A Comparison of Agent-Based Coordination Architecture Variants for Automotive Product Change Management

Janek Bender, Stefan Kehl(✉), and Jörg P. Müller

Department of Informatics, Clausthal University of Technology,
Clausthal-zellerfeld, Germany
{janek.bender,stefan.kehl,joerg.mueller}@tu-clausthal.de

Abstract. Automotive companies tend to apply modular approaches in their product development processes in order to save costs and meet increasingly diversified customer demands. In largely decentralized environments with cross-branded development projects over multiple departments in different sites this modular approach leads to very complex and large data structures. Maintaining consistency and transparency, as well as coordinating information flows in such an environment is a major task which is often accomplished manually. Based on a real world case study, this paper analyzes a key development process: the connection of geometric (geometries) and logistical data (parts). During this time consuming process information carriers (geometries and parts) with independent lifecycles that are maintained by different stakeholders (designer and purchaser) of different departments (and in this scenario even within multiple brands) are linked as these carriers themselves are mutually dependent. This paper then proceeds to model five agent-based architecture variants to support this process. In addition, an algorithm to map geometric and logistical data which aims to relieve the actors involved (regarding the organizational overhead) is outlined.The paper concludes with a comparison of the different agent architecture variants and emphasizes the most promising variants to partly automate the connection of geometric and logistical data.

1 Introduction

Rising competitive pressure in the automotive industry forces manufacturers into extensive cost saving measures. International markets constantly demand more variety in shorter time periods. While in the 1990 s the product portfolio of most car manufactures covered about eight different models, today's industry offers a steadily increasing amount of different products (e.g. the portfolio of the Audi Group covers about 50 different products) in various configurations in order to meet as many customer demands as possible [13,14]. This way of manufacturing cars specifically suited to customer needs in mass production is known as *mass customization* [11]. In order to achieve customized products at a cost level near mass production some manufacturers pursue approaches to increase the degree of

© Springer International Publishing Switzerland 2015
J.P. Müller et al. (Eds.): MATES 2015, LNAI 9433, pp. 249–267, 2015.
DOI: 10.1007/978-3-319-27343-3_14

commonality. One approach to achieve this goal is the modularization of products so that components are combined into interchangeable modules [2,5]. Ideally, a complete product can be configured using the modular design principle. However, this approach is not entirely applicable for complex products (such as vehicles), because of the large amount of connections between those modules which have to be considered [10]. Changes of such components may have a major impact on other components [3]. Furthermore, a modularization beyond an optimum range would lead to higher product costs [1].

In addition to the need for mass customization, international corporations often operate in a decentralized fashion, with several brands, different business units, and multiple departments and teams, all in geographically dispersed locations. Thus, efficient communication and coordination between sites is mandatory [12].

This work is based on three real world use cases derived from case studies at a major automotive OEM. Based on these use cases, Sect. 2 highlights certain problems in the current development process of connecting geometric and logistical data sets from different entities in a decentralized environment. Following the Virtual Product Model (VPM) proposed by Kehl et al. in [6] five architecture variants for an agent-based approach to automate this key development process are modeled in Sect. 3. Section 4 discusses the pros and cons of these architecture variants as well as their applicability to fulfill the requirements defined in Sect. 3. Section 5 compares the architecture variants; Sect. 6 concludes this paper and gives an outlook on future work.

2 Background

In previous work [6], we performed an analysis of challenges in managing development processes of complex products in the automotive industry. We proposed the concept of a *Virtual Product Model (VPM)* to manage such complex and cross-branded development processes. In practice a product data is organized in different hierarchical and static structures [4]. Each of these manually maintained structures represents a domain specific view on the product (e.g. a Bill of Material (BOM) for purchasers or an Engineering BOM for designers) which are linked to or transformed into one another [16]. In contrast, the VPM described in [6] offers a more flexible and component-based view on a product, because each domain specific structure is considered as a view on the overall product which is built dynamically, based on the information it carries and the connections between them (VPM-C). The concept of a VPM aims to fulfill three crucial requirements in a product development process:

Reusability. One VPM-C can be assigned to multiple products, yet there is an element that holds context-specific information.

Patency. From the early stages to the end of a product's development the information flow should not be interrupted. Data has to be kept consistent between all involved entities.

Transparency. It is mandatory to establish traceability of changes on components and connections between them throughout the entire development process.

Each *Virtual Component (VPM-C).* can be divided into four elements:

Part. A part represents a logistical data set describing a real world vehicle part. It contains information such as the supplier, color, count or material. In practice parts are organized in a BOM.

Geometry. A geometry is a 3D-CAD file which holds data such as the size, shape, or position of a part. Furthermore, a part may be realized by a geometry.

Feature. A feature may be a technical description or a certain functionality of the component.

Process. The production process(es) the component is assigned to. This element might contain information about different production sites.

This paper builds on the VPM concept, yet some of its elements are rendered out of scope for the problem at hand. The elements considered in this paper are *parts* and *geometries*. Also, contrary to the original VPM proposal this work limits itself (for the sake of clarity) to only two different roles involved in the mentioned use cases:

Engineer. The engineer constructs and combines vehicle parts by using CAD software. His work is based on geometries (3D-CAD files), containing information like size, shape, or position of a component.

Purchaser. The counterpart in logistics is the purchaser whose main responsibilities are the procurement of parts and materials and managing supply chains. He views product models from a logistic perspective and works based on parts (logistical data sets) containing information such as the supplier, color, count, material, among others.

2.1 Use Cases

As part of this work, three use cases, totaling six distinct workflows, have been derived from case studies from the automotive industry. These use cases show the underlying problem this paper aims to tackle and are used to conceptualize possible solutions. An overview about the three major use cases is given below.

Use Case 1: New Construction of a VPM-C. A new component is developed based on a given specification. First, the engineer designs the new geometry with his CAD software. He will then proceed to position the newly constructed geometry within a 3D environment, relative to other previously constructed geometries. Lastly, the engineer assigns the geometry to one or multiple vehicle contexts.

Use Case 2: Reuse of a VPM-C. An existing geometry is assigned to a new context. It may have to be re-positioned within the 3D environment in order to fit the new context.

Use Case 3: Further Development of a VPM-C. Contrary to use case 1, an existing geometry is developed further within its context and may have to be re-positioned within the 3D environment.

In each of the above use cases, geometric data must be connected with logistical data. Steps to achieve this connection might vary depending on the use case.

2.2 An Exemplary Workflow of a New Construction

To illustrate the problem and the solution statement a small example shall be given at this point. Assuming that a new vehicle generation should receive a door hinge constructed from scratch, the engineer would first have to design a 3D geometry using a CAD software. He would then position the new geometry in relation to close-by components, like the door frame, wiring, or the window lift, making sure the new geometry does not cut other geometries within the 3D space. The engineer would thereafter proceed to assign the new geometry to the respective vehicle context of the new vehicle generation. After the engineering is completed the data is sent to the purchaser who requests a new distinct part number. Using the part number, the purchaser will create a new branch within the Bill of Material (BOM) data structure. Finally, the geometric data and the logistical data are connected within the data structure. In the following, the establishment of this connection is referred to as *Part-Geometry-Mapping*.

The first problem in the previously described process is the decentralized generation and administration of data in multiple brands and departments. This data must be kept consistent throughout all entities involved in the vehicle construction process. Furthermore, there is little transparency on which parts belong to which geometries and vice versa. The same geometry might realize multiple parts in different vehicles and different markets. Especially when a certain part in one vehicle has shown so far unforseen problems after the product launch, it becomes crucial to quickly identify all other vehicles using this very part. Right now, it is only possible to achieve the needed transparency by devoting a lot of manual efforts to the cause. Also, by connecting and maintaining logistical and geometric data within the same data structure the used BOMs exceed a healthy size and develop redundancies at some point. Both engineers and purchasers find themselves confronted with data irrelevant to their own tasks. For instance, engineers need geometries to do their work while purchasers rely on logistical data, yet it is stored in the very same structure.

3 An Agent-Based Approach to Part-Geometry-Mapping

The problems shown in the example above arise from largely decentralized processes. Information is created and maintained in different brands with multiple departments and teams at geographically dispersed locations. Managing changes on VPM-Cs and maintaining consistency throughout all involved parties is a major task. However, there is no single entity overseeing the whole development and all its changes. Furthermore (as described in Sect. 2.2), in practice, the Part-Geometry-Mapping is a manual process. Due to the decentralized problem and the distributed stakeholders (purchasers and designers from different brands), an agent-based solution is proposed in this paper, as mentioned in our previous work [6]. The basic idea builds on so called *active components* representing certain entities in the process. An *active component* is controlled by an agent which is considered “a software system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its

delegated objectives" [8]. Furthermore, Wagner [15] has already deployed an agent-based approach into a similar environment with success. In order to investigate whether an agent-based approach is feasible and offers real advantages over the manual mapping of parts and geometries, five possible agent architecture variants have been developed. These architecture variants model structural and behavioral aspects of *active components* and may be realized through real software agents in future research. This section is structured as follows: Sect. 3.1 describes the underlying problem in detail and in the Sects. 3.2 to 3.6 the five architecture variants are illustrated.

3.1 Part-Geometry-Mapping

Before discussing the different agent architecture variants in detail, the understanding of the elements and their relations should be clarified. Figure 1 gives an overview.

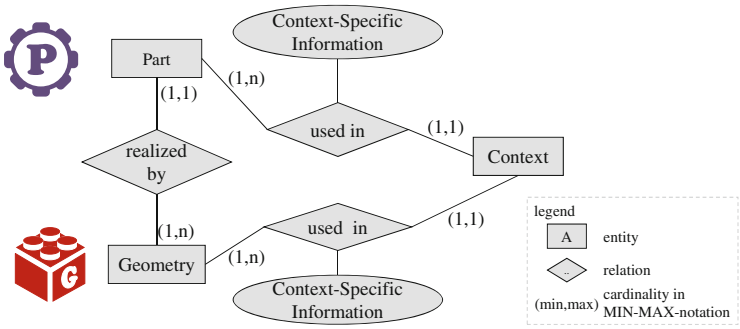

Fig. 1. Elements of the mapping process

As already mentioned *parts* are realized by *geometries*. A part or geometry can be *used* in several *contexts* (platforms or vehicles). The *usage* of a part or geometry comes with certain *context-specific information*, for example the position of a geometry in a specific context.

Thus the main requirements are:

- Contribute to establishing transparency and traceability of changes in order to be able to quickly answer the two questions:
  1. Which revisions of a given component are used in which platforms, or vehicles?
  2. Which components in which revisions are used in a given context?
- By (partially) automating the part-geometry-mapping in order to get a hold of the complex and decentralized information generation and maintenance. Make key information available where it is needed and keep it consistent throughout all involved entities.

To simplify the modeling of possible solutions both geometries and parts are abstracted into the generic term *component*. The necessary properties of the component are described below:

- Parts and geometries are elements of a VPM-C as they are both offering different views on the same real world counterpart.
- Over the iterative course of development a component forms *multiple revisions*. Each change on the component causes a new revision which can be *assigned to a context* (vehicle or platform). As an example, revision #1 might be a *development build* only used as placeholder or prototype during the development. Revision #2, a fully developed version of the component, goes into production and is thereby part of the manufactured vehicle. Now the vehicle series receives a facelift and the component needs to be adapted leading to revision #3.
- Since multiple revisions of a component may be assigned to the same context (in the above case revisions #1 and #2), each revision has a *validity* property and is only used within a specified start and end date.
- A component can offer a certain *function*. For instance, a rim allows someone to attach tires to a car which then are part of the *feature* "driving".
- Based on an idea by Wagner in [15], the participating roles can define *rules* for components. These are either *configuration rules* describing certain restrictions of the component itself or *connection rules* which regulate the relations to other components.
- A component belongs to a *pre-defined category*.
- A component is from a specified *side*. The side property allows for the division of components and identification of possible counterparts of a component from another side. In extension, the category property allows someone to draw connections between two components from different sides. Thus, both properties are mandatory and play a vital role in the *mapping process*.

For example, a new rim is developed from scratch. The component of the geometry is on side 0 and the matching component of the part information on side 1. However, both belong to the same category "Wheels". There is no mapping between these two thus far, but this information alone is sufficient to specify that these two *components* are *possible counterparts* for each other. Figure 2 illustrates the *component* term.

In the following, the five proposed architecture variants for an agent-based approach are discussed.

3.2 Architecture Variant I - Dedicated Element Agents

This architecture variant covers the most basic and intuitive approach in modeling a MAS capable of autonomously conducting part-geometry-mappings within given contexts, or even more abstract: component-mappings. Figure 3 shows an overview of proposed agent classes, marked by boxes. "Dedicated Element Agents" refers to the fact that each element involved in the process (namely

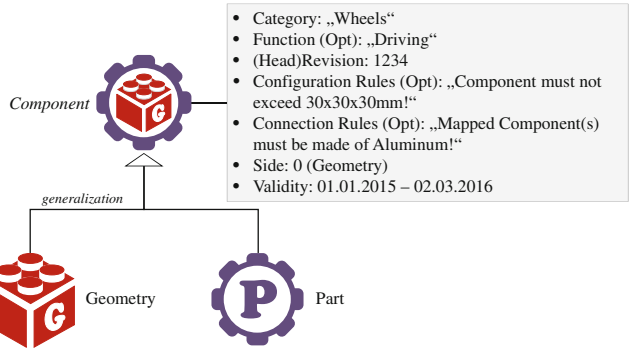

Fig. 2. The component

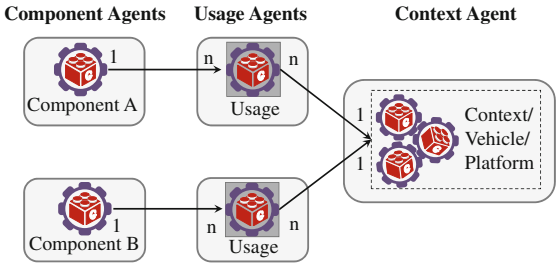

Fig. 3. Architecture Variant I - Dedicated Element Agents

geometries, parts, their respective usages / revisions, and contexts) is represented by a single agent with specified tasks and goals.

Three basic agent classes are explained below. It should be noted that both geometries and parts are considered components and thus be represented by a *Component Agent*.

Component Agent. The Component Agent represents a component within the system. It initializes once a component is first created and destroys when the underlying component is disabled. Its main purpose is to maintain static information about the component such as *category* or *ID*. Furthermore the Component Agent knows all revisions of a component and retains data about its different usages. A Component Agent will remain passive most of the time, monitoring the engineers' actions within the CAD-Program and tracking changes on the component.

Figure 4 shows a behavioral view of the Component Agent. Upon initialization a new Component Agent receives parameterized information regarding *category*, *component*, *function*, *rules*, and *side*. It will then idle in its main loop until either the underlying component is disabled, which results in the destruction of the Component Agent, or it receives new information. The latter happens if another agent sends new information to the Component Agent or the engineer

performs changes on the underlying component. If a current revision is assigned to a *new context*, the Component Agent will trigger the creation of a *Usage Agent* with the parameters *category*, *context*, *function*, *revision*, *rules*, *side*, and *validity*. Once the Component Agent registers a changed position of the underlying component, it will send the new position to the respective Usage Agent.

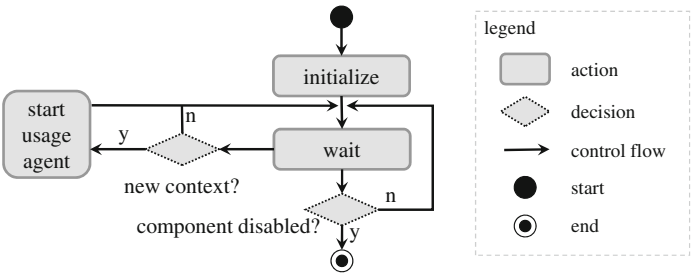

Fig. 4. Component agent behavioral view

Usage Agent. The Usage Agent represents the revision of a component in a specific context. It initializes once a revision of a component has been assigned to a new context. The initialization is triggered by a Component Agent. Passed parameters are *category*, *context*, *function*, *revision*, *rules*, *side*, and *validity*. A Usage Agent's main task is to map his own component on another component within the context. It therefore actively looks for mappable components of the *same category* but *different side* and contacts them in order to find one or more matches. A Usage Agent's component can have *multiple mappings* and the Usage Agent is only active as long as its component has zero mappings. After a Usage Agent has been successfully mapped it will still react to mapping requests of other agents but discontinue to actively look for mappings. If the component of the respective Component Agent is being disabled or the given validity of the revision runs out, the Usage Agent will be destroyed.

Figure 5 shows a behavioral view of the Usage Agent. The mapping process itself is described in detail in Sect. 4. Upon initialization the Usage Agent receives the parameters *category*, *context*, *function*, *revision*, *rules*, *side*, and *validity*. As part of the initialization the Usage Agent will contact the Context Agent to register itself. The Usage Agent remains in its main idle loop until it receives new information from another agent or becomes active by looking for mappable components or responding to incoming mapping requests from other Usage Agents. If a Usage Agent does not have any mappings yet, its top priority is to find a matching component from the same category but another side. Therefore, it contacts the respective Context Agent in order to request a list of mappable components based on category and side. It will then iterate through the returned list and contact each component's Usage Agent to take on mapping negotiations. If a negotiation ends with a negative result the respective component is removed from the list and the Usage Agent will approach the next in line. If the list has been

emptied and a successful mapping has not been found, the Usage Agent returns to idle state and will request a new list in the next cycle. A positive negotiation ending results in a successful preliminary mapping which is then communicated to both the respective Context and Component Agents. The Usage Agent will then return to the idle state and become reactive to incoming mapping requests and new information.

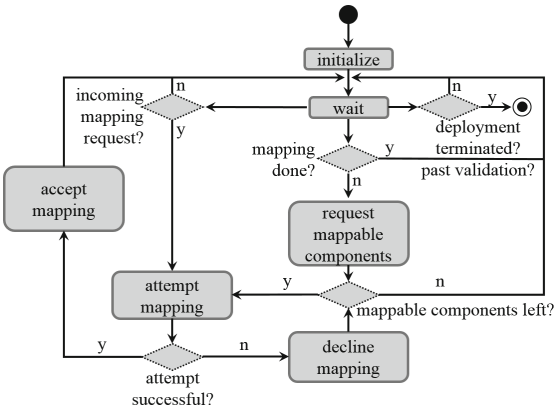

Fig. 5. Usage agent behavioral view

Context Agent. The Context Agent overlooks a specific context. It holds data concerning active Usage Agents within the context, assists in the mapping process and knows all of the successful mappings. Once a new context is being created the Context Agent initializes with the *context* parameter. It then remains in the idle state until it receives new information or requests from other agents.

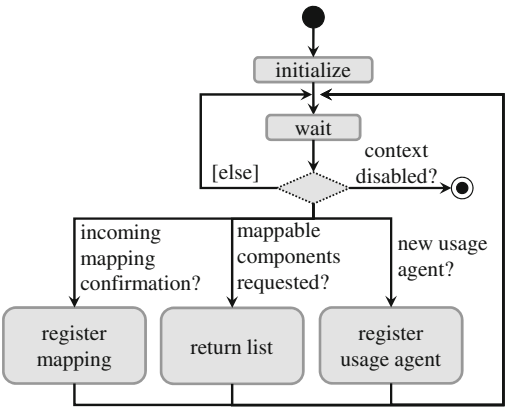

Fig. 6. Context agent behavioral view

The Context Agent's main tasks are to register new Usage Agents and successful mappings and to compile and return lists of mappable components to Usage Agents. With the termination of the context, the Context Agent is destroyed.

Figure 6 shows a behavioral view of the Context Agent. After initialization, it will remain idle until it receives new information or requests from other agents. If a Usage Agent requests a list of mappable components, the Context Agent may run a simple query on its own database.

3.3 Architecture Variant II: Extended Component Agent

This second proposed architecture variant features one agent class less. Instead of heaving dedicated agents for each element like in Architecture Variant I the Component and Usage Agents are merged into one: an Extended Component Agent. It takes over the duties of both original agents, rendering them obsolete (Fig. 7).

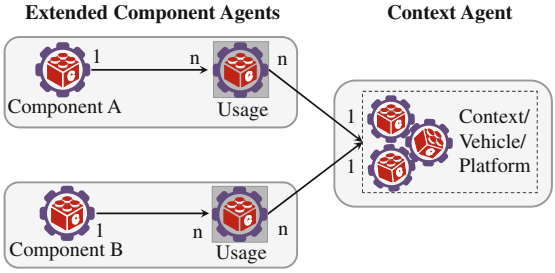

Fig. 7. Architecture Variant II - Extended Component Agent

Extended Component Agent. The Extended Component Agent represents a component and all its usages. It initializes once a component is first created and destroys when the component is disabled. Its main purpose is to maintain static information about the component such as *category* or *ID* and to conduct the mappings on other components. Furthermore, the Extended Component Agent knows all revisions of a component and retains about its different usages. The Component Agent monitors the Engineers' actions within the CAD-Program and tracking changes on the component. It becomes reactive upon incoming information requests from other agents or when the Engineer assigns a current revision to a new context. In the latter case, the Component Agent will contact the respective Context Agent in order to negotiate a component mapping.

Figure 8 shows a behavioral view of the Extended Component Agent. On initialization, a new Extended Component Agent receives parameterized information such as *category*, *component*, *function*, *rules*, and *side*. It idles in its main loop until either the underlying component is disabled which results in destruction of the Extended Component Agent, another Extended Component Agent sends a mapping request or it receives new information. The latter happens if another agent sends new information to the Extended Component Agent or the

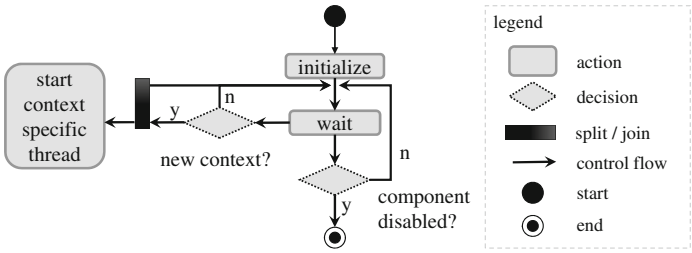

Fig. 8. Extended component agent behavioral view

Engineer performs changes on the underlying component. If a current revision is assigned to a new context, the Extended Component Agent will contact the Context Agent with the parameters *context*, *revision*, *validation*, and *category*. It opens a new thread in its behavior, dedicated to handle all context specific matters, i.e. the matters that in Architecture Variant I have been covered by the Usage Agent.

Context Agent. The Context Agent in Architecture Variant II has not changed compared to Architecture Variant I. Both its tasks and behavior can be modeled exactly the same way.

3.4 Architecture Variant III: Vertically Extended Usage Agent

Architecture Variant III models the idea of having a vertical approach at the Usage Agent. Matching components from two different sides may be represented by a single agent within a specific context. Figure 9 shows an overview of proposed agent classes, marked by boxes.

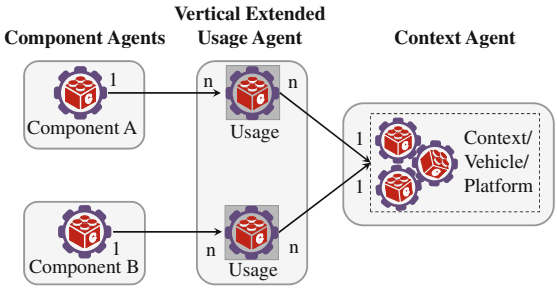

Fig. 9. Architecture variant III - vertically extended usage agent

Component Agent. This agent's tasks and behavior stay the same, aside from a small detail as shown in Fig. 10. Instead of triggering the initialization of a Vertically Extended Usage Agent, the Component Agent notifies the respective

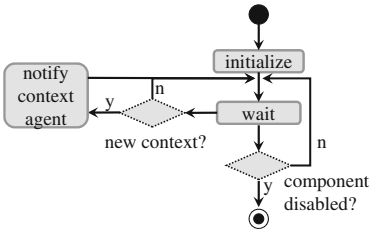

Fig. 10. Component agent behavioral view

Context Agent which will then proceed to ensure the mapping is processed. As soon as a component within a context has a Vertically Extended Usage Agent assigned, the respective Component Agents can update the information base of that Usage Agent.

Vertically Extended Usage Agent. The Vertically Extended Usage Agent functions similar to the regular Usage Agent known from Architecture Variant I. However, it will not try to achieve a mapping on its own. Instead, it remains passive until contacted by the Context Agent. Figure 11 shows the behavior in detail.

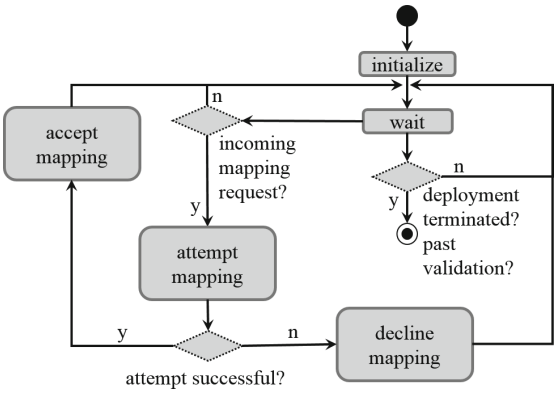

Fig. 11. Vertically extended usage agent behavioral view

Context Agent. The Context Agent in Architecture Variant III assumes a more active role than in Architecture Variant I and II. Once it receives a notification from a Component Agent about a new Component within its context, it tries to convey it to all existing Vertically Extended Usage Agents. If that fails, the new component is considered not mappable in this cycle and a new Usage Agent is deployed, representing the component. In the next cycle another new component may join the context and is shown to all existing Vertically Extended Usage Agents in order to find a match. Figure 12 shows the behavior in detail.

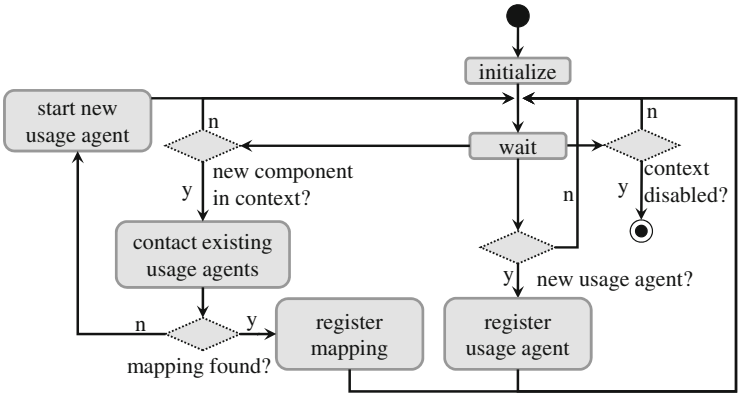

Fig. 12. Context agent behavioral view

3.5 Architecture Variant IV: Extended Context Agent

This architecture variant is based on the idea of having Component Agents directly communicating with an Extended Context Agent, which is processing all context related matters, including the mapping. Figure 13 shows an overview of proposed agent classes, marked by boxes.

Component Agent. This agent's tasks and behavior remain unchanged, aside from a small detail as already shown in Fig. 10. Instead of triggering the initialization of a Usage Agent, the Component Agent notifies the respective Context Agent which will then proceed to ensure the mapping is processed.

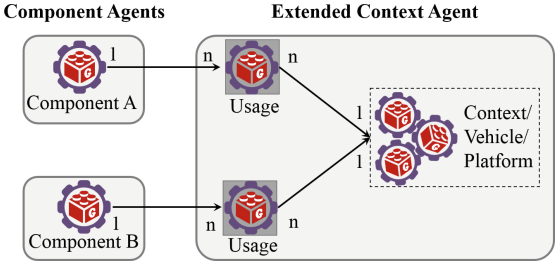

Fig. 13. Architecture Variant IV - Extended Context Agent

Extended Context Agent. As already stated, the Extended Context Agent takes over all tasks of both the regular Context Agent as well as the Usage Agent. The Extended Context Agent holds context-specific data, especially about the mappings. When a new component enters the context, the Extended Context Agent will try to map this component on an already existing component within

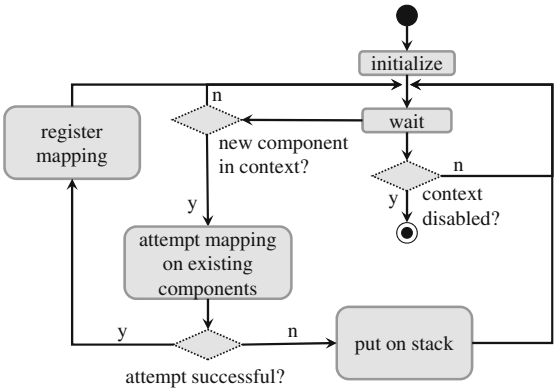

Fig. 14. Extended Context Agent behavioral view

the context. If there are no mappable components in the context or no mapping was found, the component is moved on a stack for later. As soon as new context-specific information comes up or another new component joins the context, the Extended Context Agent again tries to map the components. Figure 14 shows the behavior in detail.

3.6 Architecture Variant V: Vertically Extended Component Agent

Lastly, an agent architecture could be modeled with a Vertically Extended Component Agent which vertically connects components from two different sides.

However, we consider this architectural variant not applicable to our case for several reasons. Firstly, actual mappings can only exist between two revisions of a component and are thus not directly applicable on underlying components. Secondly, because of the decentralized nature of the problem and the generation of data throughout different brands and departments, Vertically Extended Component Agents would have to act across sites, i.e. they would have to exist in multiple spaces at the same time. Unlike in the prior architecture variants where context-related agents used to exist within a context only.

4 Sketch of the Algorithm for Part-Geometry Mapping

To apply the model introduced so far, finding an efficient and automatable way of mapping components is mandatory. At this point, an outline of such an algorithm is given. An actual implementation is planned for future work. To recapture, the agent architecture variants from Sect. 3 introduced the component term which, when set into a *context*, comes with properties relevant for the mapping process. Namely these properties are *category*, *function*[1], *rules*, *side*, and *validity*. Where

[1] It should be noted that the function property is optional as not every component fulfills a specific function. For example, a simple screw that is used a few hundred times across different locations within a vehicle. It does to some extend contribute to several functions but cannot be connected to one specific function.

rules are divided into *configuration rules* and *connection rules*, as proposed by Wagner in [15]. Configuration rules can be considered as *internal*, meaning they specify certain regulations when creating or changing the component. For example, a part needs to have a material specified. Connection rules on the other hand can be considered *external*, as they regulate connections between components. These rules may *force* or *forbid* a certain matter. For example, a geometry of a rim has a connection rule in place that dictates a mapped part must be made of aluminum.

In preparation of this paper, known consensus algorithms have been reviewed. Primary subjects of investigation were the Paxos protocol [7] and Raft [9]. However, these algorithms have not been found suitable for the task at hand as their main purpose is to coordinate client-server systems with a (fail safe) redundant server architecture. An alternative approach which we studied is based on matching algorithms known from the field of Operations Research / Graph Theory. These methods were also rendered not suitable as reviewed matching algorithms implicitely assume that matching compatibilities are a known fact. Thus, a new, special algorithm to solve the mapping problem is needed. The idea of which is outlined below.

In a first step the *respective agent*[2] tries to find a *proper subset of other components* that could fit his own. That means it compiles a subset of components in which every has *exactly not the same side* as the own component but *exactly the same category*. In extension, the respective agent rules out components with *validity data* that does not exactly fit or include its own validity.

That leaves the respective agent with a proper subset of all components within the context. It proceeds to contact these components' respective agents in order to *attempt a mapping*, i.e. enter mapping negotiations.[3] Therefore, both agents send each other *information sets* with data about the *underlying component*, *positional information*, and its *function*. Both agents proceed to check each others' information set against their own data and connection rules. If no force or forbid connection rules fail, and function and position parameters approximately map, an agent sends an accept to the other agent. If the other agent returns an accept a *preliminary mapping* is established. Both agents proceed to save the preliminary mapping and contact their *respective users*[4] for *confirmation*. An accept by the user results in a *confirmed mapping* while a decline revokes the preliminary mapping. More so, the respective agents will blacklist declined mappings and refrain from attempting a mapping with these components in the future again to prevent infite looping. Algorithm 1 summarizes this behaviour in pseudo code.

[2] Depending on the architecture variant this could be the Extended Component, Extended Context or (Extended) Usage Agent.

[3] It should be noted that the whole communication part does not apply to Architecture Variant IV because there is only one agent (the Extended Context Agent) which handles the mapping negotiations internally.

[4] Engineers and Purchasers.

Algorithm 1. Part-Geometry-Mapping

```
procedure mapComponents(List<Components> mappableComps)
    for all Components c: mappableComps do
        Agent a = c.getAgent()                          ▷ Identify the comp's agent
        Set inInfo = a.attemptMapping(outInfo)          ▷ Contact the comp's agent
        bool positionMapped = tryPosMapping(inInfo)     ▷ Check incoming data
        bool ruleCheck = checkRules(inInfo)             ▷ Check connection rules
        bool outResp = positionMapped & ruleCheck
        bool inResp = a.sendResponse(outResp)           ▷ Send and retreive responses
        if outResp & inResp then                        ▷ If both agents agree on mapping
            savePreliminaryMapping()
            requestConfirmation()                       ▷ Contact user for confirmation
    return mappedComponents
```

As shown, an agent-based approach might not be able to completely automate the mapping process. However, intelligent agents are able to cut down the number of possible mappings at least, saving the users time, and assisting the goal of establishing transparency.

5 Comparison

This section discusses the five agent architecture variants shown in Sect. 3 and their stance on the mapping algorithm outlined in Sect. 4.

Architecture Variant I - Dedicated Element Agents. Covers the most intuitive approach in which every element within the development process is representend by a single, autonomous agent. It could be argued that this approach is a variant of a peer-to-peer architecture as there are no real leading agents or agents overseeing the bigger picture. It certainly fits the idea of a decentralized, intelligent multi-agent system quite well and offers a lot of of room for extensions. The downside is the massive amount of communication needed between agents and possibly redundant data storage which needs to be kept consistent and made easily accessible to the users. The only agent class where intelligence is really needed is the Usage Agent as it is responsible for the negotiation of preliminary mappings. All other agents could be modeled as reactive sub-systems which follow a strict behavioural pattern and mainly serve as data storages or communication arrays between Usage Agents and users.

Architecture Variant II - Extended Component Agent. Merges the Component Agent and its Usage Agents into one. This reduces the communication needed but adds to the complexity of the Extended Component Agent which now has to manage all its contexts. As a revision is basically just an extension of its component, it might be more logical to represent both elements in one agent. Queries about what revisions are assigned to which contexts could be answered faster than in Architecture Variant I as it is not necessary anymore

to contact the Usage Agents in order to acquire this context-specific information. The intelligence is moved into the Extended Component Agent rendering the Context Agent as sole data storage and administration system. Fewer agent classes might make future extensions more difficult and change the system to be less flexible.

Architecture Variant III - Vertically Extended Usage Agent. Tries to realize a comprehensive approach where revisions from different sides are represented by a single agent. This approach causes less communication efforts during the mapping process as it is all handeled internally by the Vertically Extended Usage Agent, which in extend plays the only intelligent role again. If no match is found on the first try, the component's revision exists within the context but lacks an agent to represent it until a mapping has been found. Furthermore, once two revisions from different sides have been succesfully mapped, the Vertically Extended Usage Agent needs to act across dispersed locations and departments. Thus, Architecture Variant III might be feasible but probably not optimal.

Architecture Variant IV - Extended Context Agent. Leads the system's intelligence away from the components and towards the context. It is by that more centralized but less communication efforts are needed. Contrary to the peer-to-peer approach from Architecture Variant I, this architecture variant establishes a hirarchy, in which the Extended Context Agent is overseeing the whole context and all its matters. Component Agents merely act as registers and entry points for users who wish to find out where the revisions of a component are used in. All context-specific information is stored within the Extended Context Agent and by that leaves most likely the fewest redundancies of all architecture variants. However, implementing future extensions might be harder.

Solely based on the requirements established in Sect. 3 Architecture Variants I-IV deliver satisfying concepts. Architecture Variants II and IV seem to be the ones with the fewest drawbacks while Architecture Variant I offers the most flexibility and should be easy to extend with more functionality. Future implementations or more desired functionalities might lead to a clearer result or alter the architecture in ways which could not be covered in this paper.

Table 1 summarizes the prior comparison:

Table 1. Comparison of Feasible Agent Architecture Variants

	Architecture Variant I	AV II	AV III	AV IV
Communication	high	medium	medium	low
Complexity	low	medium	medium	high
Redundancy	high	low	medium	low
Flexibility	high	medium	medium	low
Location of Possible Intelligence	Usage Agent	Extended Component Agent	Vertically Extended Agent	Extended Context Agent

6 Conclusion

In this concept paper, an agent-based approach on automating a key development process in the automotive industry, the connection of geometric data (*geometries*) and logistical data (*parts*), within a decentralized environment has been modeled. Based on the concept of a *Virtual Product Model (Component)* by Kehl et al. in [6], which was applied on three use cases derived from case studies in the automotive industry, five agent architecture variants have been developed. Four of these five architecture variants have been rendered feasible in order to support both the given requirement to (partly) automate the *part-geometry-mapping* as well as contributing to a *more transparent system* with *changes made traceable*. The four feasible agent architecture variants have been discussed in detail, especially regarding their expected incurring communication, data redundancy, and extendability. In addition, an algorithm functioning in each of the agent architecture variants for the mapping process itself has been outlined which, based on given data, is not able to determine mappings fully on its own but is at least capable of significantly curtailing the amount of possible mappings.

Future work will include a prototyped implementation of the agent architecture variants. On a more conceptual level, additional requirements could be determined and the architecture variants could be extended and enriched with more functionality.

References

1. Cheng, H., Chu, X.: A network-based assessment approach for change impacts on complex product. J. Intell. Manuf. **23**(4), 1419–1431 (2012). http://dx.doi.org/10.1007/s10845-010-0454-8
2. Clarkson, J.P., Simons, C., Eckert, C.: Predicting change propagation in complex design. In: ASME 2001 Design Engineering Technical Conferences and Computers and Information in Engineering Conference. Pittsburgh, Pennsylvania (2001)
3. Fricke, E., Gebhard, B., Negele, H., Igenbergs, E.: Coping with changes: Causes, findings, and strategies. Syst. Eng. **3**(4), 169–179 (2000)
4. Glauche, M., Rebel, M., Müller, J.P.: Produktstrukturierung als Erfolgsfaktor: Systematisierung und Analyse von Einflussfaktoren der Produktstrukturierung. ZWF - Zeitschrift für wirtschaftlichen Fabrikbetrieb **11**, 878–881 (2013)
5. Jarratt, T., Eckert, C.M., Caldwell, N., Clarkson, P.J.: Engineering change: an overview and perspective on the literature. Res. Eng. Des. **22**(2), 103–124 (2011). http://dx.doi.org/10.1007/s00163-010-0097-y
6. Kehl, S., Stiefel, P., Müller, J.P.: Changes on changes: Towards an agent-based approach for managing complexity in decentralized product development. In: International Conference on Engineering Design (ICED 15). vol. 3, pp. 220–228. Milan, Italy (2015)
7. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998)
8. Michael Wooldridge: An Introduction to MultiAgent Systems. John Wiley & Sons, 2 edn (2009)

9. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference. pp. 305–320. USENIX ATC 2014, USENIX Association, Berkeley, CA, USA (2014). http://dl.acm.org/citation.cfm?id=2643634.2643666
10. Simon, H.A.: The sciences of the artificial, vol. 3. MIT Press, Cambridge (1996)
11. Tseng, M.M., Jiao, J.: Mass customization: 25. In: Handbook of Industrial Engineering, pp. 684–709. John Wiley & Sons, Inc (2007). http://dx.doi.org/10.1002/9780470172339.ch25
12. Verband der Automobilindustrie (VDA): Auto 2008 - jahresbericht. online (July 2008).www.vda.de/de/services/Publikationen/Publikation.489.html
13. Verband der Automobilindustrie (VDA): Jahresbericht 2012. online (July 2012). https://www.vda.de/de/services/Publikationen/jahresbericht-2012.html
14. Verband der Automobilindustrie (VDA): Jahresbericht 2013. online (August 2013). https://www.vda.de/de/services/Publikationen/jahresbericht-2013.html
15. Wagner, T.: Agentenunterstütztes engineering von automatisierungsanlagen. atp-online Automatisierungstechnische. Praxis **50**(4), 68–75 (2008)
16. Xu, H.C., Xu, X.F., He, T.: Research on transformation engineering bom into manufacturing bom based on bop. Appl. Mech. Mater. **10–12**, 99–103 (2008)