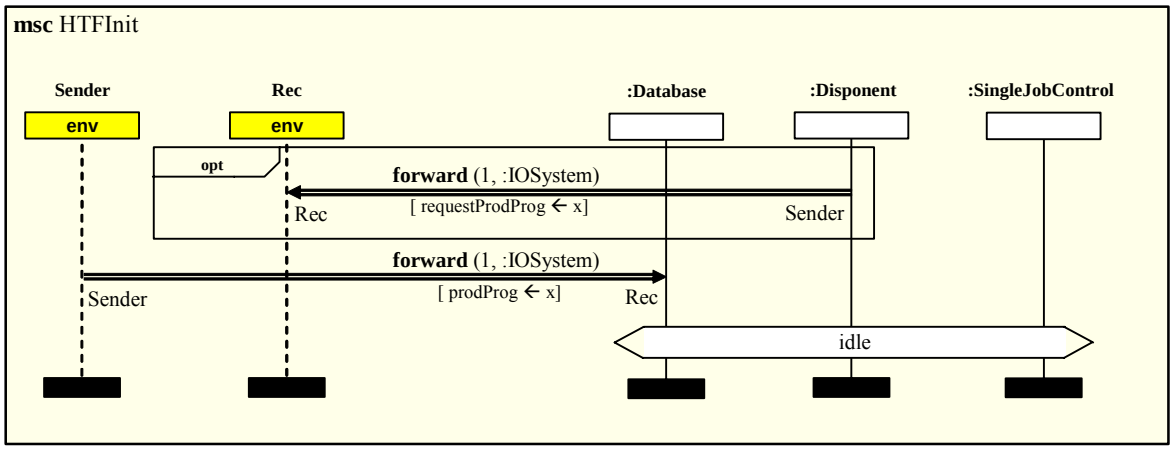


Figure 22 MSC HTFInit

The MSC *NegotiationOfOrderL* in Figure 23 describes how the negotiation of an order is handled inside an HTF that is in the listener role, i.e. does not place a bid. When the HTF is idle it firstly receives an order from the environment, then the disponent requests the status from the database. The disponent computes depending on that status its own bid. During the computation arbitrary many bids from other HTF may arrive. The arrival of these bids is encapsulated in the *getBids* connector (Figure 21). The bids are stored in the database. After finishing its computation the disponent asks the database for the current best bid. If the best bid is still better than the own bid, the HTF does nothing and just waits for the end of the negotiation, which is broadcasted from the machine tool that initiated the negotiation and returns to the idle state. To illustrate the improved conciseness of the MSC, the original MSC built without connectors is depicted in the appendix in Figure 50.

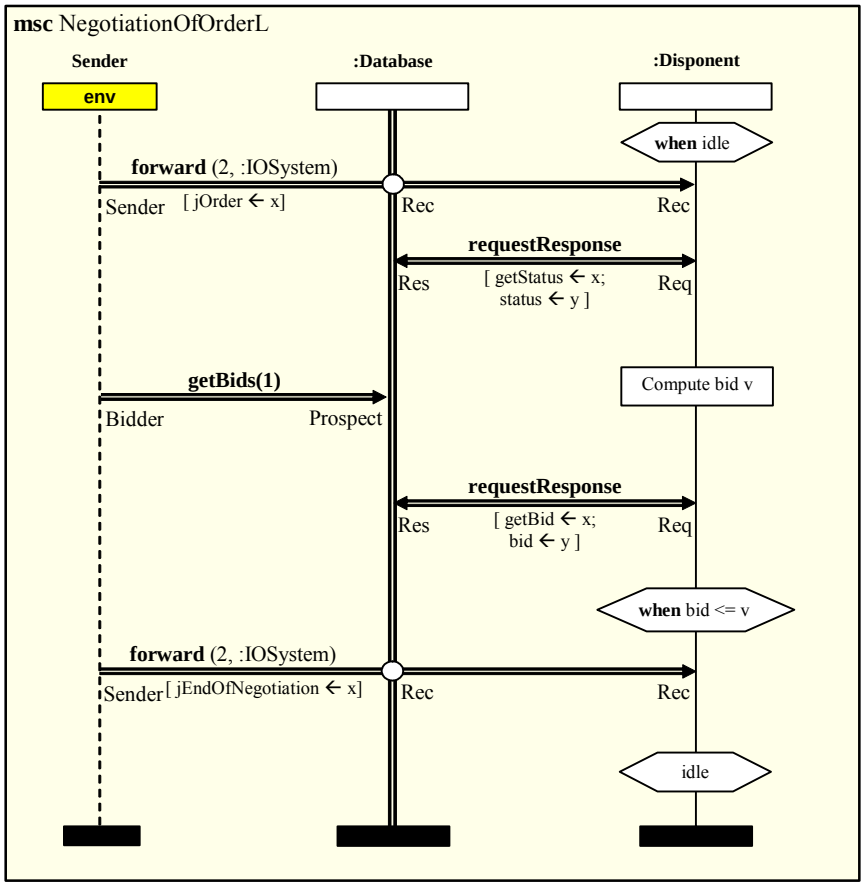


Figure 23 MSC NegotiationOfOrderL

Questions that arise looking at this MSC are if it makes sense to have connectors on this level of abstraction at all and to connect connectors to environmental instances. From our point of view it makes sense to have connectors here because it is also useful to structure the MSC on this level and make it more concise. It also makes sense to use a connector with an environmental instance because in some sense it is just an abbreviation for the real message passing. In a CASE tool it could be realised that the message passing is expanded by (double-) clicking the connector.

A negative point is that it can be confusing to have connectors on different abstraction levels, because the names of the roles can be equal or similar. Hence, it is not obvious at first sight to which connector an environmental instance belongs. This shows that good naming is important when using MSC connectors.

An alternative scenario is that the HTF under consideration places a bid in the negotiation process i.e. is an originator. In Figure 24 this scenario is described. The first part is the same as in Figure 23 but in this case its own bid is better than the currently lowest bid, therefore the disponent broadcasts the bid. Now again arbitrarily many bids can possibly come in (encapsulated by the *getBids* connector) until the end of negotiation is received. After that the disponent asks the database which HTF has made the best bid. If it made the lowest bid it inserts the job in the database. In any case it reaches the idle state again. The original MSC from [1] is again depicted in the appendix in Figure 51. It illustrates that the introduction of connectors improves the MSC in terms of conciseness.

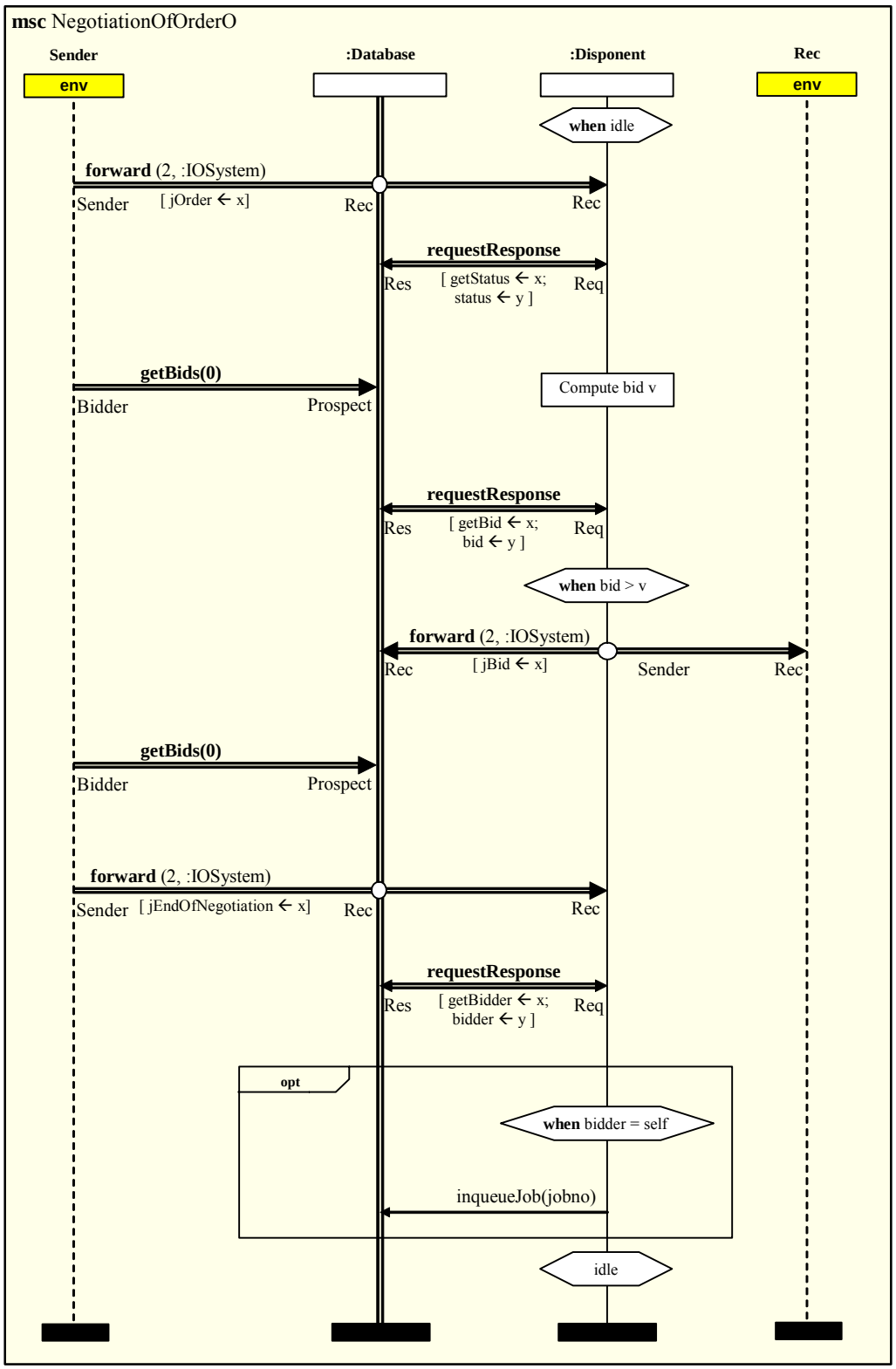


Figure 24 MSC NegotiationOfOrderO

The *SingleJobControl* uses a *busy-waiting* approach for the execution of jobs. When the *SingleJobControl* is idle it requests the next job from the *Database*. It either gets no job back if there

are no pending orders or gets a new job and changes to the *newJob* state. The latter state is the trigger for the MSC *HTFTakeWP* (Figure 26) to start transporting a work piece. The trace is depicted in Figure.

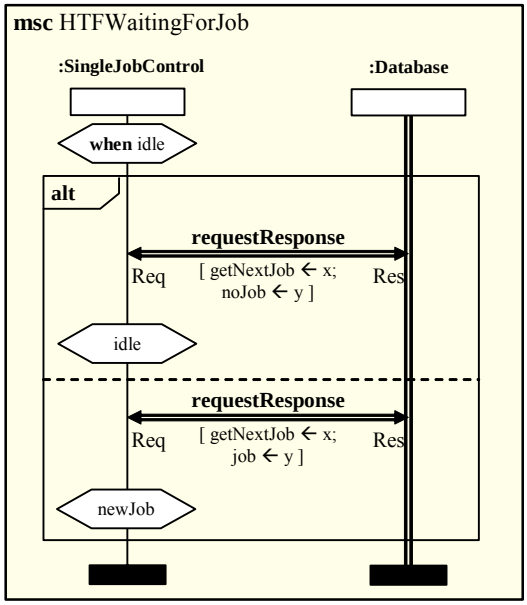


Figure 25 MSC HTFWaitingForJob

The MSC in Figure 26 starts with the condition that the *SingleJobControl* is in the *newJob* state; hence a new job has to be executed. The HTF drives to the origin location and requests the work piece. Then it broadcasts that it is transporting a work piece and enters the *transport* state.

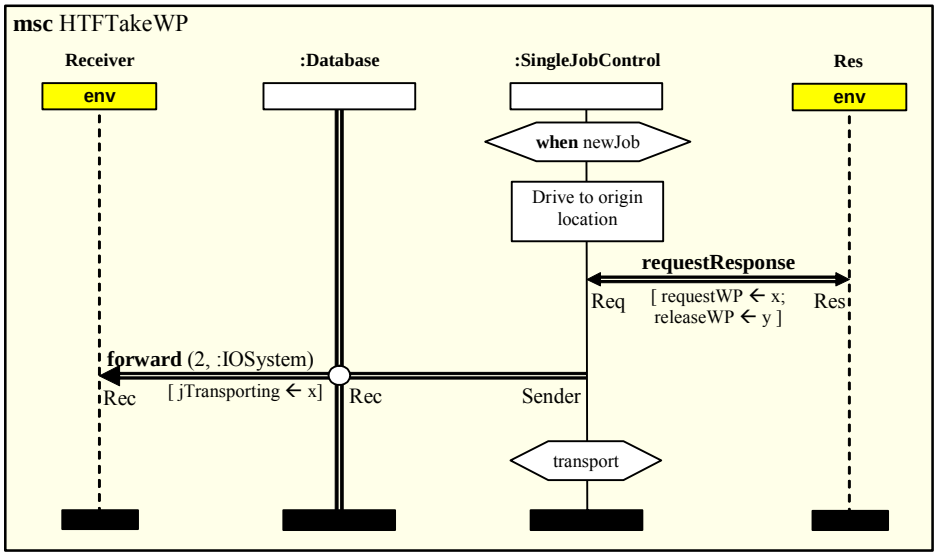


Figure 26 MSC HTFTakeWP

Following after the MSC *HTFTakeWP* the MSC in Figure 27 shows how a job is finished. It starts with checking if the current state is *transport*. The HTF drives to the destination location and requests a place for the work piece it is carrying. The environment, i.e. the machine tool or output storage, responds with the release of a place. The *SingleJobControl* then broadcasts that it has finished the job and returns to the *idle* state.

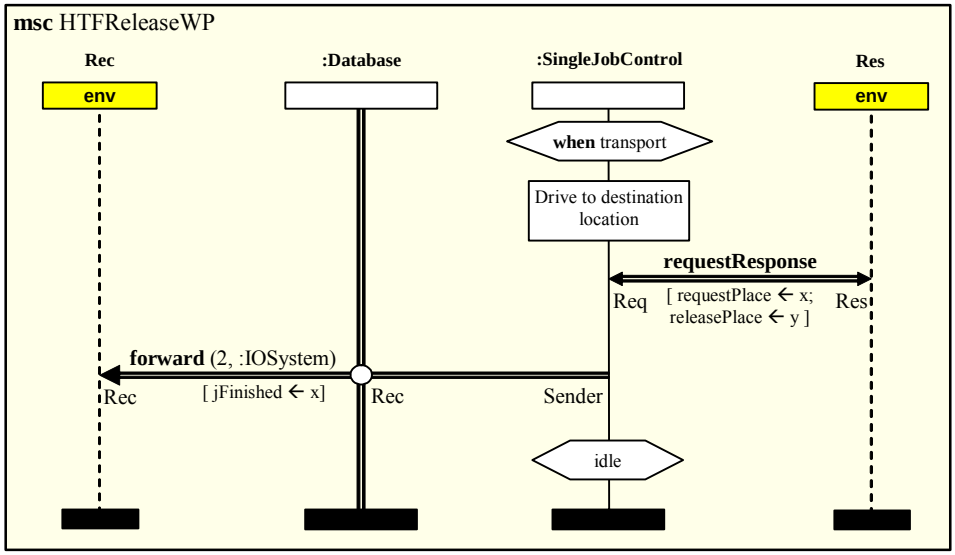


Figure 27 MSC HTFReleaseWP

The MSC in Figure 28 describe the listening of an HTF to the broadcast messages of the other HTFs. It basically shows that the *jTransporting* and the *jFinished* messages are sent to the internal *Database*.

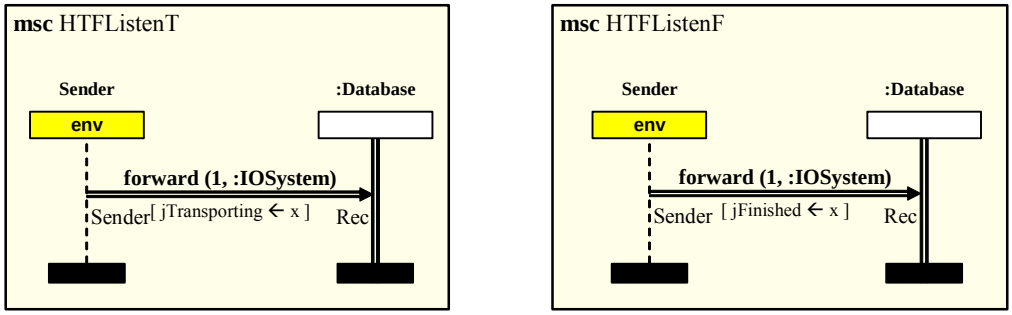


Figure 28 MSCs HTFListenT and HTFListenF

5.3 Second Iteration: Refining the Database

The next iteration has the aim to describe the database of an HTF in greater detail. In this context it is not completed, but only shows an example for a further refinement. The database of an HTF is chosen for the refinement because it has the most complex interaction.

Considering the aim of this case study, showing the whole refinement step is not useful, because it does not involve substantially new aspects. Therefore we describe one exemplary MSC with its used connectors. One interesting issue here is that the database uses its own *IOSystem* which packs the messages from the database into a kind of envelope and unpacks the incoming messages for the database. This is different to the other connectors as we use parameters inside connectors.

The connectors for this service are depicted in Figure 29. They only forward to a single component, in contrast to the forward connector in Figure 20. This connector allows one to ignore the packing and unpacking in the normal MSC.

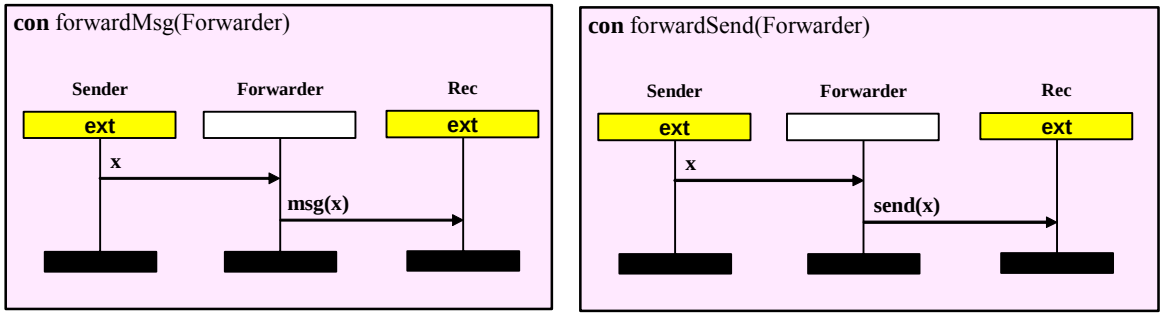


Figure 29 CON forwardMsg and CON forwardSend

The MSC connector *reqResForward* in Figure 30 is the combination of the two forward connectors above and combines them to a request/reply type connector similar to the *requestReply* connector in Figure 13. This shows again that this type of communication is common in communicating systems and can be elegantly modelled with a connector.

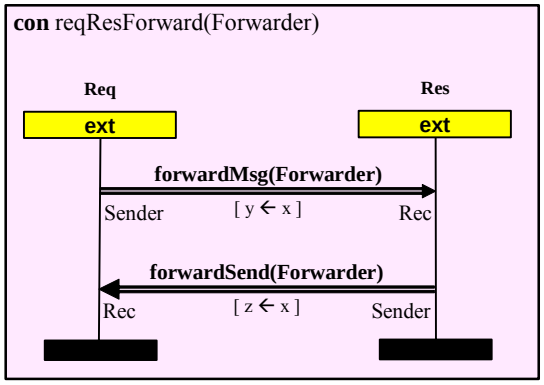


Figure 30 CON reqResForward

In Figure 31 the negotiation of an order is depicted from the point of view of the database of the originator HTF. At first the database receives the broadcast of an order. In case the job with the job number as in the order does not exist already in the database, a new job is created and the negotiation starts. The database requests the current status and stores it. Then two threads run in parallel: The environment (i.e. the disponent) requests the currently lowest bid and the database returns that value, and the job receives the bids from the other HTFs. The end of the negotiation is sent to the job and the job sends the lowest bidder to the environment. In case that the HTF containing the database is the lowest bidder, the database gets the message to in queue the job.

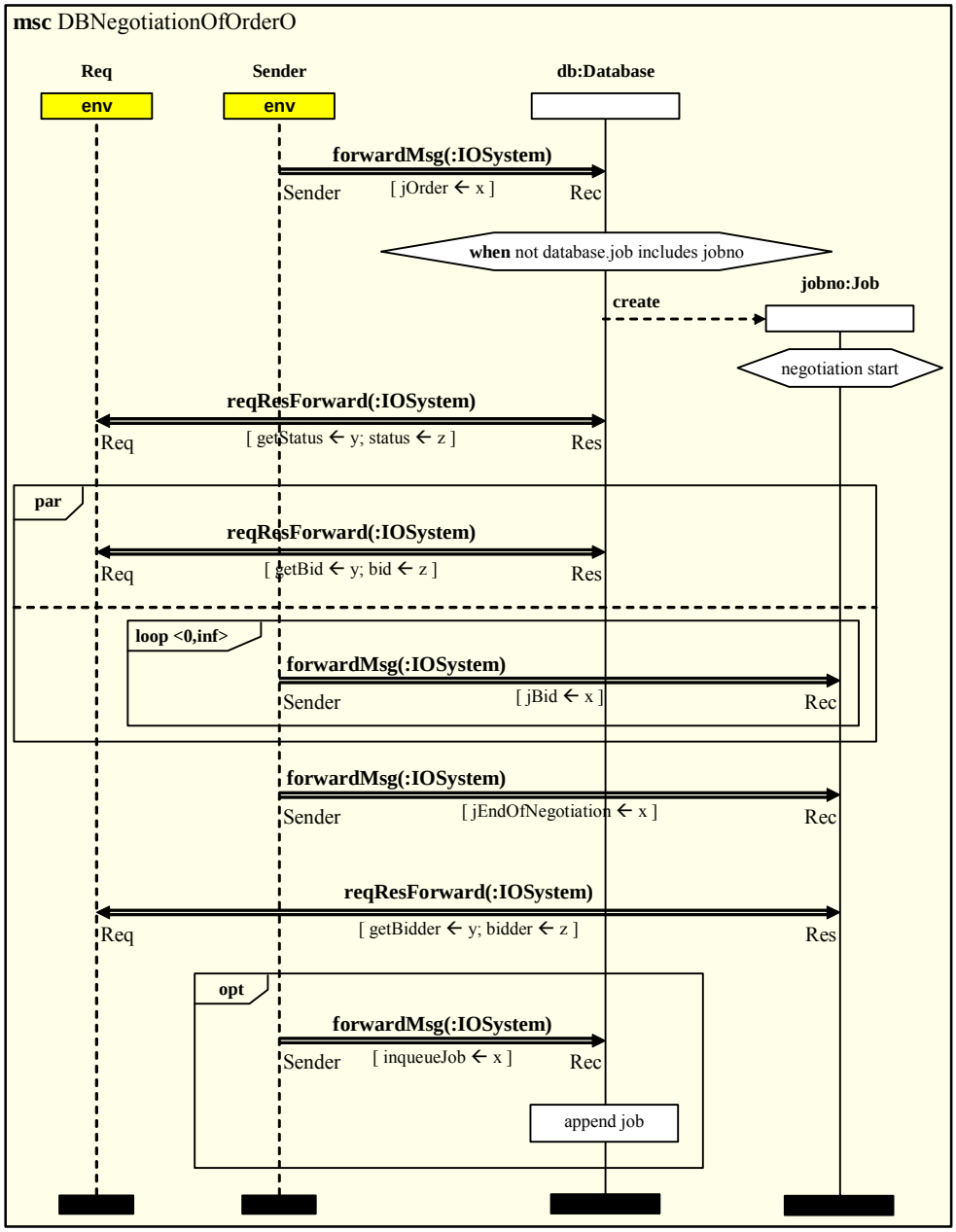


Figure 31 MSC DBNegotiationOfOrderO

6 Variations for a System-Family Environment

The issue explored in this case study so far is the usefulness of connectors in the description of a system. The overall aim of this study is however to examine MSCs and MSC connectors in the design of system families. To move further in that direction, we introduce variations of the system to build a system family. More specifically we define the scope of the system family.

Changing Amounts. Some simple changes in the system are increasing the amount of HTFs and/or machine tools. It results in a substantial change of what the system is capable of but requires only small changes in the MSCs developed above. These changes do actually not affect the connector

definitions but only the connector applications. Assumed we want to increase the amount of machine tools to 5 and the amount of HTF to 4, then we only have to change the parameter of the broadcast connector used in most MSCs. We also have to change the MSCs themselves to reflect the increased amount but that does not affect the connector. An example MSC that reflect both changes is the MSC *Init* depicted in Figure 32.

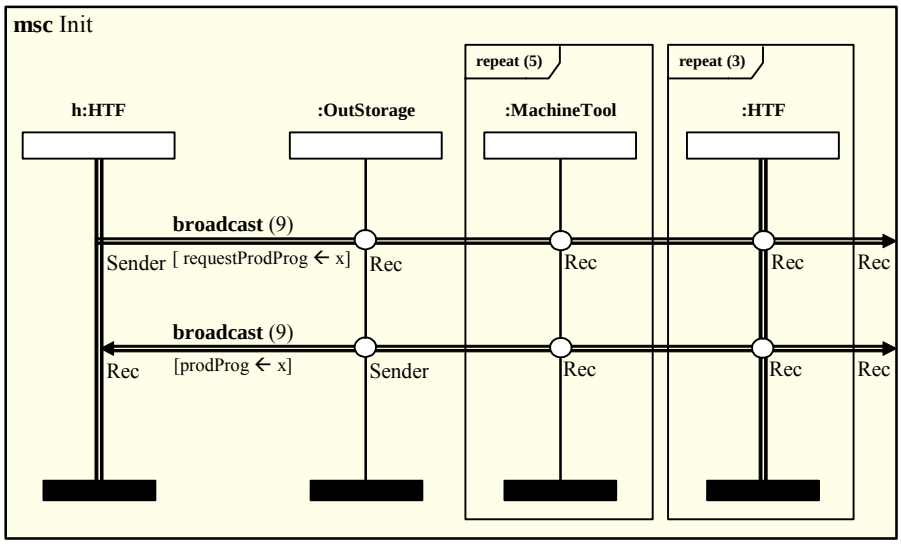


Figure 32 MSC Init (Variation)

Variable processing sequence. A more complicated but interesting variation is to make the sequence of the working process variable. This means that we do not have the same order of processing for all work pieces but every work piece has an individual processing sequence. We assume that the *InStorage* gets this information at the time it receives the new work piece and it is forwarded from the machines to HTF and the other way round. This implies the following changes:

- The message *jOrder* needs an additional parameter with the next location for the work piece.
- Either *relaseWP* and *requestPlace* need additional parameters to forward the processing sequence for the work piece, or a change in protocol and additional messages are incorporated into the connector. We look further into the second possibility because of the more interesting changes for the connectors.

The first change can not be expressed directly in the connectors because the parameters are not shown in the current design. The second change has effects on the MSC *TakeWP* (Figure 17) and the MSC *ReleaseWP* (Figure 18). That is where the advantages of the connectors come in. We do not fundamentally change the MSCs but replace the general connector *requestReply* with the more specific *transferWP*. This connector can be used in both MSCs because we have the exact message names as parameters and can define the transfer direction this way. The protocol addition to transfer the processing sequence can be encapsulated inside the connector. The new connector is depicted in Figure 33 and the application can be seen in the changed MSC in Figure 34.

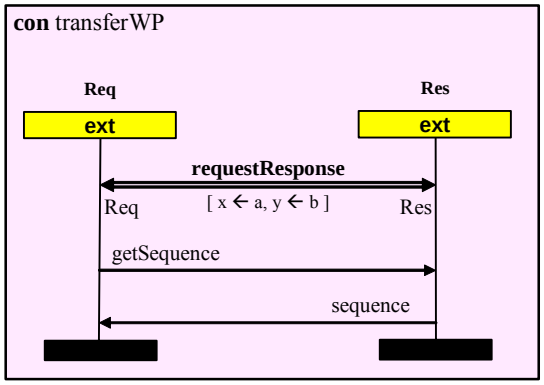


Figure 33 CON transferWP

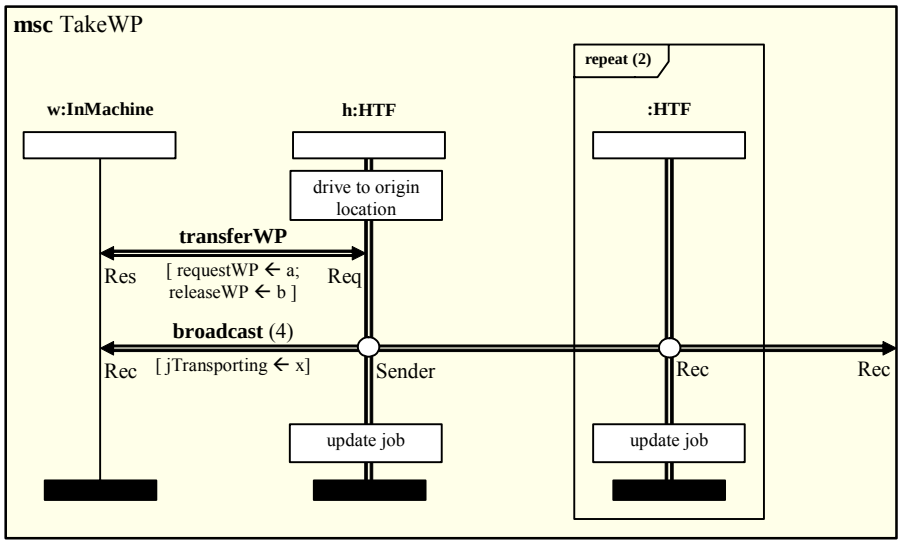


Figure 34 MSC TakeWP (Variation)

The only changes necessary are the replacement of the connector *requestReply* by *transferWP* and to change the message variable names to *a* and *b* because of a naming conflict. This shows that it can be a good idea to define connectors which have an application-oriented meaning, like *transferWP* here, and just encapsulate simpler connectors like *requestReply* to allow an easier introduction of variations. The change in *ReleaseWP* is analogous.

Fixed route for an HTF. Another variation is not to complicate the system but to simplify it by introducing an easier negotiation of order. We achieve that by assigning one HTF for every two machine tools. However, in the fixed sequence case it simplifies the negotiation of orders because the machine tool does neither have to broadcast the order but to send it to its assigned HTF, nor to wait for an answer. This assumes that an HTF has a buffer for the jobs (as it is the case in our design in Section 5.2). This change affects the three MSCs *NegotiationOfOrder*, *TakeWP* and *ReleaseWP*.

With the aim of changing the MSC from Figure 16 only slightly we can achieve the desired behaviour by just changing some numerical parameters. We set the parameter of the repeat construct for the other HTF to *0*; similarly we set the parameters of the second and the third connector to *0*. This has the effect that only one HTF remains and that the *jBid* and *jEndOfNegotiation* messages are not sent. The parameter of the first connector is set to *1* because only one HTF receives the message. However, that

approach is not elegant because it leaves the timer and the "compute v" and "update job status" in the MSC although they have no actual meaning. Thus we move everything below the first connector into an optional region in Figure 35 and allow an easy switch between system configurations.

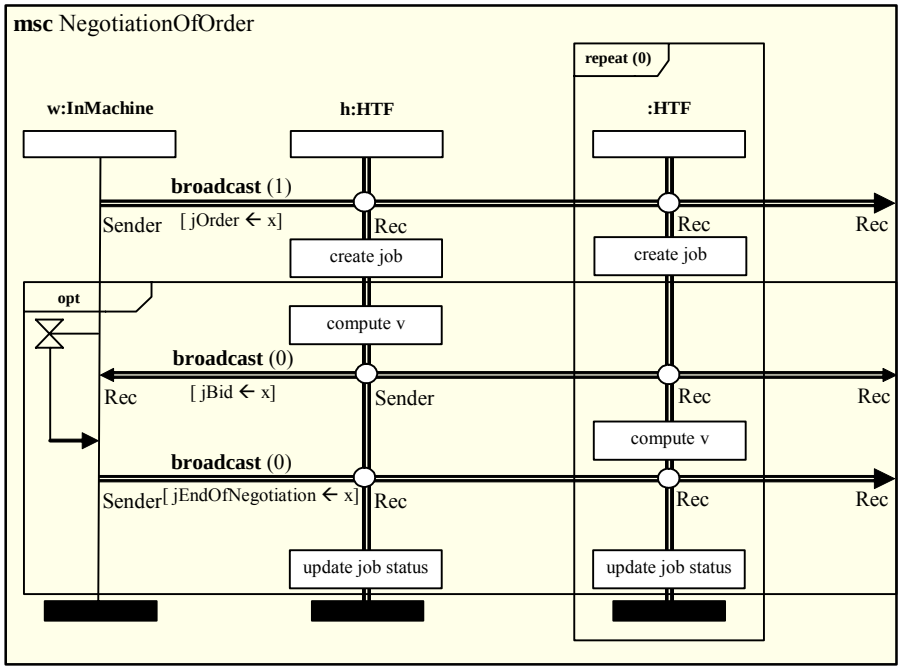


Figure 35 MSC NegotiationOfOrder (Variation)

The MSCs *TakeWP* and *ReleaseWP* do not have to be changed to conform to the behaviour but it is not necessary that all HTFs are informed that one HTF is transporting a work piece or finished a job. Therefore we change the MSCs using the same method as above, by setting the parameter of the repeat construct to *0* and reducing the parameter of the broadcast connector to *1* in Figure 36. The MSC *ReleaseWP* has to be changed accordingly.

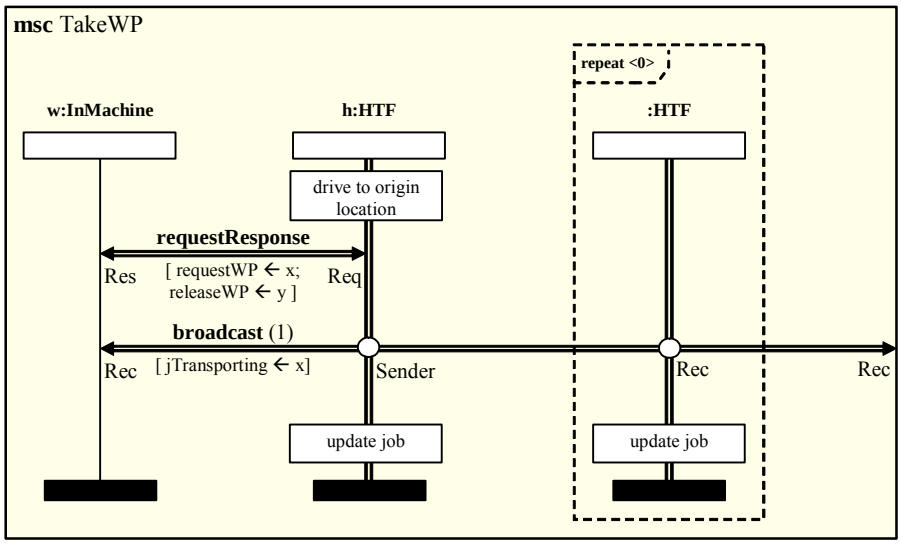


Figure 36 MSC TakeWP (Variation)

Central instance for negotiation of orders. If we want to keep the flexibility of the HTF but still want to simplify the negotiation process, we could introduce a central instance that distributes the orders. This would reduce the complexity of the HTF and the demand for the network. For this, we need to have a class *OrderDistributor* that keeps track of the positions of the HTF, receives the orders of the machine tools, decides which HTF has to execute the job and sends a message to it.

The MSC *NegotiationOfOrder* has to be changed quite extensively, which is obvious, because we changed exactly the negotiation process. We introduce the *OrderDistributor* which receives the order from the machine, selects an appropriate HTF, forwards the order, and sends an acknowledge message that the job is assigned to an HTF to the machine. Because of the extensive changes we give it a new name and call it *AssignmentOfOrder* in Figure 37.

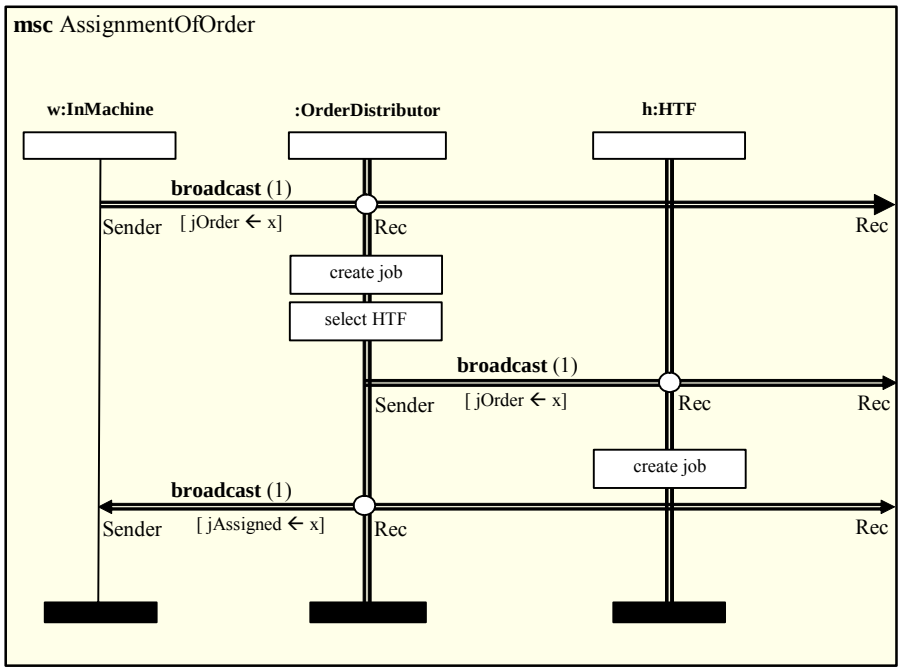


Figure 37 MSC AssignmentOfOrder

The MSCs *TakeWP* and *ReleaseWP* have to be changed as well but only slightly. We need only one HTF in the MSCs but add the OrderDistributor which is notified when an HTF drives to a destination and finishes a job. The altered MSC *TakeWP* is shown in Figure 38. The change in *ReleaseWP* is similar.

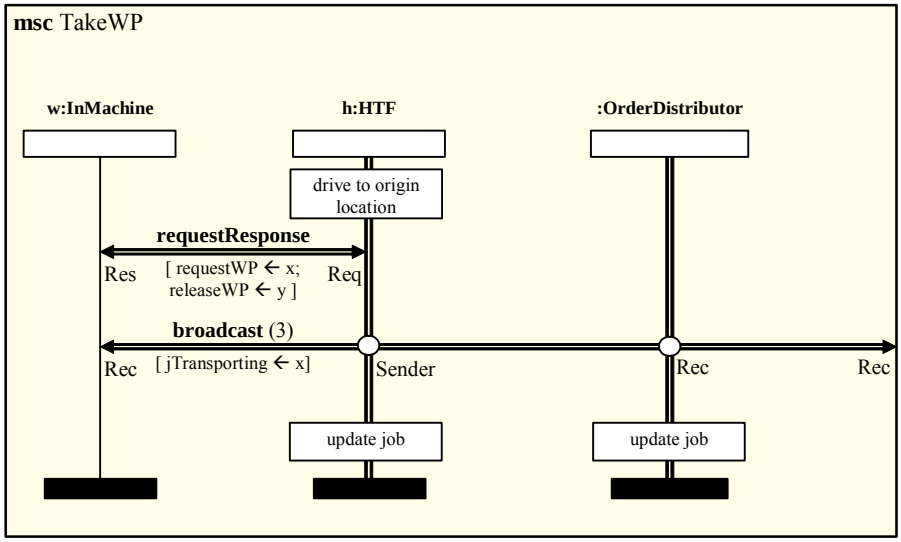


Figure 38 MSC TakeWP (Variation)

Several similar machine tools. It may be that there are slow and fast machine tools and then it would make sense to have several similar slow machine tools that are served by a fast one. This implies that we first have a negotiation between several machine tools before the normal negotiation process can begin.

This negotiation process is very similar to the original negotiation from the MSC *NegotiationOfOrder*. Hence, we can specify some connectors that could be reused in both MSCs in Figure 39.

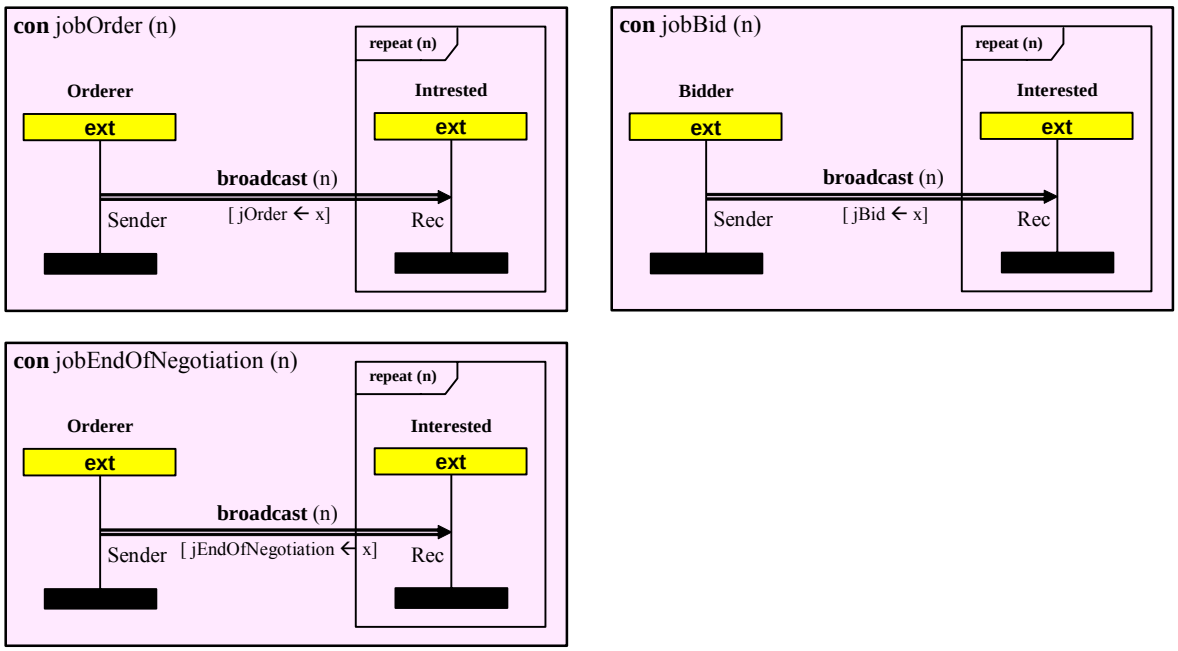


Figure 39 CON jobOrder, CON jobBid, and CON jobEndOfNegotiation

The use of the connectors can be seen in Figure 40 that describes the negotiation of orders between machine tools. The main benefit of the application-oriented connectors apart from reuse is that names can be chosen that are much more adequate.

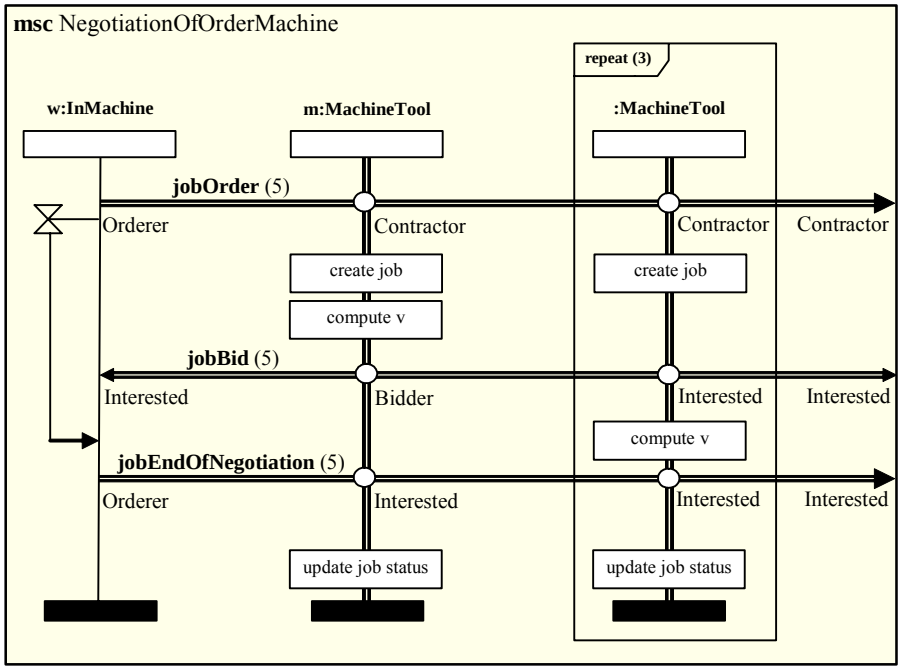


Figure 40 MSC NegotiationOfOrderMachine

Furthermore the same variation of the negotiation process as with the HTF can be introduced by having a central order distributor that handles the orders for the machines.

7 Family Model

In this section, we build a single model that incorporates all the features developed in Section 6. This model can then be used to describe the system family and be easily configured according to the desired variation. For this we make extensive use of the metaoperators defined in Section 2.2.

7.1 Features

At first we sort the variations found so far and determine the dependencies among them to finally get a basic system with various possible features. The system we started with had a fixed number of HTFs and machine tools; the transportation of the work pieces was organised using a negotiation process between the HTFs.

We introduce the possibility to change the number of both, the HTFs and the machine tools in the system. Actually, this offers two possible features: Either the numbers can be determined arbitrarily in advance (preset before delivery) or they can be modified during the production process. We decided to allow both possibilities in our system family. In order to facilitate the discussion, let us introduce a few abbreviations for the various features. The variability of the complete number of HTFs is called *HVarA*, that of the machine tools *MVarA*. The alternative features, that is to have different numbers of HTFs or machine tools set before delivery which cannot be changed by the customer later during system operation, are called *HFixedA* and *MFixedA*, respectively.

A variable processing sequence is added in contrast to the fixed order of processing in the initial system. We shorten it to *VarProcSeq* and consider it completely independent of the other features.

To simplify the system, we introduce a new feature: it only allows HTFs to serve a fixed, predefined route. This has the as a consequence that the negotiation process can be omitted. Obviously, in this variant, the feature *HVarA* cannot occur because there is no longer the possibility to input a route for a later added HTF. The shorthand term for this feature is *FixedRoute*.

Another possibility is to have a central instance (distributor) to replace the highly interactive negotiation process between the HTFs (*HDist*). Obviously, it is only necessary if the routes of the holons are managed, thus the *FixedRoute* variant and *HDist* cannot occur together. In addition, we can also allow such a distributor for machine tools if the negotiation process described in the following is not used. This feature is called *MDist*.

As mentioned above, the last variation we introduce is a kind of load balancing (*MLoadBal*) between machine tools with the same functionality. This implies that either *MDist* is required or some kind of negotiation process (*MNeg*).

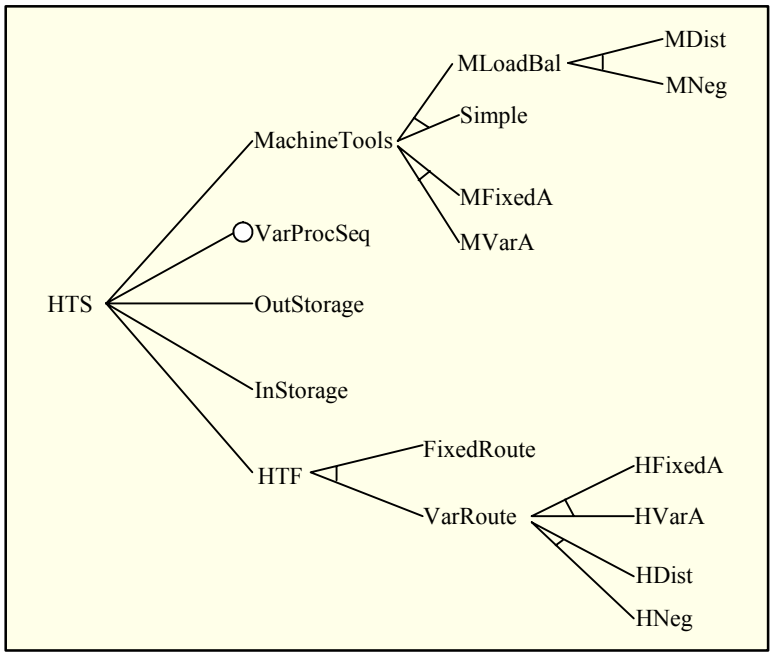


Figure 41 Feature model of the holonic transport system family

The dependencies of the introduced features are organised into a feature model following the notation from [10] in Figure 41. Normal edges in the graph are incident to mandatory features, edges with a circle at the end point to an optional feature, and edges with a vertical line between them connect alternative features.

7.2 Mapping of Features to Variation Points and Variants

We need to build the connection between the feature list and dependencies from Section 7.1 and the MSCs from the earlier sections to approach a family model that incorporates all the features. For this we introduce variants, variation points, and parameters that are later instantiated in the MSCs.

We need four parameters for the MSCs to describe differences in the cardinality of several components: *mt* is the number of machine tools, *ht* the number of HTF, *mod* the number of order distributors for machines and *hod* the number of order distributors for HTF. The exact description of these parameters in dependence of the feature model can be found in the appendix.

For the identification of the variation points and variants, only the feature model is needed. We can map the features onto variation points and variants following the approach from Section 2.1. This results in the following variation points using the notation from Section 2.2.

- HTS (VarProcSeq)
- HTF (VarRoute, FixedRoute)
- MNumber (MFixedA, MVarA)
- MTrans (MLoadBal, Simple)
- HNumber (HFixedA, HVarA)
- HLoadBal (HDist, HNeg)
- MLoadBal (MDist, MNeg)

These variation points have various occurrences in the MSCs that describe the system family.

7.3 High-Level View

This section describes a high-level view over the family model. We use the variation points and parameters from Section 7.2 to describe a general model for all possible configurations of the system family. To design a specific configuration, one has to choose the desired features from the feature model and derive the variation points and parameters. Using these, the corresponding MSCs can be constructed considering the variation point occurrence.

Figure 42 shows a high-level MSC that describes the general sequences in the system. We introduce the additional notation element *vp* to denote a variation point occurrence in the sequences.

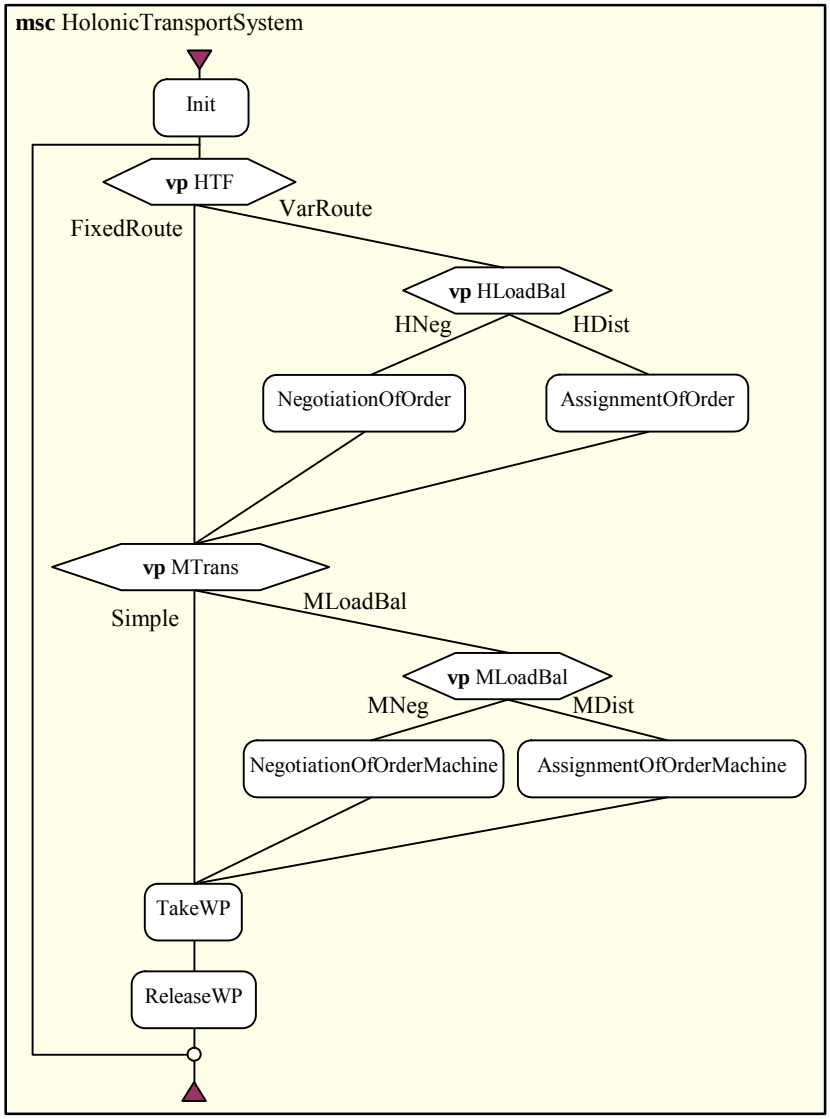


Figure 42 MSC Overview (Family)

In Figure 43 the general initialisation process is depicted. This is mostly the same for all configurations apart from the components and the cardinality of the components involved. It can be seen here that the *repeat* construct is most useful for the description of product lines in MSCs. All variations in cardinality and the presence and absence of components in specific MSCs can be modelled with it. We have four variation point occurrences for the variations that are modelled by these repeat constructs. However there is also a side-effect on the connectors in the other part of the MSC because it uses also the parameters from the variants occurrences.

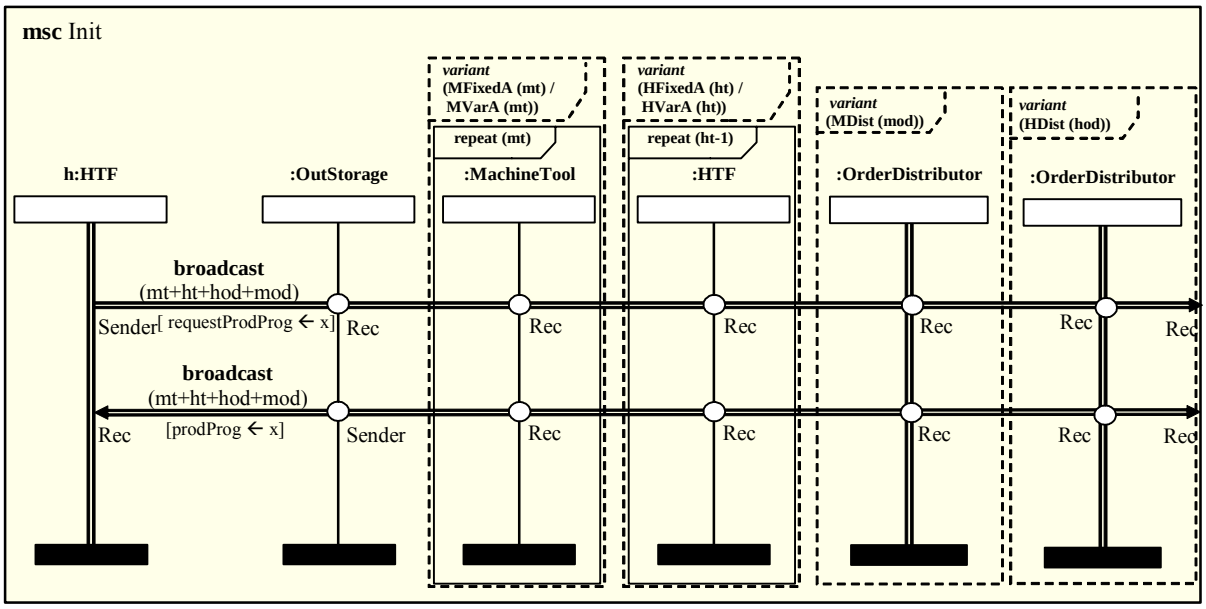


Figure 43 MSC Init (Family)

The next two MSCs show the order process for HTFs. Either negotiation of a machine tool with all HTF or an assignment through an order distributor is necessary. This is determined by the features *HNeg* and *HDist*. If there is an explicit negotiation process in the MSC *NegotiationOfOrder* is determined by the variant occurrence of *VarRoute* in Figure 44.

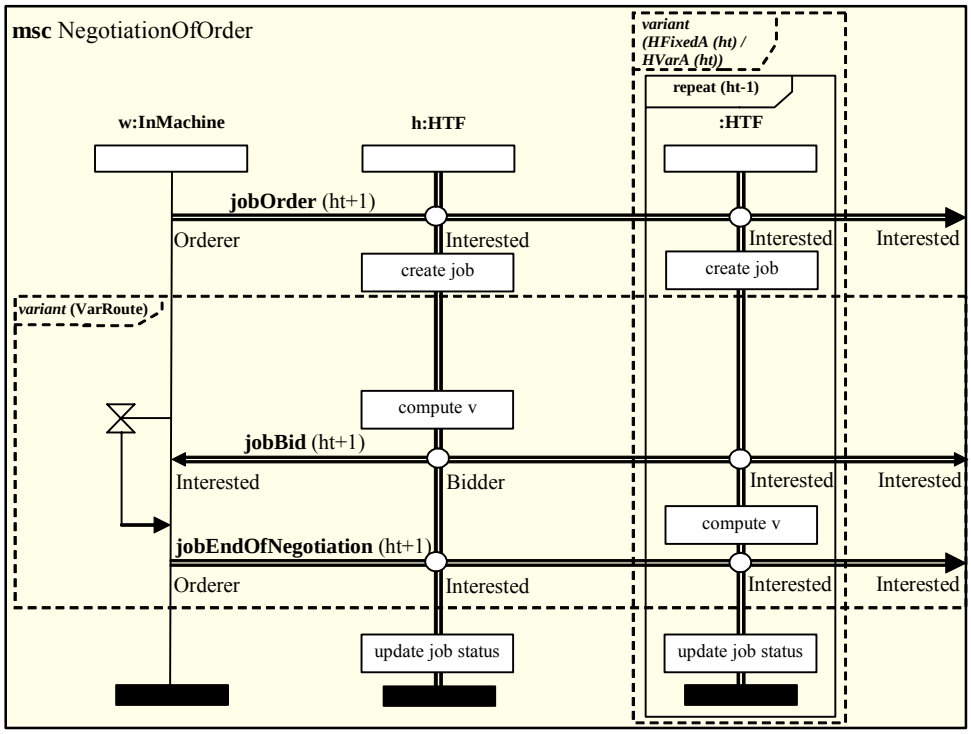
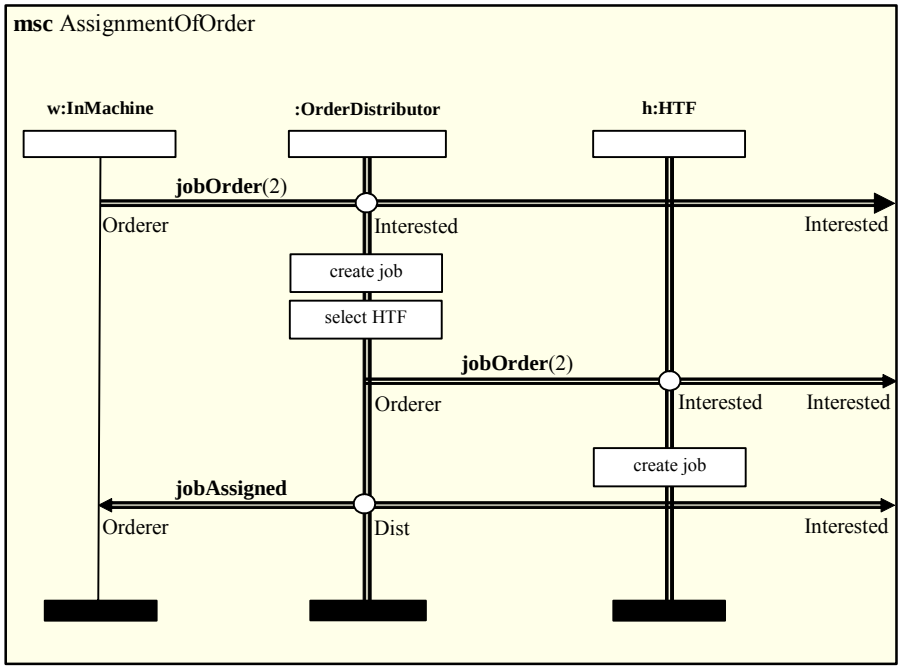


Figure 44 MSC NegotiationOfOrder (Family)

The MSCs *AssignmentOfOrder* and *AssignmentofOrderMachine* in Figure 45 do not contain any variant occurrences because the whole sequence is only used if the feature *HDist* or *MDist* is chosen, respectively.



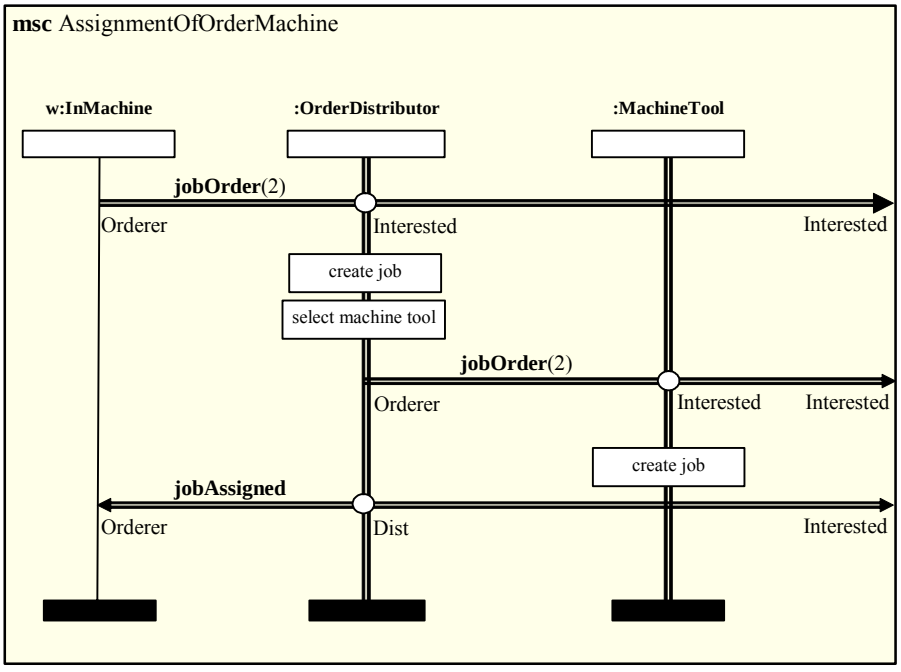


Figure 45 MSC AssignmentOfOrder (Family) and MSC AssignmentOfOrderMachine

The MSCs in Figure 46 and Figure 47 show again the usage of the repeat construct in variability modelling in this case study.

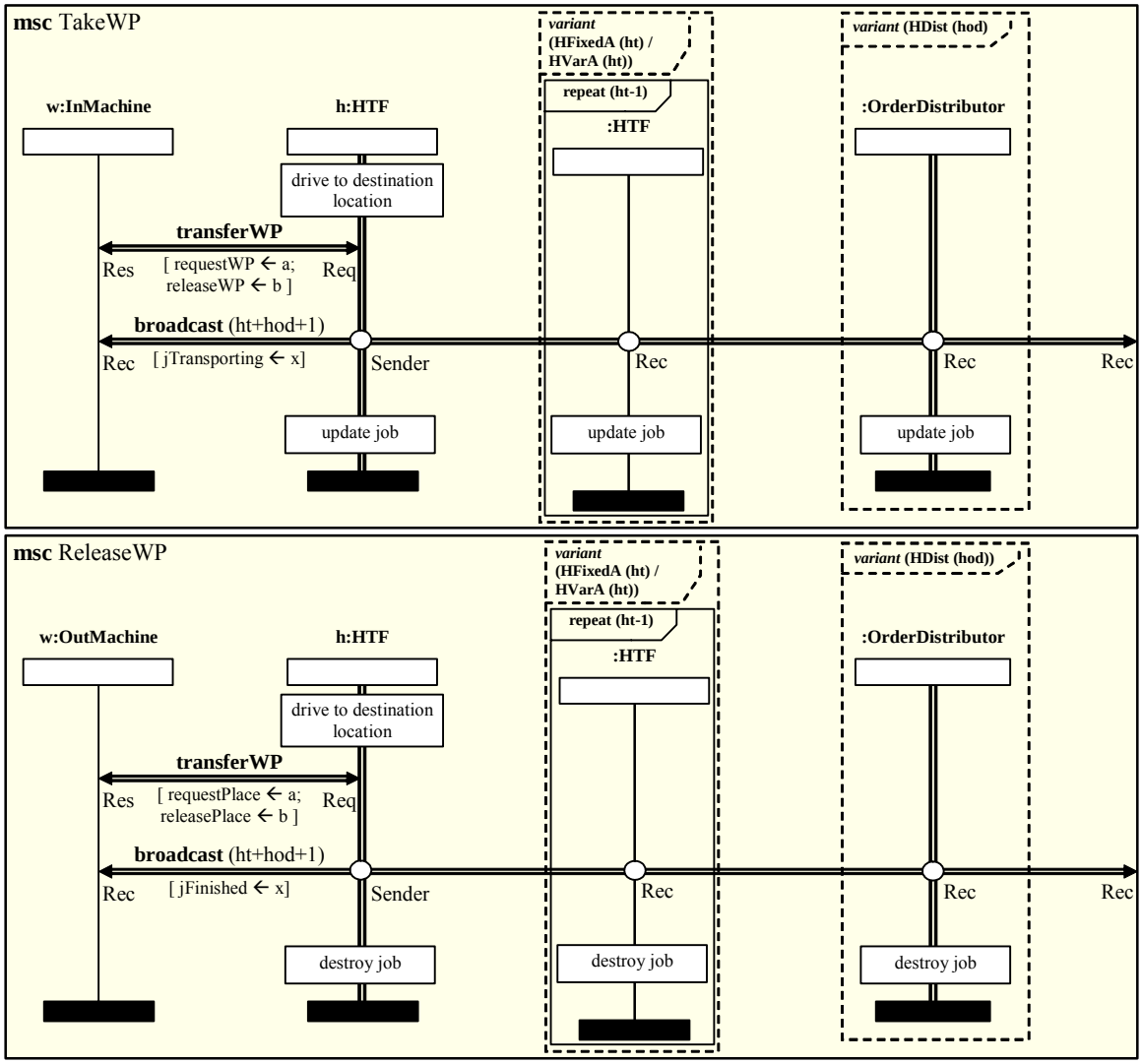


Figure 46 MSC TakeWP (Family), MSC ReleaseWP (Family)

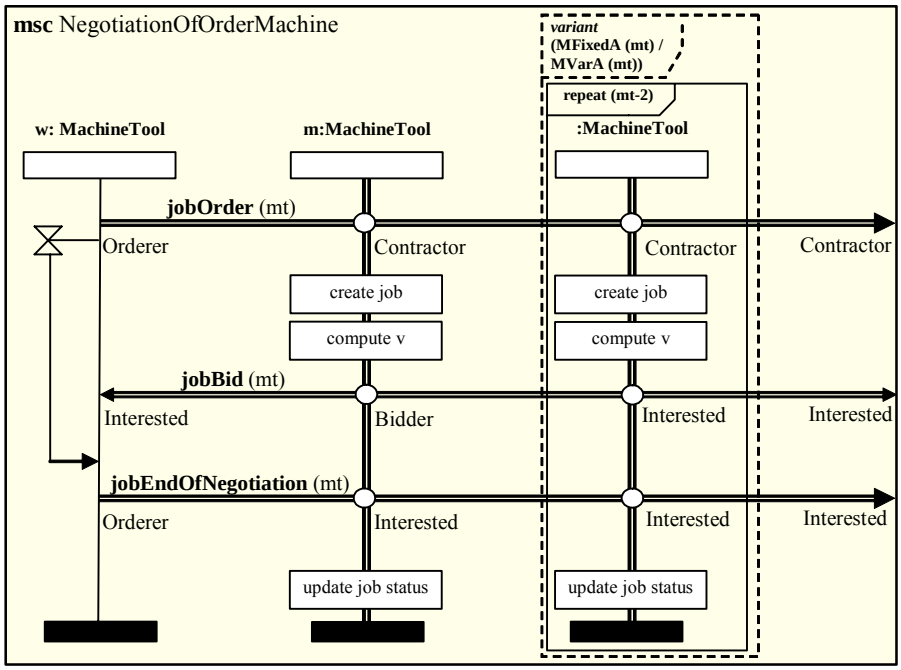


Figure 47 MSC NegotiationOfOrderMachine

In the following, the connectors *transferWP* and *jobAssigned* used in the MSCs above are described. We do not repeat connectors that were introduced in earlier sections and are unchanged in the family model. These are the *broadcast* (Figure 14), *jobOrder*, *jobBid*, and *jobEndOfNegotiation* (Figure 39).

The connector in Figure 48 is an example where connectors help in introducing system family variation. It contains an optional exchange of the messages *getSequence* and *sequence* that determine the sequence of machine tools for a work piece. This part is guarded by a variant occurrence of *VarProcSeq*.

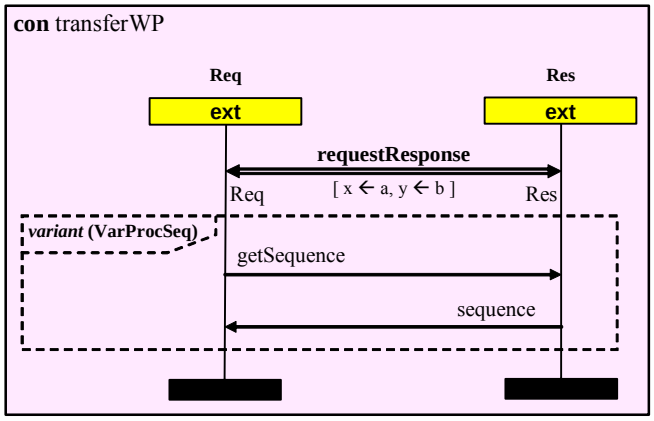


Figure 48 CON transferWP

Finally, the connector *jobAssigned* in Figure 49 is needed for the MSCs that describe the assignments of orders for HTF or machine tools. It simply returns acknowledges for the orderer and the other interested component.

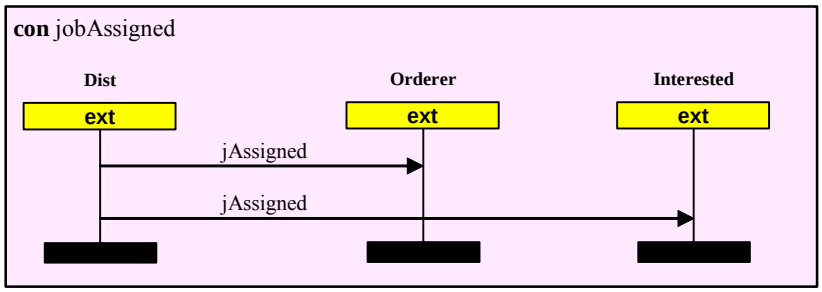


Figure 49 CON jobAssigned.

8 Related Work

KobrA [19] and its predecessor PuLSE [18] are development processes specifically focused on system family development. They introduced the stereotype <<variant>> to denote variants in UML models but have no defined transition from features to variation points and variants in sequence diagrams.

We borrowed the feature analysis approach from feature-oriented domain analysis (FODA) [10] and its successor feature-oriented reuse method (FORM) [22]. The latter gives some guidelines on how to map the features from a feature model to architecture models.

The approach in [27] uses the special symbol <<V>> to represent variability in UML class diagrams that serve as domain models. It allows the expression of alternatives only in accompanying text.

The most similar approach to ours is [16] that also uses class diagrams with dependencies expressed with OCL. The extension to a UML profile for product lines is proposed in [17] that introduces several stereotypes for sequence diagrams. The vertical dimension however can only be marked optional in that approach.

As an alternative to the further formalisations of a feature model described in the appendix, a formalisation using description logic is presented in [25]. It differs strongly because it was built with the aim to find feature interaction, not for the modelling of system families.

9 Conclusions

MSC Connectors. An observation we make in this case study is that connectors are indeed a means to make MSCs more concise. It is important hereby to choose the right names. An MSC with just a few connectors with speaking, intuitive names can help the reader to grasp the communication pattern of components more quickly and more easily than using only standard messages in the MSCs. If the names of the connectors were not chosen appropriately, it would however confuse the reader more than it would help.

Furthermore there is also additional information in the MSC compared to standard MSCs. The introduction of explicit environmental instances can increase the grasp of the reader for the MSC. Instead of the anonymous environment, it is made explicit which roles from the environment interact with the instances in the MSC.

Finally another interesting aspect concerning MSC connectors to notice is that the use of connectors enforces the designer to consistency. If the forward connector with the *IOSystem* is used in the MSCs to communicate with the environment, it is ensured that the *IOSystem* can not be forgotten. The designer can concentrate at one task at a time.

Repeat Operator. We introduced the additional inline operator *repeat* in the MSC language that proved to be useful especially for describing system families. The *repeat* construct specifies the cardinality of instances and can therefore be parameterised to describe variations there.

Variation Point and Variant. Moreover two metaoperators were introduced, *variant* and *vp*, that allow the description of a whole system family with MSCs. The metaoperator variant manipulates MSCs in their horizontal and vertical dimension in that it adds, modifies, or removes interactions and instance axes. The metaoperator vp is used in HMSCs and defines high-level decision nodes which are resolved at the time of configuration. These metaoperators have been proved to suffice for the purposes of the case study. Moreover, due to the fact that a MSC is exactly the union of the above mentioned tree dimensions (i.e., vertical and horizontal dimensions plus abstraction), we think that those metaoperators are indeed enough for variability representation in MSCs.

Current and Future Work. The meaning of the extensions introduced was given in informal terms, a formal semantics must be defined. One possibility is to consider a kind of precompiler, which takes an extended (H)MSC and a configuration, and delivers a (H)MSC without occurrences of the metaoperators. The configuration could be conceived as a subtree of the feature model without alternatives and with optional features possibly removed.

Regarding the whole process, from the analysis of commonalities and variabilities of a system family to be through feature and use case modelling to the specification of MSCs, we feel it could conceptually be carried out as follows. Initially, one is more or less precisely aware of the set of all desired system traces, in any of the members of the system family. Those traces are associated to at least one feature, and the set of all traces can be divided (not partitioned) into subsets associated to the features. This reasoning also helps to produce a feature model.

Using the knowledge of which features are optional and which ones are alternative to other ones, the corresponding trace sets are declared variants. In doing so, alternative features are organised around a single variation point, whereas optional features belong each to a variation point of their own.

The (still ideal) traces are then used for two purposes: on the one hand for a use case modelling including variation points, and on the other for a raw specification of MSCs with occurrences of variation points and of variants.

In a next step we will formalise the semantics of the extensions introduced and elaborate on the development process.

Acknowledgements. We are grateful to Manfred Broy who commented on a draft version. Furthermore we want to thank Ingolf Krüger, Wolfgang Prenninger, and Robert Sandner for the preceding work on the holonic flow of material.

References

1. I. Krüger, W. Prenninger, R. Sandner: *Development of an Autonomous Transport System using UML-RT*. Technical Report TUM-I0215, Institut für Informatik, Technische Universität München, 2002.
2. A. Braatz, A. Ritter: *Spezifikation des verteilten Steuerungskonzeptes für den holonischen Materialfluss in einem werkstattorientierten Fertigungssystem auf der Basis autonomer, freifahrender Transportsysteme*. Referenzfallstudie, Fraunhofer-Institut für Produktionstechnik und Automatisierung IPA, 2002.
   Available at: http://tfs.cs.tu-berlin.de/projekte/indspec/SPP/RefPAv2.ps
3. J. Grabowski, P. Graubmann, E. Rudolph: *HyperMSCs with Connectors for Advanced Visual System Modelling and Testing*. In: SDL 2001: Meeting UML, Proceedings of the 10th International SDL Forum, Copenhagen, Denmark, June 2001 (Rick and Jeanne Reed, editors), Springer LNCS 2078, 2001.
4. P. Graubmann: *Describing Interactions between MSC Components - The MSC Connectors*. In: Rick Reed (Guest Editor): Computer Networks, Special Edition: ITU-T System Design Languages (SDL), Volume 42, Issue 3, 21 June 2003, pp 323-342, Elsevier Science B.V.

5. P. Graubmann: *MSC Connectors - The Chamber of Secrets*. In: Rick and Jeanne Reed (eds.): SLD 2003: System Design. Proceedings of the 11th International SDL Forum, Stuttgart, Germany, July 2003. LNCS 2708, Springer 2003.
6. P. Graubmann, E. Rudolph, J. Grabowski: Component Interface Descriptions using HyperMSCs and MSC Connectors. IEEE Visual Languages and Formal Methods, Stresa, Italy, September 5-7, 2001.
7. P. Graubmann, E. Rudolph: *MSC Connectors - The Philosopher's Stone*. In: Edel Sherratt (ed.): Telecommunications and beyond: The Broader Applicability of SDL and MSC. Proceedings of the 3rd SAM Workshop 2002, Aberystwyth, June 2002, Revised Papers, LNCS 2599, Springer 2003..
8. ITU-T Rec. Z.120 (MSC-2000): Message Sequence Chart (MSC). (O. Haugen, editor), Geneva, 1999.
9. B. Selic, J. Rumbaugh: *Using UML for modeling complex real-time systems*. Whitepaper. ObjectTime Limited, 1998.
10. K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, and A.S. Peterson: *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.
11. R.L. Constable: *The Structure of Nuprl's Theory*. In: Logic of Computation. M.Broy and H. Schwichtenberg (editors). Springer, 1997.
12. D. Hirsch, S. Uchitel, D. Yankelevich: *Towards a Periodic Table of Connectors*. Proceedings of COORDINATION'99, Third Int. Conference on Coordination Models and Languages, Springer LNCS 1594, 1999.
13. S. Bühne, G. Halmans, K. Pohl: *Modelling Dependencies between Variation Points in Use Case Diagrams*. Proceedings of the 9th International Workshop on Requirements Engineering – Foundations for Software Quality (REFSQ'03), 2003.
14. G. Halmans, K. Pohl: *Communicating the Variability of a Software-Product Family to Customers*. Software and System Modeling 2(1): 15-36, 2003.
15. P. Clements, L. Northrop: *Software Product Lines*. Practices and Patterns. Addison-Wesley, 2002.
16. T. Ziadi, L. Hélouët, J.-M. Jézéquel: *Product Line Derivation with UML*. In: Groningen Workshop on Software Variability Managment, 2003.
17. T. Ziadi, L. Hélouët, J.-M. Jézéquel: *Towards a UML Profile for Software Product Lines*. In: Software Product-Family Engineering: 5th International Workshop (PFE'03), LNCS 3014, Springer, 2003.
18. M. Anastasopoulos, J. Bayer, O. Flege, and C. Gacek: *A Process for Product Line Architecture Creation and Evaluation. PuLSE-DSSA – Version 2.0*. IESE-Report 038.00/E, Fraunhofer IESE, 2000.
19. C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Peach, J. Wust, J. Zettel: *Component-Based Product Line Engineering with UML*. Addison-Wesley, 2001.
20. I. Jacobson, M. Griss, P. Jonsson: *Software Reuse: Architecture, Process, and Organization for Business Success*. Addison-Wesley, 1997.
21. D. Muthig, C. Atkinson: *Model-Driven Product Line Architectures*. In: 2nd International Software Product Line Conference, LNCS 2379, Springer 2002.
22. K.C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, M. Huh: *FORM: A feature-oriented reuse method with domain-specific reference architectures*. Annals of Software Engineering 5, 1998.
23. Object Management Group: *UML 2.0 Superstructure Final Adopted Specification*, August 2003. OMG Document ptc/03-08-02.
24. I. Krüger, W. Prenninger, R. Sandner, M. Broy: *Development of Hierarchical Broadcasting Software Architectures using UML 2.0*. In: Integration of Software Specification Techniques for Applications in Engineering, LNCS 3147, Springer 2004.
25. R. van der Straeten, J. Brichau: *Features and Feature Interaction in Software Engineering using Logic*. In: ECOOP 2001 Workshop on Feature Interactions in Composed Systems, 2001.
26. M.V. Cengarle and A. Knapp: *UML 2.0 Interactions: Semantics and Refinement*. In: Workshop on Critical Systems Development with UML (CSDUML'04), 2004. To appear.
27. D. McComas, S. Leake, M. Stark, M Morisio, G. Travassos, and M. White: *Addressing Variability in a Guidance, Navigation, and Control Flight Software Product Line*. In: Product Line Architecture Workshop at Software Product Line Conference (SPLC1), 2000.

Appendix: Original MSCs

We give two examples of the original MSCs from [1] to demonstrate the improvement in terms of conciseness. The two MSCs are the most obvious ones to illustrate this because they include the most communication. They correspond to the MSCs in Figure 23 and Figure 24.

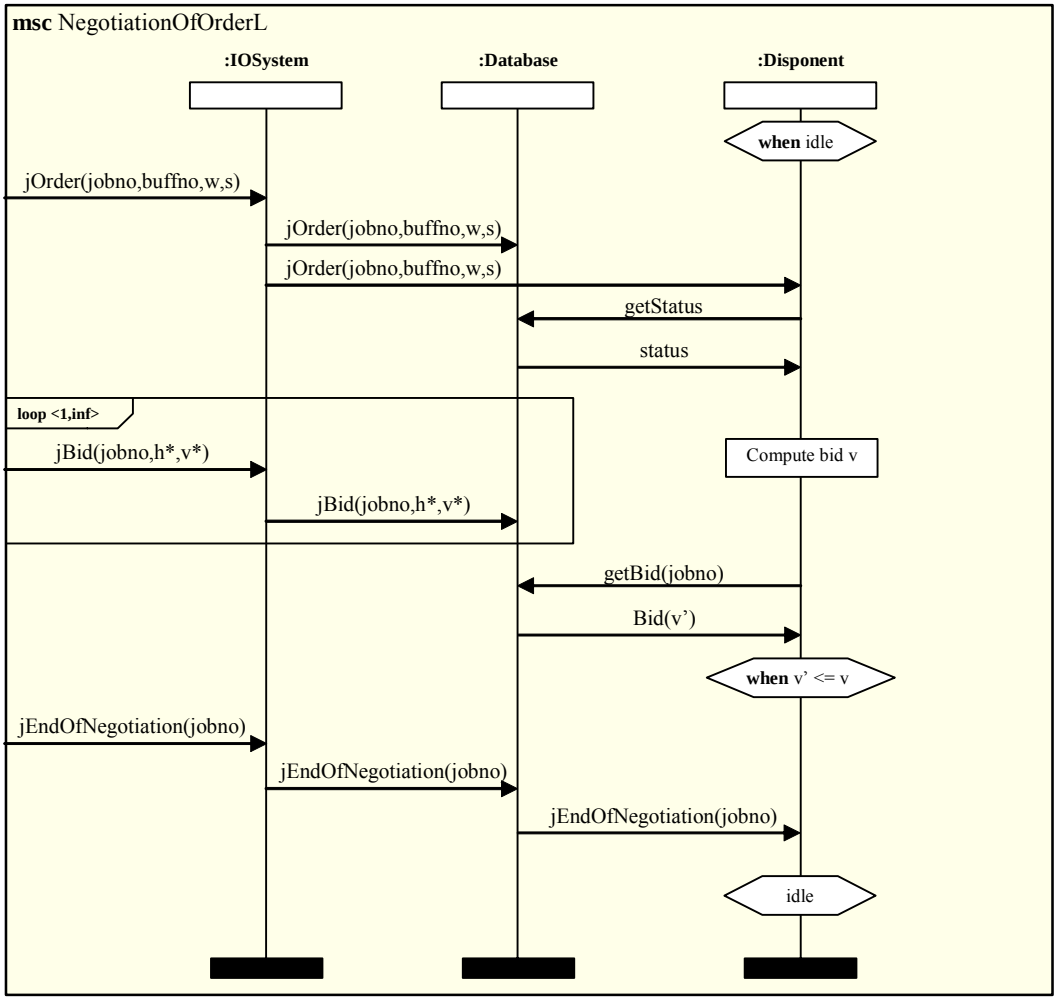


Figure 50 MSC HTSNegotiationOfOrderL (original)

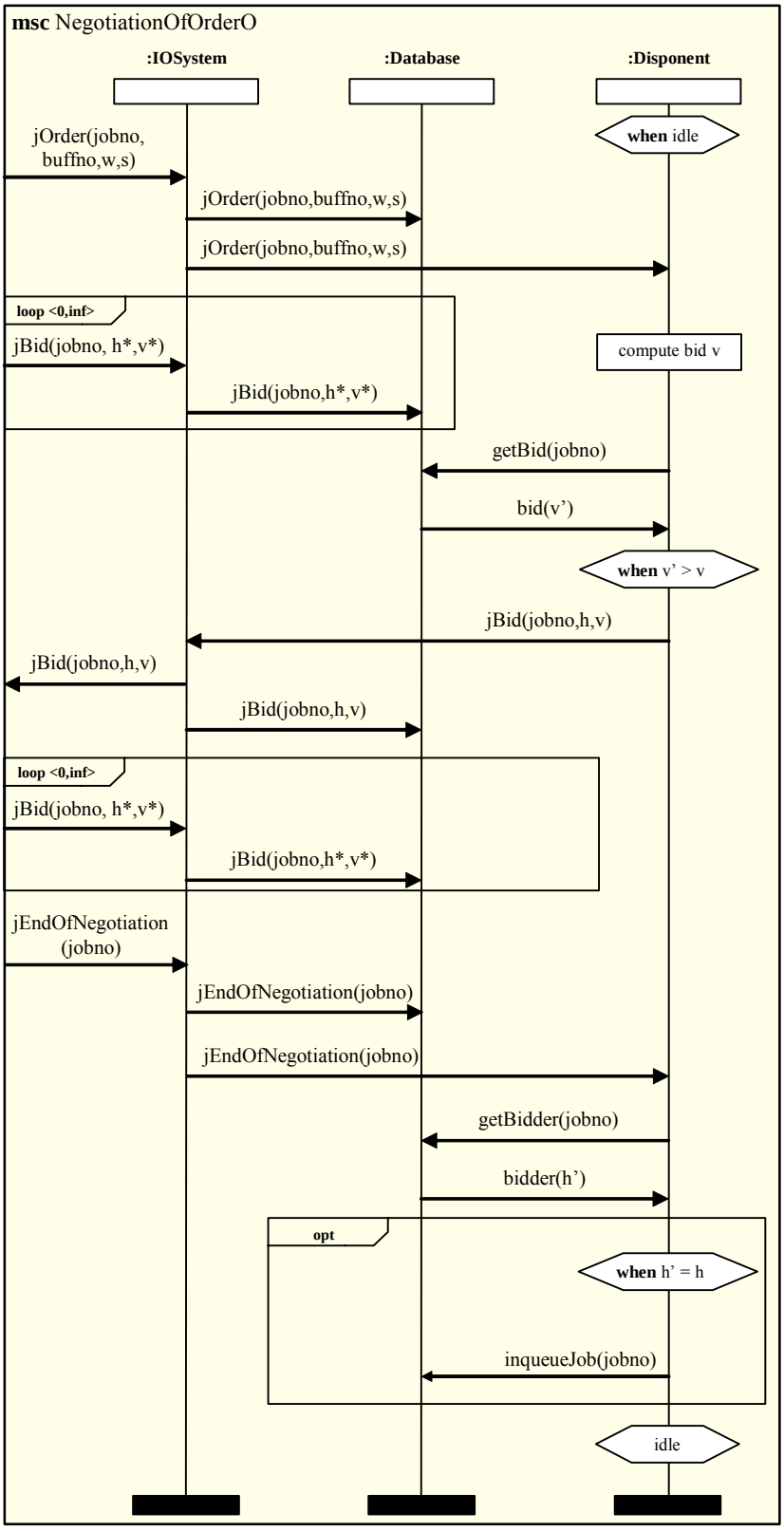


Figure 51 MSC NegotiationOfOrderO (original)

Appendix: Alternative Formalisations of a Feature Model

In the following alternative formalisations of the feature model are provided. This is intended to help in choosing only a valid configuration and also provides support in the derivation of parameters for the MSCs. We developed to alternatives formalisations so far that are both presented in the following.

Logic. The first approach describes the feature dependencies as logical formulas. The primitive features are described with the values *true* and *false*. We use $\oplus$ as notation for the *exclusive disjunction*. For the feature model from the case study (Figure 41) this looks like the following:

$HNeg, HDist, HVarA, HFixedA, MVarA, MFixedA, MNeg, MDist, InStorage, OutStorage, VarProcSequence \in \{true, false\}$

$VarRoute = (HFixedA \oplus HVarA) \wedge (HDist \oplus HNeg)$

$HTF = FixedRoute \oplus VarRoute$

$MLoadBal = MDist \oplus MNeg$

$MachineTools = (Simple \oplus MLoadBal) \wedge (MVarA \oplus MFixedA)$

$HTS = (HTF \wedge MachineTools \wedge InStorage \wedge OutStorage \wedge \neg VarProcSeq) \vee (HTF \wedge MachineTools \wedge InStorage \wedge OutStorage \wedge VarProcSeq)$

These Boolean values of the features can then be used to determine the values of MSC parameters. The parameter *mt* is the number of machine tools, *ht* the number of HTF, and *mod* and *hod* the number of order distributor for machine tools and HTF, respectively. For this, we interpret the MSC parameters as partial functions that map Boolean values to natural numbers or a subset of it. The undefined parts of the functions represent configurations that are not possible or not allowed.

$mod : Bool \times Bool \rightarrow \{0,1\}$

$mod : (true,true)$ undef.

$mod : (true,false) \mapsto 0$

$mod : (false,true) \mapsto 1$

$mod : (false,false)$ undef.

$hod : Bool \times Bool \rightarrow \{0,1\}$

$hod : (true,true)$ undef.

$hod : (true,false) \mapsto 0$

$hod : (false,true) \mapsto 1$

$hod : (false,false)$ undef.

$mt : Bool \times Bool \rightarrow \mathbb{N}$

$mt : (true,true)$ undef.

$mt : (true,false) \mapsto 3$

$mt : (false,true) \mapsto m \in \{2,3,5,7,9\}$

$mt : (false,false)$ undef.

$ht : Bool \times Bool \rightarrow \mathbb{N}$

$ht : (true,true)$ undef.

$ht : (false,true) \mapsto h \in \{2,3,4,6,8\}$

$ht : (true,false) \mapsto 3$

$ht : (false,false)$ undef.

Dependent Types. Another possibility would be to use dependent types [11]. We interpret the features as types and types of types. The symbol ε denotes no feature. LL stands for lowest level and returns the types for the lowest level in the feature model. Again the feature model of the case study is shown as an example.

$$
\begin{aligned}
LL(VarRoute) &= \{HVarA,HFixedA\} \times \{HDist,HNeg\} \\
LL(FixedRoute) &= \{\varepsilon\} \\
HTF &= r \in \{FixedRoute,VarRoute\} \times LL(r) \\
LL(MLoadBal) &= \{MDist,MNeg\} \\
LL(Simple) &= \{\varepsilon\} \\
MachineTools &= \{MVarA,MFixedA\} \times m \in \{Simple,MLoadBal\} \times LL(m) \\
HTS &= HTF \times MachineTools \times \{InStorage\} \times \{OutStorage\} \times \\
&\quad \{VarProcSequence,\varepsilon\}
\end{aligned}
$$

This can as well be broken down to the parameters for the MSCs. This time we interpret the parameters as total functions that map the *Holonic Transport System (HTS)* type to the natural numbers or a subtype of it. Then we give the definition of the functions using assignments of HTS. However, we only give the assignments needed for that definition and use the underscore symbol (_) for any assignment.

$mod : HTS \rightarrow \{0,1\}$

$mod : (\_,(\_,\_,MDist),\_,\_,\_) \mapsto 1$

$mod : (\_,(\_,\_,MNeg),\_,\_,\_) \mapsto 0$

$hod : HTS \rightarrow \{0,1\}$

$hod : ((\_,(\_,HDist)),\_,\_,\_,\_) \mapsto 1$

$hod : ((\_,(\_,HNeg)),\_,\_,\_,\_) \mapsto 0$

$mt : HTS \rightarrow \mathbb{N}$

$mt : (\_,(MFixedA,\_,\_),\_,\_,\_) \mapsto 3$

$mt : (\_,(MVarA,\_,\_),\_,\_,\_) \mapsto m \in \{2,5,7,9\}$

$ht : HTS \rightarrow \mathbb{N}$

$ht : ((\_,(HFixedA,\_)),\_,\_,\_,\_) \mapsto 3$

$ht : ((\_,(HVarA,\_)),\_,\_,\_,\_) \quad \mapsto h \in \{2,3,4,6,8,10\}$