KNOWLEDGE-BASED ADAPTIVE AGENTS FOR MANUFACTURING DOMAINS

Stefano Borgo
Amedeo Cesta
Andrea Orlandini
Alessandro Umbrico
ISTC – Institute of Cognitive Science and Technology
CNR – National Research Council of Italy
{stefano.borgo,amedeo.cesta,andrea.orlandini,alessandro.umbrico}@istc.cnr.it

ABSTRACT

Modern production systems are increasingly using artificial agents (e.g., robots) of different kinds. Ideally, these agents should be able to recognize the state of the world, to act optimizing their work towards the achievement of a set of goals, to change the plan of action when problems arise, and to collaborate with other artificial and human agents. The development of such an ideal agent presents several challenges. We concentrate on two of them: the construction of a single and coherent knowledge base which includes different types of knowledge with which to understand and reason on the state of the world in a human-like way; and the isolation of types of contexts that the agent can exploit to make sense of the actual situation from a perspective and to interact accordingly with humans. We show how to build such a knowledge base (KB) and how it can be updated as time passes. The KB we propose is based on a foundational ontology, is cognitively inspired and includes a notion of context to discriminate information. The KB has been partially implemented to test the use and suitability of the knowledge representation for the agent's control model via a temporal planning and execution system. Some experimental results showing the feasibility of our approach are reported.

KEYWORDS

Autonomous agent, Adaptive agent, Knowledge-based modeling, Ontology, Context, Production system, Planning

1. INTRODUCTION

The research community in Robotics and Artificial Intelligence has been building sophisticated robots for decades applying many available technologies. Still a limited understanding exist on what are the essential features of a generic artificial robotic agent. Furthermore, today robots are successfully applied in a large spectrum of applications (i.e., from simple washing machines to swarm robots and even humanoids), but it is still unclear how to best integrate different parts and modules into a single entity that can smoothly operate in an environment, understanding the actual situations and acting in order to achieve some given goals. These problems also hold in the more restricted area of industrial robots, which are built to enhance the productivity at the shop floor, and which operate in controlled, or at least constrained, environments.

In the manufacturing context, an open problem for researchers is the construction of systems that can quickly adapt (and possibly anticipate) changes in the production requirements [33]. Moreover, the concept of Industry 4.0 [45] is pushing manufacturing systems to evolve towards customer-oriented and personalized production, while trying to guarantee the advantages of mass production systems in terms of both productivity and costs [43]. Such systems are being conceptualized counting on the employment of highly flexible and reconfigurable machines [44]. Traditional approaches rely on centralized hierarchical control structures, not easily adaptable to different production settings without strongly affecting the productivity of the shop-floor. Indeed, they usually require major overhauls of their control code each time that any sort of system adaptation and/or reconfiguration is required. This is especially true in dynamic working environments like *Reconfigurable Manufacturing Systems* (RMSs) [23] where the actual capabilities of an agent and even the production processes of the factory may change very quickly. Different configurations of the shop-floor (e.g., increasing the number of available working machines) or the introduction of new production capabilities (e.g., considering new type of work-pieces or tool) may affect the type of tasks agents carry out as well as the related production procedures. Artificial Intelligence (AI) techniques like, e.g., plan-based control, usually rely on a once-defined, *static* model of the world which could become obsolete very soon in dynamic contexts. The model requires a great design effort to capture all the

possible configurations and it needs a continuous *maintenance* which may negatively affect productivity of the plant.

The research goal of this work is to enhance the flexibility of "classical" plan-based control architectures by integrating *knowledge representation and reasoning* mechanisms into the control loop (i.e., the closed-loop control of high-level activities, goals and states). The pursued approach is that semantic technologies can provide the representation and reasoning functionalities that robotic agents need to dynamically recognize particular production settings and adapt control models accordingly. The paper considers a concrete case study to provide an example of how knowledge representation techniques can significantly improve *flexibility* and *adaptability* of AI plan-based architectures by dynamically regenerating the control model according to the specific state of the working environment.

More specifically, this paper presents parts of a new cognitive architecture, suitable for artificial agents such as, e.g., robots, which leverages an ontological approach for knowledge classification and management in order to smooth the information flow from the knowledge about the environment and robot's goals (desired states) to the planning and execution module. More precisely, the proposed contribution relates to i) the structure of a Knowledge Base (KB) of the robot, ii) a *knowledge manager* that stores the *model of the world* and infer new information to recognize operational situations, and how the KB can be exploited to gather the information needed by the agent's control module where planning and execution take place. Admittedly, this approach takes care of just one of the local properties, the KB module, and one of the global properties, the connection between the KB and the control module. Yet, this connection is today critical: it involves the manipulation of symbolic information at different levels of generality and its translation into state variables for controlling each component of the robot and, consequently, its interaction with the environment.

Structure of the paper: Section 2 presents the state of the art on the use of ontologies in robotics. Section 3 presents the structure of the considered KB. Section 4 introduces the DOLCE ontology and then presents specific extensions to cover engineering high-level notions. In the same section, a contextual classification of information is presented. Section 5 describes how the KB is exploited for extracting the information needed by the control module . The general plan-based control loop is presented in Section 6 and some tests for the validation of the approach are presented and discussed in Section 7. Section 8 concludes the paper.

2. THE STATE OF THE ART

In robotics and more generally in manufacturing, ontologies have been recognized as true enablers of adaptable and flexible systems compared to classic approaches [28,34] and, consequently, have been exploited in the attempt to design more autonomous, flexible, adaptive and proactive artificial agents. Since researchers have applied ontologies to solve or at least mitigate a variety of problems, applications differ in their assumptions and goals.

In [31], a Robot knowledge framework (OMRKF) is exploited. OMRKF includes a series of articulated ontologies layers, including a robot-centered and a human-centered ontology. Beside a perception layer, enabling sensory data, the system includes an object layer (model), a context layer and an activity layer. The lack of the foundational approach can be seen in the object classification, e.g., "living room" is classified as a space region and not as the role of the region (the problem becomes clear by observing that a region of space is fixed while the living room can be located in different parts of the building at different times, and can even disappear from the building), as well as in the lack of activity vs. functionality distinction, e.g., "avoid obstacle" is not a behavior but a function which can be *implemented* by different behaviors like, e.g., moving away or turning around.

Relatively to the connection between the KB and the planning module, [19] exploits a model filtering approach based on a Hierarchical Task Network (HTN). The agent's knowledge of the environment is stored in a fixed ontology and some filters on this knowledge are set up. Given a planning task, the system selects one of the filters to isolate a suitable subset of the system's knowledge and uses this subset to constrain the plan by deleting non-reachable constants. While this technique can be efficient in terms of plan adaptation, the knowledge is only filtered, thus cannot be augmented nor modified, not even contextualized to the problem showing a lack in flexibility. It is important to generate planning models tailored to the actual status of the agent and its environment: changes on these can easily make the starting plan obsolete as it happens, for instance, when the capabilities of the agent are affected by a sudden lack of resources or a (perhaps partial) failure of a component.

[3] also treats tasks within the HTN approach and uses an ontology for the domain knowledge which encodes

task information. The goal is to combine procedural knowledge and ontological knowledge. The planning knowledge is modeled by adding task concepts to the ontology and also predefined decomposition methods that are applied in order to decompose task concepts into executable tasks. New methods can be inferred by reasoning on combinations of executable tasks. The paper focuses on the correspondence between subsumptions among task concepts in the ontology and corresponding decompositions in the planning domain. The starting ontology is assumed to be given and is clearly limited by the expressivity of the chosen language (Description Logic). This makes reasoning feasible at the cost of information quality: even basic ordering information among the tasks must be stored outside the ontology. Other research projects, like KnowRob [32] and ORO [25] focus on learning and symbol grounding and use ontologies for obtaining an action-based knowledge representation able to support cognitive functionalities. At the ontological level, these knowledge systems show problems similar to those discussed earlier (e.g., functionality is confused with activity so that it is not possible to "discover" new ways of performing functions). A further issue is the lack of stability in the ontology since some parts are developed on demand or depend on the interaction with other agents, that is, the ontology itself is data dependent. This approach can potentially introduce consistency problems that, unfortunately, are hard to detect in a dynamic knowledge base.

Ontologies have been applied to increase flexibility in modeling and planning of, e.g., mechatronic devices [2], resources in collaborative environments and the whole enterprise [30], collaborative robots [21] and navigation robots [12]. [2] uses a fixed ontology to collects static and dynamic information relative to a robot. The basic actions of the robot are hard-coded but the ontological system, which includes a PDDL ontology where knowledge about actions is stored, adds some flexibility like the possibility to learn articulated actions and to act with partial information, e.g., information about the location of the object to move. In this application, beside the lack of functionality/activity distinction, the robot has very limited knowledge of the environment. [30] describes an ontological model aimed to represent the resource capabilities for the development of products and processes. The model is part of a larger ontology for collaborative and integrated development of products, processes and resources.

This approach is based on the foundational ontology DOLCE, which we introduce in the next section, and takes a general perspective. While this work goes in the right direction, it is not aimed to furnish a knowledge system targeting single artificial agents; the goal is to be able to integrate all the relevant perspectives in the enterprise including the modeling of both engineering and management activities. The approach does not include contextual knowledge. [21] focuses on the scenario of robot guides offering a tour of a building, and develop a system for heterogeneous robot collaboration (e.g., receptionists and companions). The idea is to use ontologies for contextual information: there is an ontology for the user, one for the robot and one for the topic. For instance, the user ontology contains the user's profile, topics of interests, and prior knowledge. This information is updated during the tour. Since the work concentrates on the collaborative interaction among robots, there is no particular effort towards flexible knowledge modeling. The ontologies are fixed: the topics are preselected and so is the content associated to the levels of detail or the user profiles. In [12] a robotic platform based on contextual knowledge design has been proposed to enhance performance for behavior specialization, perception, navigation, exploration, localization and mapping tasks. The idea is to use contexts as classes of problem solving situations and to use behavior knowledge to select a suitable plan.

The underlying view is to see high-level information as environment contextual knowledge. Similarly, goal information and agent's information are distinguished as mission and introspective contexts, respectively. This approach lacks the framework of an ontology and indeed the environment contexts are either left to the designer or built on top of the sensors' data that the robot itself generates. An ontological approach would help to structure the classification of entities (after all a physical object may be a landmark in a certain context but does not ceases to be a physical object because of this) and to clarify how and when different contexts overlap.

Finally, an ontology-based reconfiguration agent is presented in [1] where an ontology is used to formalize knowledge of the manufacturing environment. The proposal is close to the view we present below but has important differences. First, it uses an ontology developed *ad hoc* for the given scenario. Second, it does not introduce contexts. Third, it is unclear how it can deal with device's failures. We improve on the system by adopting a foundational ontology with which it is possible to reason at the functional level and to integrate the system with contextual information.

3. FLEXIBLE CONTROL ARCHITECTURE

Despite the variety of uses of ontology in robotic applications, the on-line management of information needed for dynamic planning remains an open issue. This is the main considered challenge to deal with to develop adaptive autonomous robots. We propose a radical approach relying on two elements: (i) a foundational ontology to organize the information, and (ii) a knowledge-based control loop for updating the data and managing the flow of information to the planning system. The aim is to realize a system that can be actually deployed. Indeed, we consider an industrial case study as a first validation scenario [5].

3.1. A Case Study

A real pilot plant is considered for validation consisting of a reconfigurable manufacturing system (RMS) for recycling Printed Circuit Boards (PCB). The plant is composed of different automatic and manual machines devoted to perform loading/unloading, testing, repairing and shredding of PCBs and a reconfigurable transportation system connecting them. The transportation system is composed by 15 reconfigurable mechatronic components, called Transportation Modules (TM). Figure 1 shows an example of the TM used in the pilot plant. The figure shows also the schema of a TM together with the supported transfer service. The set of supported services depends on the configuration of a TM.

In general, all the TMs composing the reconfigurable transportation system of the case study can support two main (straight) transfer services. They are the Front transfer service and the Back transfer service, respectively tagged with F and B in Figure 1. Then, each TM can support a number of cross-transfer services that range from a minimum of zero to a maximum of three. A cross transfer service enables Left (LCx) and Right (RCx) transfers between adjacent TMs. The schema in Figure 1 shows the configuration of a TM equipped with two cross-transfer services that allow respectively LC1/RC1 and LC2/RC2 transportation directions.

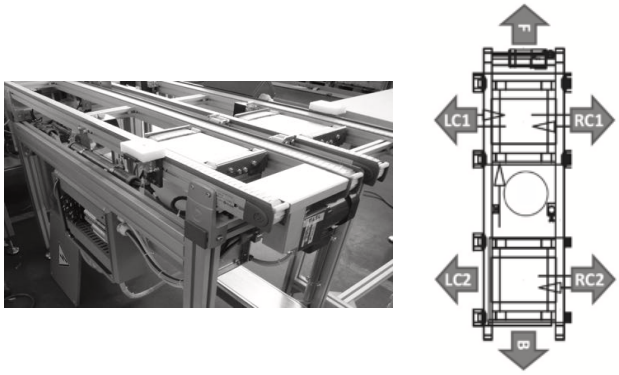

Figure 1 An example of a Transportation Module and the schema of a possible configuration

The destination of a pallet carrying a PCB is computed at runtime while the production process is carried out (e.g., also depending on the results at the testing station, the repairing station or the loading/unloading cell). Thus, the final destination of a pallet is available only at execution time requiring the transportation system to keep adapting to the newly generated destination. The TMs of the transportation system cooperate in order to define and optimize the paths the pallets have to follow to reach their destinations [5,13] overcoming problems due to failures and unresponsive agents.

3.2. Putting Knowledge and Control in a Loop

The foundational ontology, needed to ensure the coverage of the domain and a coherent global management of knowledge, and the context framework for factual knowledge that will be introduced in Section 4, form the structure of the knowledge management of the artificial agent. We now need to show how to make an operational use of the knowledge management, that is, we need to articulate the relationship between knowing the world and acting on the world. In particular, we are interested in using the agent knowledge for autonomous behavior to enhance the capability of an artificial agent. To this purpose we integrate distinct cognitive functions as shown in Figure 2. The figure shows the overall cognitive architecture resulting from the integration of a *Knowledge Manager* (top left), which contains the *built-in* know-how of the agent on its structure and the environment, and a *Deliberative Controller*, which constitutes a "classical" a plan-based control architecture like those described, e.g., in [24].

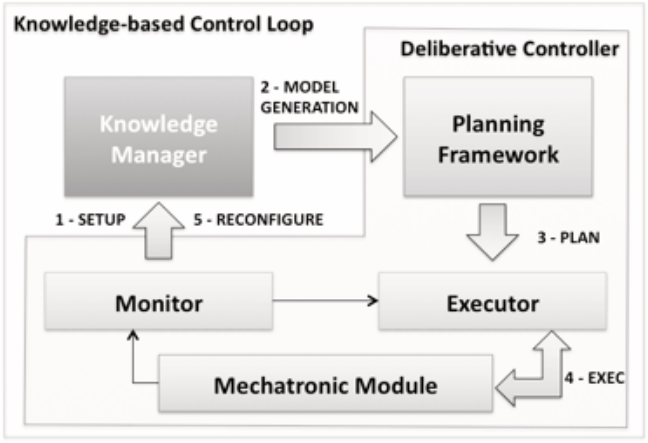

Figure 2 The proposed integration between knowledge management and deliberative control in an agent architecture

Plan-based control architectures realize closed-loop controllers that allow an artificial agent to autonomously take decisions and execute actions needed to achieve some objectives. Such architectures are usually organized in three layers: (i) a deliberative layer which provides the agent with the capability of synthesizing the actions needed to achieve a goal (i.e., the Planning Framework in Figure 2); (ii) an executive layer which executes actions of a plan and continuously monitor their actual outcome with respect to the expected status of the system and the environment (i.e., the Monitor and the Executor in Figure 2); (iii) the the mechatronic system (and its functional processes) which represents the system and the environment to be controlled (i.e., the Mechatronic Module in Figure 2 and the related transportation system).

Starting from the bottom, the mechatronic system and the environment determine the capabilities of the agent and the internal/external dynamics that must be taken into account in order to properly perform actions. Given the considered case study, the mechatronic system consists of a particular TM of the plant that must be controlled, the configuration of the overall transportation system (global topology) and the internal configuration of the TM as well as its connections with other TMs of the plant (local topology) are supposed to be known. The Monitor and the Executor are responsible of "physically" interacting with the TM by sending commands and receiving signals about their execution (i.e., feedbacks). Specifically, the Executor sends commands to the TM according to the actions that compose the plan to be executed. The Monitor receives signals about the (either positive or negative) outcome of the execution of such commands from the TM and checks whether the actual status of the mechatronic system complies with the plan or not. The Planning Framework synthesizes a set of "primitive" transportation activities (i.e. actions) that the TM must perform in order to carry out a set of desired (complex) transportation tasks. The Planning Framework relies on a planning model modeling the actual capabilities of the considered TM and the associated local topology. The deliberative layer synthesizes plans according to this general model. In case the plan execution leads to a failure (i.e., the actual status of the physical system does not comply with the plan) then, a "replanning process" is trigger for the Planning Framework in order to (re)generate a new plan according to the actual status of the mechatronic system.

The Deliberative Controller in Figure 2 relies on a static planning model which completely characterizes the capabilities of a TM and the associated working environment. However, such a model is not capable of dynamically capturing changes in the configuration of the transportation system like e.g., changes concerning the local topology of a TM or changes concerning the internal configuration of a TM and therefore affecting the associated capabilities. The Knowledge Manager enhances the flexibility of the Deliberative Controller by dynamically generating planning models. In Section 3.3 will be shown how the Knowledge Manager leverages an ontological approach to process low-level signals from a TM and dynamically infer its internal configuration, the local topology and active capabilities (i.e., the set of transportation tasks a TM can actually perform). This information is represented and managed through a Knowledge Base (KB) which is continuously updated according to the signals/events received from the Mechatronic Module in Figure 2. Then, a model generation process dynamically creates a new planning model every time a change in the KB occurs. Specifically, a new planning model is needed every time a change due by, e.g., a physical reconfiguration, in the actual capabilities of a TM is detected.

3.3. The Knowledge-based Control Loop Architecture

The goal is to have a coherent and optimized flow of information from the *Knowledge Manager* to the *Deliberative Controller* and to extend the capabilities of the overall system by leveraging knowledge processing capabilities. The *Deliberative Controller* is a complex component: it realizes a sense-plan-act cycle by means of a *Planning Framework* (Figure 2, top right) and an *Executor system* (bottom right). In our implementation both modules use timeline-based technology [14]. The Planning Framework generates a planning model to

synthesize the set of commands and signals for the continuous planning and execution actions of the artificial agent.

The *Mechatronic Module* (Figure 2, bottom) is the composition of a *Control Software* and a *Mechatronic Component* (not shown in the figure) that are typical in industrial components like, e.g., transportation modules, robots or working machines. This control software is usually based on standard reference models (e.g., IEC61499) and each mechatronic component is represented by dedicated hardware/software resources encapsulating the module control logic. The planning model contains an abstraction of the device to be controlled, the environment's parameters in which the device operates, and a number of relevant constraints that guarantee physical consistency during execution. Note that the planning model is a *static* representation of the domain: it keeps track of just a subset of possible changes of the agent configuration and of the environment that are directly caused by the plan execution.

We call *Knowledge-based Control Loop* (KBCL) the interaction across these modules that enables the agent to dynamically represent its capabilities, its internal status and its environmental situation, and to automatically infer the set of available functionalities to generate a coherent planning model. Besides a classical *sense-plan-act* loop , the KBCL continuously monitors the environment collecting information from the agent's components (via sensors, actuators and interactions with other agents) and updates the state of the environment in the Knowledge Base.

3.4. The KBCL at Runtime

The KBCL supports a Setup phase (Point 1 in Figure 2), when the mechatronic device is activated, generating the initial KB of the agent. More specifically, the *Monitor* (Figure 2, bottom left) collects the raw data from the *Mechatronic Module* with which a knowledge processing mechanism (i) initializes the KB by adding the instances that represent the actual state of the device (Point 1 of Figure 2) and (ii) dynamically generates the control model providing a first planning specification (Point 2). Then the planning system generates a production plan (Point 3) and the plan execution is performed through the executive system (Point 4). When the *Monitor* detects a change in the structure of the agent and/or its collaborators (due, for example, to a total or partial failure of a sensor/actuator or of a neighbor), the KBCL process starts a Reconfiguration phase (Point 5) entailing the update of the KB, e.g., adding/removing instances, and starting a new iteration of the overall loop.

The KB is updated only when the detected changes prevent or possibly deteriorate (depending on the plan rules) the execution of the plan. It is worth underscoring that the Reconfiguration phase is activated in case of failures or when new capabilities are added. For instance, when a collaborating agent enters a maintenance period, its presence and capabilities are first deleted when entering the maintenance and re-inserted in the KB when operative again.

4. MODELING KNOWLEDGE WITH ONTOLOGY AND CONTEXTS

We have seen different applications of ontology and context highlighting some drawbacks crucial to our goal, namely, to provide a framework suitable to dynamically manage and control generic artificial agents. The aimed generality pushes us to look for a structure that is neither tailored to a specific type of agent nor to a specific type of situations, it is not based on an information model at the enterprise or shop floor level nor developed for some specific type of action.

To put it positively, we aim to build knowledge frameworks that should be evaluated in terms of flexibility. By flexibility we mean that the same framework can be used for different kinds of robots applied in different industrial scenarios and for different goals provided these scenarios and goals fall within the usual human common-sense perspective (this, e.g., excludes robots operating within a quantum perspective as needed in specialized domains). That is, one should be able to upload the same knowledge framework in different types of agent without compromising the quality of their functionalities, behaviors and reliability relatively to the available hardware. It follows that such knowledge framework must be independent from specific sensors and actuators, and can have only generic information on what could be in the environment. The advantage, if we succeed, is the design of robots with a new level of autonomy and adaptability compared to today's standards. Before building the knowledge structure, the ontological approach helps us to analyze the types of information that an industrial robot could face.

The first result of this analysis is the separation of two layers of information: *organizational* knowledge and *factual* knowledge. Organizational knowledge is the foundational knowledge, i.e., knowledge about the basic assumptions in the domain like the notion of object, agent, production etc. including their

relationships. This knowledge fixes what kind of entities, events and interactions there can be in general. Factual knowledge, instead, identifies how the actual scenario is out of all the possible configurations: which objects are present and where, which actions are executed and by which agent, which changes occur and to which object. Factual knowledge can be extended (without changing the foundational knowledge) as needed, e.g., to include knowledge about new devices (tools, machines) or changes in the shop floor layout. Changes in these two parts of the knowledge framework follow different principles and have different consequences. By keeping them apart, we can make them interoperate covering all the knowledge needed in production systems [35].

For the organizational knowledge, we start with the foundational ontology DOLCE, the Descriptive Ontology for Linguistic and Cognitive Engineering [26]. This is a domain-independent top-level ontology that has been exploited at different levels in the engineering and industrial domains, e.g., [9, 30, 29, 6]. DOLCE furnishes the basic structure of our knowledge system which we will enrich with domain knowledge, for instance adding the notions of artificial agent and of engineering function. The knowledge framework that we make available to an agent will be an extension of this ontological system. Since DOLCE is based on a first-order language with formal semantics, the ontology and the resulting knowledge base can be exploited via automatic reasoning.

4.1. DOLCE

The DOLCE ontology is a formal system built according to an explicit set of philosophical principles that guide its use and extension [26]. Our first interest is to introduce in DOLCE engineering notions which are central to our application concerns. Since DOLCE is a large and complex system, we cannot introduce it in detail but describe the minimal elements relevant to our work. We refer the interested reader to [26, 10] for motivations and technical aspects. DOLCE (Figure 3) focuses on *particulars*, as opposed to *universals*. Roughly speaking, a universal is an entity that is instantiated or concreted by other entities (like the property "being a tool" or "being a production process"). A particular, an element of the category PARTICULAR, is an entity that is not instantiated by other entities (like the Eiffel Tower in Paris or Donald Trump). PARTICULAR includes physical entities, abstract entities, events and even qualities as we will see below.

The DOLCE ontology formalizes the distinction between things like a car and an organization (this category is called ENDURANT), and events like transporting by means of a car and resting (category PERDURANT), see again Figure 3. The term 'object' is used in the ontology to capture a notion of unity as suggested by the partition of the category PHYSICAL ENDURANT (a subcategory of ENDURANT) into categories AMOUNT OF MATTER, like the plastic with which my water bottle is made, PHYSICAL OBJECT, like my car, and FEATURE. Features are entities that existentially depend on other objects, e.g., a bump on a road or the workspace for a robotic arm. We will also exploit two subcategories of PHYSICAL OBJECT, namely, AGENTIVE PHYSICAL OBJECT, e.g., a person, and NON-AGENTIVE PHYSICAL OBJECT, e.g., a drill.

DOLCE also provides a structure for individual qualities (elements of the category QUALITY like the weight of a given car), quality types (weight, color, and the like), quality spaces (spaces to classify weights, colors, etc.), and quality positions or *qualia* (informally, locations in quality spaces). These, together with measure spaces (where the quality positions get associated to a measure system and to numbers), are important to describe and compare devices and processes. The exact list of qualities may depend on the entity: *shape* and *weight* are usually taken as qualities of physical endurants, *duration* and *direction* as qualities of perdurants. An individual quality, e.g., the weight of my hammer, is associated with *one and only one* entity; it can be understood as the particular way in which that hammer instantiates the general property "having weight". That individual weight quality is what we measure when we put the hammer on a scale (if we put another hammer, no matter how similar, we would measure *another* individual quality, i.e., that of the second hammer even if the scale indicates exactly the same value). The change of an endurant in time is explained in DOLCE through the change of some of its individual qualities. For example, with the substitution or damaging of a component, the value of the weight quality of my car may change. DOLCE's taxonomic structure is pictured in Figure 3. Each node in the graph is a *category* of the ontology. A category is a subcategory of another if the latter occurs higher in the graph and there is an edge between the two. PARTICULAR is the top category. The direct subcategories of a given category form a partition. In the graph, dots indicate that not all the subcategories of that category are listed.

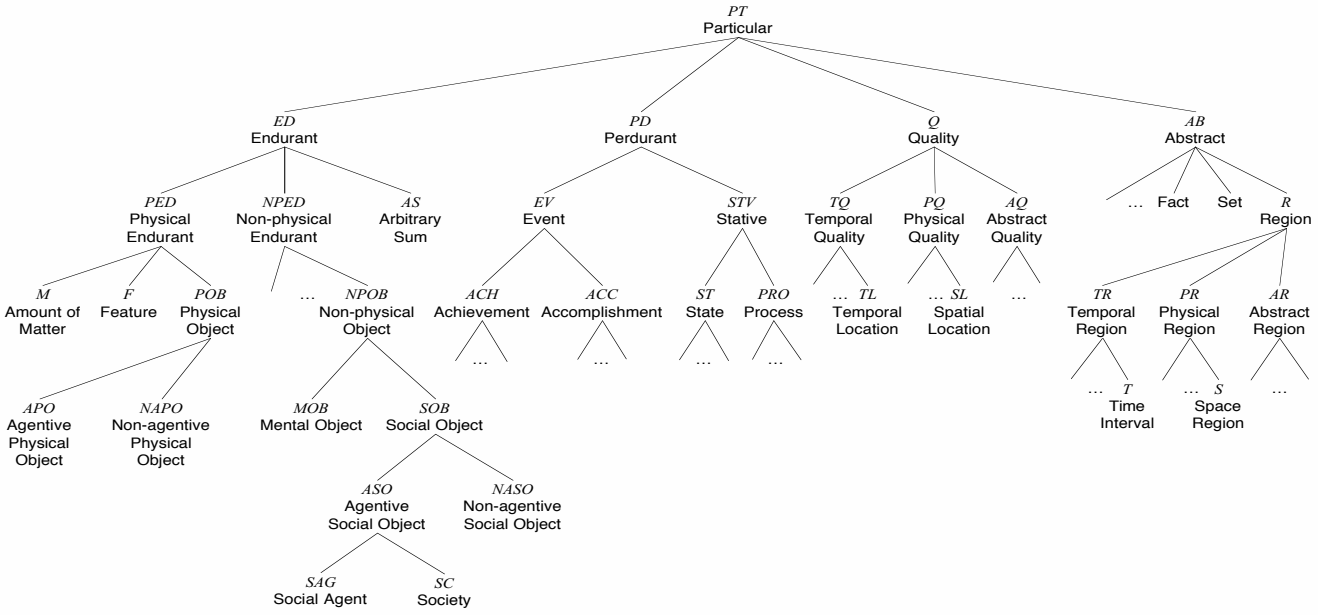

Figure 3 The DOLCE taxonomy of particulars

Some relations are particularly relevant for our work, e.g., the *parthood* relation: "*x* is part of *y*" (written: P(*x*, *y*)), with its cognates the *proper part* (written: PP(*x*, *y*)) and *overlap* relations (written: O(*x*, *y*)). These apply to pairs of endurants (e.g., the joint is part of the robotic arm) as well as to pairs of perdurants (e.g., riveting is part of the assembling process). On endurants, parthood has an additional temporal argument since an endurant may lose or gain parts throughout its existence (e.g., after substituting a switch in a radio, the old switch is not part of the radio). Another important relation is constitution, indicated by K: K(*x*, *y*, *t*) stands for "entity *x* constitutes *y* at time *t*", e.g., the amount of iron *x* constitutes the robot *y* at time *t* (this relation allows to say that part or all the iron *x* may be substituted over time without changing the identity of robot *y* like when substituting a worn out component).

4.2. An Ontological View of Artificial Agents and their Environment

Recently there has been increasing interest in the ontological modeling of artificial agents, and robots in particular [29], which led to an IEEE standard [17]. Today's approaches to robot modeling are the result of long reflections on the difference between types of agency but further work is needed. For instance, we find problematic the proposal to model a robot mixing the notions of object and of role taken in [17]. According to the standard, a robot is "an agentive device [...] in a broad sense" while a fully autonomous robot is "[a] role for a robot performing a given task in which the robot solves the task without human intervention [...]." The intention of the standard is to discriminate between the different ways the robot can act: autonomous, automated, tele-operated etc. In this view, the notion of robot that emerges is that of an agentive entity whose actual actions are limited by the role it plays. However, being in the "automated role" does not prevent an agent from acting autonomously like playing the "teacher role" does not prevent a person from making non educational acts. Most likely, the intention of the standard was to model the ontological notion of phase: being in an automated phase does prevent an agent from acting autonomously because the phase determines what the agent can do. The notion of phase is used in ontology to distinguish important changes in entities like the caterpillar and the butterfly. If we take the DNA perspective, these are the very same entity but the possible actions of the caterpillar are very different from those of the butterfly: which actions are possible is determined by the phase in which the entity is. Similarly, a robot in an autonomous phase has different possibilities than a tele-operated robot, for instance the latter does not make plans. The notion of role is not suitable to model this kind of change and we favour a revision of the standard in this sense.

Leaving aside these modeling choices, there are doubts whether robots should qualify as agentive entities in the strong sense since they lack intentional states. For most of today so-called robots even the qualification of robot in the weak sense seems unjustified since often they have only a conventional stimulus-response behavior. On the other side, we tend to distinguish a robotic arm

from a can opener: they are both artifacts [8] but we have a strong intuition that the second is a tool and the first a robot. Up to today, any attempt to draw the line between tools and robots has met important criticisms.

In this part of the paper, we propose an extension of the DOLCE ontology to include robots, robotic parts and tools. The goal of this extension is to start from the notions of artifact and of agent, as introduced in foundational ontologies, and to propose a way to discriminate among types of artifacts as needed to model industrial scenarios. Although we would like to achieve a wide characterization, in this paper our analysis is heavily influenced by the focus on robots used in industrial settings. Ontologically speaking, following the analysis in [11], a robot is an artifact: it is intentionally selected (via construction) and has attributed technical capacities. Technical capacities can vary considerably depending on the robots: they can be quite limited, like in ant robots, or flexible and multipurpose like in industrial or humanoids robots. Since we focus on industrial settings, thus on robotic arms, transportation modules and the like, the robots we aim to model are actually technological artifacts [8]: they are manufactured by following precise production plans and selected via dedicated quality tests. Thus, from the formal viewpoint we classify industrial robots as (technological) artifacts, i.e., elements of the ARTIFACT subcategory of NON-AGENTIVE PHYSICAL OBJECT [11].

The typical robots in the production scenarios are rational, reactive and may present some degree of autonomy. Today they are rarely adaptive and embedded although these are desirable features. They can also be proactive: they have goals, typically provided by the production system to which they belong, and can sometimes choose, or at least reschedule, plans to optimize their achievements. In short, these robots are artifacts whose behaviors resemble agents' behavior for the same goal(s). Since this behavior is expected from them, we propose to see a robot as an artifact whose attributed quality is to *behave agent-like*. Note that this modeling choice keeps agents and robots apart: a member of the latter group just mimics agents. The behavior can range from basic stimulus-response actions to activities controlled by sophisticated planning and goal adaptations, depending on what kind of agentivity the robot can behaviorally simulate. This is definitely acceptable for today's robots and it does not exclude that future generations of robots might be considered as full-fledge agents. Note that we will continue to talk informally about robots as agents in the rest of the paper.

Let us use ROBOT for the predicate 'being a robot' and BehSp for the generic space of behaviors. Using the language of DOLCE from [26,10,11], we can formally model the ontological status of robots as follows:

$$\mathtt{ROBOT(r)} \rightarrow \mathtt{ARTIFACT(r)} \quad (1)$$

$$\mathtt{ROBOT(r)} \wedge \mathtt{AttribCap(a)} \wedge \mathtt{qt(a,r)} \wedge \mathtt{ql(v,a,t)} \rightarrow \mathtt{Loc(v,BehSp)} \quad (2)$$

The first formula says that a robot is an artifact. The second states what distinguishes a robot from other artifacts: the capacity attributed to the robot (AttribCap(a) $\wedge$ qt(a, r)) has values (ql(v, a, t)) that belong to the space of behaviors (Loc(v, BehSp)).[1]

Robot's parts are themselves artifacts, thus elements of the ARTIFACT category. These are typically not robots, so their attributed qualities are of different types. The main distinction here is between the parts that are components, i.e., that constitute the robot like the engines that move the robotic arm structure and the structural pieces that are moved by the engines; and the parts that are tools used by the robot like the different types of gripper that can be substituted depending on the task to execute. These types of parts are isolated for their functional or structural contribution. Following DOLCE, we use part (of an artifact) as a generic term to indicate any arbitrary portion of the artifact. We call *component* any part of the artifact which is itself an artifact and whose behavior contributes to the behavior of the larger artifact. These parts, following [46], are also called functional parts. We also assume that components are persistent parts, i.e., they are always present in the artifact exceptions being typically limited to maintenance time. One could further distinguish structural vs operational components. In the first class there are elements like shafts, in the second devices like electric engines and sensors, that is, elements that can take some type of input and transform it into a different type of output (see the ontology of function in the next section). Note that these are not disjoint classes since an operational component can also be a structural

[1] The existence of quality a is enforced by formula (1) and the theory in [11]. The characterization of the space of behaviors is still under investigation.

component. However, there are structural components which are not operational like a shaft used to transfer torque. Finally, a *tool* is an artifact which can be a functional part of another artifact but is however not a necessary part for that artifact. In other words, a tool is an optional functional part of an artifact. Thus, the categories of tool and component are disjoint. For practical purposes, we require that a tool has been at least for some time part of an artifact. There are, of course, also arbitrary parts like the upper half of the skeletal frame, which do not have special properties or functionalities and thus are not relevant in terms of knowledge and planning. Components (tools) can be in an active/inactive (available/non-available, respectively) state for the robot. Sensors are listed among the components but note that we do not distinguish between sensors and actuators since these are seen as roles of the agent's components (a drill can play both of them at the same or at different times). Finally, an object that is a component is such until substituted (or dismantled) while a tool may remain such even if substituted. A richer classification of functional parts in genuine, replaceable, persistent and constituent is presented in [48].

By *environment* one usually understands an area (a location) and the elements in it. In the case of agents, this is the area of interest in which the agent could act. For artificial agents, the environment might also include the requirements and specifications about the software components and their development. Since we will deal with languages and software constraints in terms of contexts (see Section 3.4), the notion of environment that we use focuses on the notion of location. Given our aims, we start from a general understanding of the term. The choice has essentially two motivations: the notion of environment should be sufficiently broad to include at least the usual scenarios and possibly more, and it should be sufficiently flexible to make it reusable via specializations. Thus, at each point in time, we take the robot's environment to be the location where the robot is (it's area of movement) including the elements it contains plus the entities that, even though not in the location, can interact (positively or negatively) with the robot's activities and goals within a relevant temporal span.

This view is fairly general and assumes that the environment depends on the robot's features as well as on the features of other entities, possibly not in the vicinity of the robot. It is crucial to us to understand that the environment can change whenever the robot or its location or the entities there change. While our notion suffices for robots interacting wirelessly with other robots, we leave aside the characterization of environment in robots connected to internet since their environment is potentially much richer. In the case of production scenarios, the robot's environment can be identified with the collection of physical entities that are within a certain range from the robot (where the range may be bounded by physical barriers like floor, walls, ceiling, fences, etc.). As said, the environment is not necessarily limited to a precise region of space; it includes also entities with which the robot can interact via, say, radio communication and signaling in general. In ontological terms, the environment is a physical object obtained by the mereological sum of all the physical objects that are within the interaction range (workspace) of the robot. The location of the environment corresponds to the location of the objects in the environment plus the locations reachable by the robot itself.[2]

4.3. Ontology and Engineering of Functions

The classifications of the robots, the physical entities that may interact with them and their environments take care of the "static" part of our modeling problem. Since a robot is supposed to act to reach its goals, it must also have the conceptual machinery to know what it can do and how, thus to plan its actions. For this, reasoning on (engineering) functions is unavoidable. The formalization of functions in robotics is unfortunately rarely addressed and is too often confused with the notion of action, i.e., the performance of a function.

To overcome this problem, we extend the DOLCE ontology with an ontology of high-level functions. This function ontology is integrated, via DOLCE, with the ontology of the robot and robot's parts making possible to model what a robot can do and how. We adopt a notion of function-as-effect (Figure 4) which we adapt borrowing from well-known functional approaches in engineering design like the FOCUS/TX [22] (for the distinction "what to" vs. "how to" and the notion of behavior), the Functional Basis [27, 20] (for the idea of a function list), and the Function Representation [16] (for the distinction between environment-centric and device-centric function). The guiding idea is to make

[2] The location is fixed for robots like robotic arms, it is parametric (in particular, it may depend on the task) for mobile robots.

possible the identification of the high-level function (or sequence of functions) that needs to be executed to reach a given goal. For this, one explores the difference between the actual state and the desired state, and isolates the changes to be made. From this information, the robot can travel the taxonomy to identify the effects of the high-level functions and find a suitable combination.

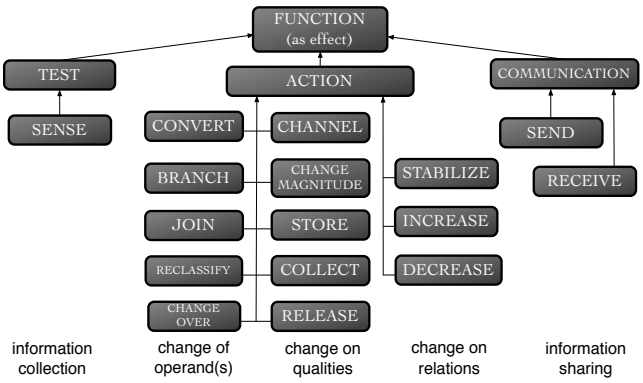

Figure 4 A (partial) ontological taxonomy of functions and its rationale

Figure 4 shows our top-level ontology of functions organized in 5 branches: functions to collect information, functions to change the operand(s) integrity, functions to change the operand's qualities, functions to change the quality relationships, and functions to share information. For instance, "reclassify" stands for the function to change the classification of an operand, e.g. when, after a test, a workpiece is classified as malfunctioning; "change-over" applies when, e.g., a robot acts on itself to activate/deactivate some component; "channel" stands for the moving of an operand (change of its location); "stabilize" for maintaining relational parameters like when tuning electronic components to regulate the input-output relationship; "sense" for the operand testing function, i.e., to acquire information without altering the status or the qualities of the operand; finally, "send" stands for the function to output information like a signal that a workpiece is going to be transferred or that a failure occurred.

Of course, this information is not enough since it would model just the *ideal* capacities of the robot. Aiming to have a robot adapting its plan at *run-time*, we have to model the actual capacities of the robot, which implies to take into account malfunctioning and/or missing parts or even deteriorated behaviors. This information depends on the capacities of self-inspection built-in in the robot as well as on the possibility to compare the "ideal actions" descriptions and the actual performances.

4.4. The Use of Contexts in Industry

As said, an ontology is a conceptual tool used to structure information. Ontologies deal mainly with necessary information like the properties that an object must manifest (shape, weight, mass etc.) or the types of event (states, actions, processes and so on). Factual information, being information that depends on contingent data (like spatio-temporal location, agent's setting, goals etc.), is generally characterized at the level of knowledge-bases. While this distinction might not be fully justified (and not even sharp), it remains important not to structure the ontology relying on factual knowledge. Unfortunately, this principle is rarely recognized in applications and in particular in the development of ontologies for industrial application.

Note that we insist on the distinction between necessary and contingent information only relatively to the development of the ontology structure: it is important that factual information finds its place in the factory information system. Our solution to this problem is to include factual information in the KB (built on top of the ontology) via contexts. This allows the system to classify and reason on factual information, for example, to understand the actual scenario and possible evolutions, to evaluate optimal production plans out of those that are actually possible, and even to establish the status of the resources or maintenance schedule. To act in real and evolving scenarios, factual information is thus essential. Contextualization allows us to manage it with an ontologically sound approach. Contextualization gives also an advantage at the reasoning level: it allows to differentiate types of information depending on their usefulness in reasoning on a situation or task.

After an ontological analysis based on [4,10,18], we identified three contextual models dedicated to factual knowledge, and use them on a par with the ontological framework. In particular, these contexts provide the time-dependent information needed to select how to execute high-level functions in the actual scenario. However, we do not exclude to expand the number of contextual models in the future since the use of factual classification is only partially explored at this stage of the work. Finally, note that our setting of the context framework makes possible to distinguish relevant consequences of the available knowledge depending on the type of the agent. The three context types are called global, local and internal, respectively. The *global*

context collects information the agent cannot control nor modify like the shared language of the system, the agents present in the system, the system's performance parameters. The *local* context collects information on the relationship between the agent and its neighbor elements (typically the human and artificial agents directly interacting with it), thus providing a local view of the topological setting. Finally, the *internal* context collects the information the agent has about itself as well as its capabilities toward itself (change-over) and toward the environment (communication and manipulation) [7].

We have so far described the KB we use for our artificial agents. We started from a foundational ontology which was already constructed from a cognitive stand and showed how we extended it to cover other kinds of knowledge, from artificial agents to functions. We also indicated how knowledge can be contextualized, thus allowing the KB to use a classification feature largely exploited by humans.

4.5. Applying Ontology and Contexts to the Manufacturing Case Study

Given a particular application like the manufacturing scenario of the case study [5], it is necessary to define the relevant knowledge for a KBCL process in order to dynamically infer the specific capabilities of an agent and adapt the control model accordingly. Thus, we have extended the DOLCE ontology including the type of needed information by applying the context-based analysis and the functional characterization described in the previous sections.

The extended ontology aims at characterizing the knowledge concerning the general *structure* of a TM part of the plant, the related *working environment* and the general *functional capabilities* of TMs. This information represents the general knowledge (i.e. the *TBox*) that a KBCL process instantiates according to the specific features of the TM to be controlled, in order to generate the specific KB of a particular TM (i.e. the *ABox*). Figure 5 shows the extension of the DOLCE taxonomy of particulars with respect to the NON-AGENTIVE PHYSICAL OBJECT category.

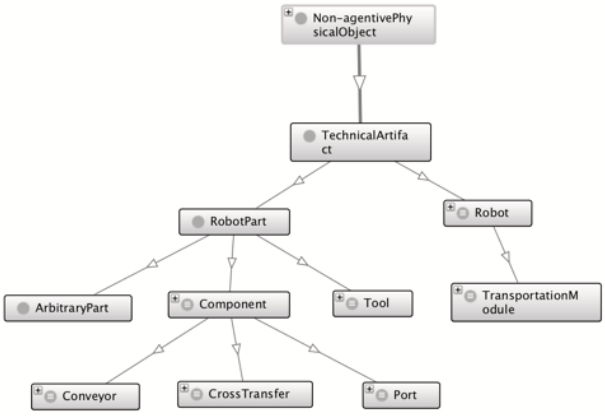

Figure 5 Extension of DOLCE ontology

According to our extension of the DOLCE ontology, robots and their parts are modeled as subcategories of TECHNICAL-ARTIFACT. As described in section 4.2, both robots and their components are artifacts but they differ in terms of the types of attributed qualities. Parts can be further distinguished into components, and parts that are only used by the robot to perform its tasks without constituting its structure i.e., tools. Thus, the set of elements belonging to the ROBOTPART category can be partitioned into elements belonging to the COMPONENT category and elements belonging to the ROBOTOOL category. Components have a *spatial location* within the robot structure (this would not be enforced for tools since they can be external to the robot), have an *active* state and *functional capabilities*. Let COMPONENT(x, y, t) mean that x is a component of y at time t and ROBOTOOL(x) mean that x is a tool, then we have:

$$\text{COMPONENT}(x, y, t) \rightarrow \text{ROBOTPART}(x, y, t) \wedge \text{Artifact}(x) \wedge \text{ROBOT}(y) \quad (3)$$

$$\text{COMPONENT}(x, r, t) \rightarrow \text{COMPONENT}(x) \quad (4)$$

$$\text{ROBOTOOL}(x) \rightarrow \text{Artifact}(x) \quad (5)$$

$$\text{ROBOTOOL}(x) \rightarrow \exists\, y,t\, (\text{ROBOT}(y) \wedge \text{ROBOTPART}(x, y, t)) \quad (6)$$

Thus, the general class axiom characterizing the COMPONENT category can be formally defined as follow:

$$\text{ROBOT}(r) \wedge \text{ROBOTPART}(p, r, t) \wedge \text{hasLoc}(p, l_p, t) \wedge \text{hasCapacity}(p, f) \wedge \text{hasOpStat}(p, \textit{active}, t) \rightarrow \text{COMPONENT}(p, r, t) \quad (7)$$

where ROBOTPART(p, r, t) is a predicate asserting that p is a part of a robot r and the formula states that when a part of a robot has a functional capacity (f) and is in *active* state, then the part must be a component of r.

In our case study, considering the *internal* structure of a TM, it is possible to define three different types of component: (i) the *conveyor component*; (ii) the *port component*; (iii) the *cross-transfer component*. Thus, we have extended the COMPONENT category by introducing the PORT category, the CONVEYOR category and the CROSS-TRANSFER category. The main distinction among the elements of these categories is the type of function the associated component can perform. Conveyors are the engines composing the structure of a TM that enable movements of pallets. These elements has *channel capabilities* that allow a TM to actually move pallets between two *adjacent spatial locations* connected through conveyors. The formula below formally characterizes the CONVEYOR category:

$$\text{ROBOT}(r) \wedge \text{ROBOTPART}(p, r) \wedge \text{hasLoc}(p, l_p, t) \wedge \text{hasCapacity}(p, f) \wedge \text{hasOpStat}(p, \textit{active}, t) \rightarrow \text{CHANNEL}(f) \rightarrow \text{CONVEYOR}(p) \quad (8)$$

where elements of the CHANNEL category represent functions whose *effect* changes the *spatial location* quality of an operand (i.e. a *pallet*). The execution of a *channel* changes the location of the pallet from the *start location* to the *end location*. Cross-transfers are another type of engines that allow a TM to change its *physical configuration*. These elements have *change-over capabilities* that change the *internal* connections of a TM. Different configurations (i.e., internal connections) enable different directions of movements of pallets within a TM (i.e., different paths). The CROSS-TRANSFER category can be formally defined as follows:

$$\text{ROBOT}(r) \wedge \text{ROBOTPART}(p, r, t) \wedge \text{hasLoc}(p, l_p, t) \wedge \text{hasCapacity}(p, f) \wedge \text{hasOpStat}(p, \textit{active}, t) \wedge \text{CHANGE-OVER}(f) \rightarrow \text{CROSS-TRANSFER}(p) \quad (9)$$

Finally, ports are structural elements that allow a TM to *be connected* with other TMs. These elements have *communication capacities* that allow a TM to *send* and *receive* pallets to and from other TMs of the plant. The PORT category can be formally defined as follow:

$$\text{ROBOT}(r) \wedge \text{ROBOTPART}(p, r, t) \wedge \text{hasLoc}(p, l_p, t) \wedge \text{hasCapacity}(p, f) \wedge \text{hasOpStat}(p, \textit{active}, t) \wedge \text{COMMMUNICATION}(f) \rightarrow \text{PORT}(p) \quad (10)$$

Channel functions can be combined together in order to realize *complex channels*. Components that collaborate to perform channel functions must be spatially connected. Thus, the *internal structure* for this kind of functionality is determined by the connections of the components' locations.[3] The choice of modeling the elements of a TM with different categories rather than using the general COMPONENT category relies on the different properties that these elements bring to implement *functional capabilities*. Their contribution to infer the functional capabilities of the TM will be discussed in Section 5.

The ROBOT category has been extended by introducing the TRANSPORTATION-MODULE category in a similar way to the ROBOTPART category. It characterizes TMs from a functional point of view. TMs are modeled as a particular type of robot capable of performing *channel* functions by collaborating with other *agents* (i.e., other TMs or working machines) of the plant. The collaborators of a TM constitute the *(working) environment* of a TM and therefore they are part of the local context of a TM. The general class axiom characterizing the elements that belong to the TRANSPORTATION-MODULE category can be formally defined as follows:

[3] In general, it suffices that the components have connected working areas. E.g. in a robot with a robotic arm and a container, the locations of the arm and the container are disconnected but the arm must be able to reach objects in the container to implement a Channel function, so the working areas must be connected.

ROBOT(r) ∧ ROBOT(c) ∧ hasCollab(r, c, t) ∧

CHANNEL(f) ∧ hasCapacity(r, f)

→ TRANSPORTATION-MODULE(r) (11)

It is important to point out that, in this context, a collaborator is a relative concept which depends on the particular *configuration* of the considered TM. It represents a relationship between a TM and other directly connected agents that could be either other TMs or working machines in the shop floor. Thus, the concept of COLLABORATOR is modeled as a role that an agent may play according to *local connections* (i.e. local context) of a TM.

5. CONNECTING KNOWLEDGE AND CONTROL

The Knowledge Manager module (KM) in Figure 2 is responsible for managing the lifecycle of the Knowledge Base (KB) within the KBCL process. In the specific context of the manufacturing case study, each KB models a particular TM of the transportation plant by specifying the associated internal structure, the connections with other TMs and the resulting functional capabilities. The management of such a KB relies on a *knowledge processing mechanism* implemented by means of a *Rule-based Inference Engine* which leverages a set of *inference rules* to generated and updated a KB of an agent.

The knowledge processing mechanism dynamically builds the KB by elaborating *raw data* received from the *Diagnosis Module* and infers knowledge concerning the structure, the *working environment* and the functional capabilities of the agent. This mechanism involves two reasoning steps, depicted in Figure 6, that are (i) the *low-level reasoning* step and (ii) the *high-level reasoning* step. Specifically, these two reasoning steps refine the knowledge about the agent by combining a set of dedicated *inference rules* with the general information of contexts and functions of the ontology described in the previous section.

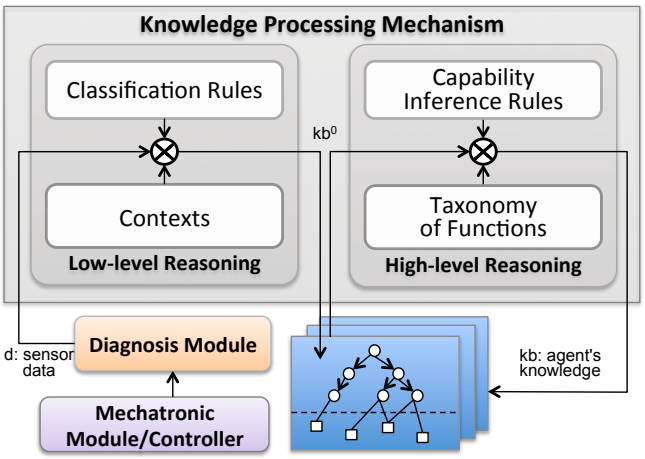

Figure 6 The knowledge processing mechanism

The first reasoning step, called the *low-level reasoning*, aims at characterizing a TM in terms of the components that actually compose its structure like e.g., ports, conveyors, etc., and the associated collaborators. It relies on the internal and local contexts of the ontology and a set of *classification rules*. The resulting KB, named kb^0 in Figure 6, characterizes the structure of the TM and the related *local working environment*. This initial KB describes the agent in terms of its internal and local contexts.

The second reasoning step, called *high-level reasoning*, starts from the KB elicited after the previous step (i.e., the kb^0) and further refines it by inferring knowledge about the functional capabilities of the TM. Specifically, the *high-level reasoning* step relies on the *taxonomy of functions* and the *capability inference rules* to complete the knowledge processing mechanism. The KB on which the *high-level reasoning* works encodes the particular internal and local context of the agent. Thus, the inference mechanism can determine the set of functions that the agent can actually perform by analyzing its structure and the associated *working environment*. The outcome of this second reasoning step (and the overall knowledge processing mechanism) is the *final* KB encoding a complete interpretation of the structure of the agent, the *working environment* (from the agent perspective) and the *functional capabilities* the agent can actually perform. Such knowledge is then exploited to generate the *plan-based control model* of the deliberative controller. The next two subsections provide a more detailed discussion of the two reasoning steps constituting the knowledge processing mechanism.

5.1. The low-level Reasoning

The *low-level reasoning process* is responsible for inferring information concerning the internal and local contexts of the TM. Namely, the result of this inference step is an initial KB describing the operating devices that compose the TM (i.e. the components) and the available collaborators. Namely, it builds an initial version of the KB by classifying data received from the *Diagnosis Module* on the basis of contexts categorization. The input data represents a set of *individuals* concerning the parts that compose the TM, their connections and their capabilities. Figure 7 provides a (partial) graphical representation of a possible set of individuals and predicates the knowledge processing mechanism receives as input from the *Diagnosis Module*. In particular, the figure shows the different contexts the individuals belong to, the reasoning step exploits to provide these data with additional semantics.

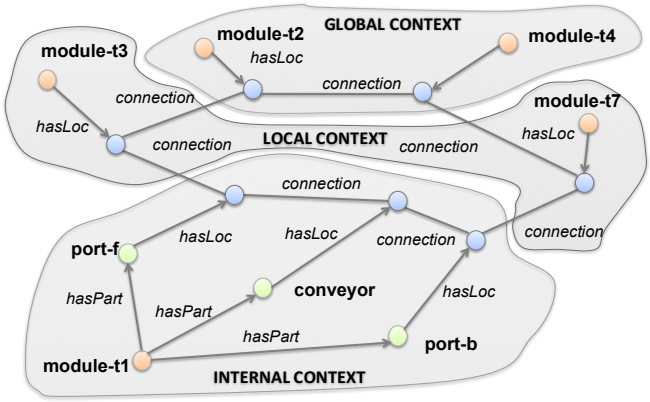

Figure 7 Raw data received from the Diagnosis Module

Given this set of data, the first rule applied by the *low-level reasoning* aims at identifying the set of active parts the TM can actually use to perform functions. These set of active parts are represented as TM's *components*. According to axiom (7), an element of the COMPONENT category is a structural part of a robot, it has an *operative state* and some *functional capabilities*. The *inference process* exploits this functional characterization of components to interpret input data and identify the components of the TM. According to the axiom (7), being p a structural part of a robot r, with the capability of performing some function f, it is possible to *infer* that p is an element of the COMPONENT category. Therefore, due to axioms (4), the predicate COMPONENT(p) is true.

The applied ontological approach models the different types of components that may compose a TM (see Figure 5) according to the different types of functional capabilities. Leveraging this interpretation, the *low-level reasoning process* applies axioms (8), (9) and (10) to classify inferred components and identify the different types of component that actually constitute the TM. As soon as the components are classified, the *low-level reasoning process* ends by inferring the set of available collaborators of the TM. Collaborators are TMs of the plant that are in *operative state* and are directly connected to the TM. Namely, collaborators are TMs of the plant that can actually *receive*/send pallets *from*/*to* TM. This, collaborators are inferred by applying the following rule:

$$\begin{aligned} &\mathrm{ROBOT}(r) \wedge \mathrm{PORT}(p) \wedge \mathrm{hasLoc}(p, l_p, t) \wedge \\ &\mathrm{ROBOTPART}(p, r, t) \wedge \mathrm{hasOpStat}(p, \mathrm{active}, t) \wedge \\ &\mathrm{ROBOT}(c) \wedge \mathrm{hasLoc}(c, l_c, t) \wedge \mathrm{connection}(l_p, l_c, t) \\ &\rightarrow \mathrm{hasCollab}(r, c, t) \end{aligned} \quad (12)$$

where connection(l_p, l_c, t) is a predicate asserting that the location of the TM's port p is connected with the robot c. Figure 8 provides a (simplified) graphical representation of a possible KB resulting from the applications of rule (12) (the dotted arrows represent the inferred properties concerning collaborations).

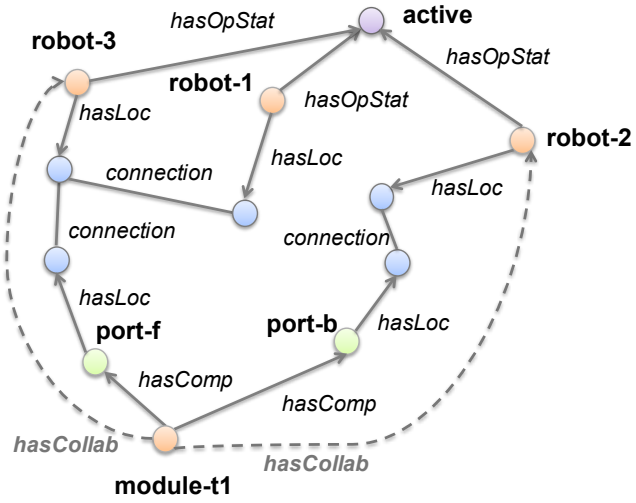

Figure 8 Inferring collaborators of a TM

5.2. The high-level Reasoning

The *high-level* reasoning step extends the KB elicited from the previous step inferring the capabilities the TM is actually able to use on the basis of its current status and current production environment.

Given the information about the components and collaborators of a TM, the first rule applied by the *high-level reasoning* aims at inferring the *primitive channels* the TM can perform according to its internal structure. Indeed, active components can be used by a robot to perform functions. For instance, a conveyor allows a TM to perform channel functions. According to this interpretation it is possible to define a rule to infer the set of *primitive channels* a TM can perform as follows:

$$\begin{aligned}&\mathtt{ROBOT}(r) \wedge \mathtt{CONVEYOR}(c_1) \wedge \\ &\mathtt{hasOpStat}(c_1, \mathit{active}, t) \wedge \mathtt{COMPONENT}(c_2) \wedge \\ &\mathtt{COMPONENT}(c_3) \wedge \mathtt{hasLoc}(c_1, l_1, t) \wedge \\ &\mathtt{hasLoc}(c_2, l_2, t) \wedge \mathtt{hasLoc}(c_3, l_3, t) \wedge \\ &\mathtt{connection}(l_2, l_1, t) \wedge \mathtt{connection}(l_1, l_3, t) \wedge \\ &\rightarrow \mathtt{hasCapacity}(r, f) \wedge \mathtt{CHANNEL}(f) \wedge \\ &\mathtt{cStart}(f, l_2) \wedge \mathtt{cEnd}(f, l_3) \wedge \mathtt{cConnect}(l_2, l_3)\end{aligned} \quad (13)$$

where $(\mathtt{CONVEYOR}(c_1) \wedge \mathtt{hasOpStat}(c_1, \mathit{active}, t))$ asserts that c_1 is a conveyor component of the TM whose operative state is *active* at time t. Namely, the conveyor c_1 is an active component of the TM and therefore, can be actually used to perform functions. Figure 9 shows a (simplified) graphical representation of the KB resulting from the application of rule (13). In particular, the figure represents predicates (i.e. the dotted arrows) and the individual (the channel-1 node) inferred and added to the KB.

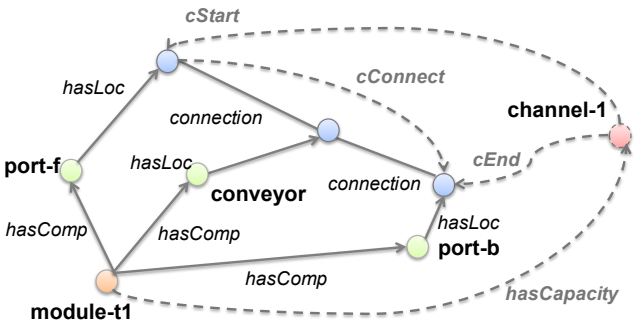

Figure 9 Inferring primitive channels of a TM

The rationale of rule (13) relies on the *functional interpretation* of the CONVEYOR category as the set of components that *can perform channel functions*. Thus, if a conveyor component connects two components of the TM through its *spatial location*, see the clause $(\mathtt{connection}(l_2, l_1, t) \wedge \mathtt{connection}(l_1, l_3, t))$ in (13), then the conveyor can perform a *primitive channel* function between the components' locations. Moreover, the $\mathtt{cConnect}(l_2, l_3)$ is a transitive predicate which allows to *connect* different channel functions. Indeed, if two spatial locations are connected through the cConnect predicate then there exists a composition of *primitive channel functions* that "connect" them.

A *primitive channel* involves components of one TM. However, the knowledge process mechanism aims to infer the channel capabilities that involve the collaborators of a TM. Namely, *channel functions* that allow a TM to exchange pallets with other TMs of the plant. We call such functions *complex channel* and they are inferred by applying the following rule:

$$\begin{aligned}&\mathtt{ROBOT}(r) \wedge \mathtt{ROBOT}(rc_1) \wedge \mathtt{ROBOT}(rc_2) \wedge \\ &\mathtt{hasCollab}(r, rc_1, t) \wedge \mathtt{hasLoc}(rc_1, rl_1, t) \wedge \\ &\mathtt{hasCollab}(r, rc_2, t) \wedge \mathtt{hasLoc}(rc_2, rl_2, t) \wedge \\ &\mathtt{PORT}(c_1) \wedge \mathtt{hasOpState}(c_1, \mathit{active}, t) \wedge \\ &\mathtt{hasLoc}(c_1, l_1, t) \wedge \mathtt{PORT}(c_2) \wedge \\ &\mathtt{hasOpState}(c_2, \mathit{active}, t) \wedge \mathtt{hasLoc}(c_2, l_2, t) \wedge \\ &\mathtt{connection}(l_1, rl_1, t) \wedge \mathtt{connection}(l_2, rl_2, t) \wedge \\ &\mathtt{cConnect}(l_1, l_2) \rightarrow \mathtt{hasCapacity}(r, f) \wedge \\ &\mathtt{CHANNEL}(f) \wedge \mathtt{cStart}(f, rl_1) \wedge \mathtt{cEnd}(f, rl_2)\end{aligned} \quad (14)$$

A key point of the rule (14) is that a *complex channel function* is interpreted as the composition of some primitive channels a TM can internally perform. This is a quite *flexible* and general interpretation of a *channel function*. If one or more parts of a TM stop working (i.e., their operational state passes to *inactive*), then a TM will no longer be able to perform the associated *primitive channels* and the *high-level reasoning* step will not be able to infer the associated *complex channel functions* that depend from these parts. Similarly, if new components are added to the TM, the *high-level reasoning* step will be able to infer additional *complex channel functions* according to the resulting structure. Finally, note that we do not add the converse formulas of (13) and (14) since these would prevent the discovery of alternative ways to perform the functions.

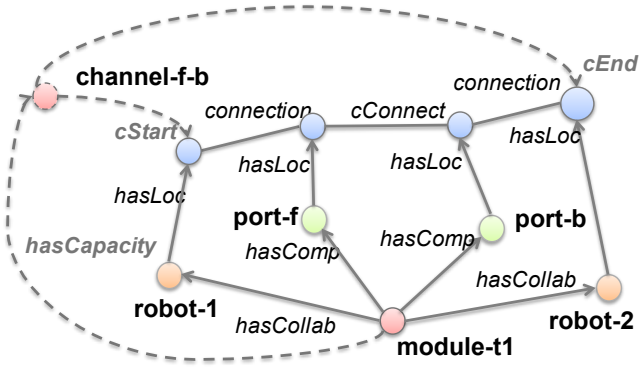

Figure 10 Inferring *complex* channels of a TM

5.3. Implementation notes

Although the ontology is provided in first order logic, at this stage only preprocessing (primarily to ensure conceptual consistency) is done in that language. Most of the inferences at runtime are done in the Web Ontology Language (OWL) version of the KB where we exploit primarily the contextual classification and relationships. The ontology editor Protégé[4] has been used for KB design and testing. For runtime reasoning in the Knowledge Manager, we have used the Ontology and RDF APIs and Inference API provided by the Apache Jena Software Library[5]. Finally, the *Deliberative Controller* has been realized by means of the GOAC architecture [14] whose deliberative features are implemented by means of APSI-TRF [15]. A more detailed description of how the KBCL has been implemented goes beyond the scope of this paper.

6. PLAN-BASED CONTROL LOOP

The *Planning Framework* element in Figure 2 endows the KBCL process with deliberative capabilities by exploiting a timeline-based planner [36]. The planner relies on a timeline-based planning model automatically generated from KB's information. Before describing the details of the process which generates the planning domain, this section provides a brief description of timeline-based planning and the pursued modeling approach.

6.1. Planning with Timelines

The timeline-based approach to planning has been introduced in early 90s [37] and takes inspiration from the classical control theory. It models a complex system by identifying a set of relevant features that must be controlled over time. This approach has been successfully applied to real world contexts (especially in space applications) and several planning frameworks have been developed for the synthesis of timeline-based P&S applications, e.g. EUROPA [38], ASPEN [39], APSI-TRF [40].

Broadly speaking, timeline-based planning applications aim at controlling a complex system by synthesizing temporal behaviors of its features in shape of timelines. A timeline consists of a sequence of states/actions the related domain feature (e.g., a component of the device to control) may assume/perform over time. Every value on a timeline is temporally allocated and represents the value/action the feature assumes/performs during the related temporal interval. *Temporal flexibility* allows to allocate values to flexible temporal intervals, i.e., intervals with flexible start time and end time. The resulting timeline represents an envelope of possible temporal evolutions of the related feature. Thus, a timeline-based plan, which consists of the union of all the timelines of the domain, represents the sets of all the possible temporal evolutions of the domain features. It is important to point out that the temporal flexibility in such a plan can be exploited at execution time by an executive system to gain robustness [41].

6.1.1. Modeling approach

State Variables model the features of a system that must be controlled over time. A state variable describes the temporal behaviors of a specific feature by means of causal and temporal constraints. More specifically, it describes the values the feature can assume over time, their duration constraints and the allowed transitions. In this regard, starting from the ontological analysis of the functional capabilities and the structure of agents described above, we define a modeling methodology of timeline-based planning domains.

The key idea is that the planning domain is to describe the functional capabilities of the system we want to control, the features of the elements that compose the system and the features of the working environment that must be taken into account in order to successfully carry out the desired functionalities. From the control perspective, it is possible to identify three different classes of state variables: (i) *functional* state variables; (ii) *primitive* state variables; (iii) *external* state variables. *Functional state variables* model a physical system as a whole in terms of the high-level functions it can perform (notwithstanding its internal structure). *Primitive state variables* model the physical and/or

[4] http://protege.stanford.edu

[5] http://jena.apache.org

logical elements that compose a physical system. In particular, these state variables model the elements we must actually control to execute high-level functions. *External state variables* model elements of the domain whose behavior is not directly under the control of the system. For example, these variables model conditions that must hold in order to successfully perform operations..

The behavior of state variables must be further constrained by specifying inter-component causal and temporal requirements, called *synchronization rules*. These rules specify additional constraints that allow to coordinate the behaviors of the domain features in order to perform high-level functions (i.e., planning goals). Following a hierarchical approach, synchronization rules map the values of functional state variables into a set of constraints among the values of primitive and/or external state variables that guarantee the proper functioning of the overall system and its elements. Namely, synchronization rules specify how high-level functions are implemented by an agent. These rules describe dependencies between the different variables of a planning domain and therefore may determine a hierarchy among them. A comprehensive formalization of timeline-based planning is provided in [47].

6.2. The model generation process

Key role for the dialogue between the Knowledge Manager and the Deliberative Controller is the Model Generation process (Step 2 of Figure 2). The KB generated by the Knowledge Manager provides an abstract representation of the capabilities, the structure and production environment of an agent. The Model Generation process analyzes such a KB in order to dynamically generate a timeline-based planning domain for the Deliberative Controller.

The process encodes the hierarchical modeling methodology described in the previous section and builds the model by leveraging the context-based characterization of the KB. The information concerning the *global context* and the *taxonomy of function* allow to define the functional state variables that provide a *functional view* of the agent as a whole. These state variables indeed describe the high-level tasks the agent can perform over time.

The *internal context* contains structural information about the agent and therefore it is suited to generate the *primitive state variables* of the domain. These variables describe the physical/logical features that compose the agent. Usually, the values of this type of variables directly correspond to states or actions that may assumed/performed over time by the related feature.

The *local context* manages information concerning the *working environment* of the agent and therefore it is suited to build the set of *external state variables* of the model. These variables model the *collaborating agents* (e.g., the directly connected TMs of the plant in the case study) whose behavior may affect the capabilities of the agent, even if not directly controllable.

```
1: function BUILDCONTROLMODEL(KB)
2:     // extract agent's information and initialize the P&S model
3:     agent ← getAgentInformation(KB)
4:     model ← inititalize(KB, agent)
5:     // define components of the model
6:     svs ← buildFunctionalComponents(KB, agent)
7:     svs ← buildPrimitiveComponents(KB, agent)
8:     svs ← buildExternalComponents(KB, agent)
9:     // build the set of task decomposition rules
10:    S ← buildSynchronizationRules(KB, agent)
11:    // update the P&S model
12:    model ← update(model, svs, S)
13:    return model
14: end function
```

Figure 11 The model generation procedure

The Algorithm in Figure 11 describes the general procedure of the model generation process. The procedure consists of four specific sub-procedures that analyze different areas of the knowledge about the agent in order to generate different parts of the control model. The procedure starts by extracting information related to the agent and initializing the P&S model (rows 3-4). According to the hierarchical approach described above, a set of *functional*, *primitive* and *external* state variables is generated (rows 6-8). Finally, the hierarchical decomposition of functional values (i.e., the values of the functional state variables) is described by means of a suitable set of generated synchronization rules (row 10). The resulting timeline-based model is then composed and returned as the outcome of the procedure. (rows 12-13).

Then, the *buildControlModel* procedure allows the model generation process to automatically build the timeline-based specification by leveraging the knowledge about the agent. As described in [42], every time a change occurs in the KB, a new instance of the *model generation process* is triggered in order to generate an updated control model of the agent. The next subsections provide some details about the sub-procedures of the process as well as an example of a possible timeline-based control model that can be generated for a TM in the case study plant.

6.2.1. Building State Variables from Contexts

The *functional state variable* generation procedure creates a set of state variables concerning the functional capabilities of the agent. The procedure relies on the set of *capabilities* the *knowledge processing mechanism* has inferred through application of rules (13) and (14). The procedure generates a state variable for each function of the taxonomy (see Figure 4) the agent can perform. Namely, given a particular function *f* of the taxonomy, if the KB contains at least one *individual* for that function *f* (i.e., if the *knowledge processing mechanism* has inferred at least one way for the agent to perform *f*), then a state variable *sv* for *f* is created. The *individuals* of *f* in the KB represent all the possible implementations of *f* that the agent can perform (i.e., all the *capabilities* of the agent with respect to *f*). Thus, for each *inferred* individual of *f* the procedure adds a value to the related (functional) state variable *sv*.

```
1: function BUILDFUNCTIONALCOMPONENTS(KB, agent)
2:     // initialize the list of functional variables
3:     svs ← ∅
4:     // get types of functions according to the Taxonomy in the KB
5:     taxonomy ← getTaxonomyOfFunctions(KB)
6:     for all function ∈ taxonomy do
7:         // check if the KB contains individuals of function
8:         capabilities ← getCapabilities(KB, agent, function)
9:         if ¬ IsEmpty(capabilities) then
10:            // create functional variable
11:            sv ← createFunctionalVariable(function)
12:            // add a value for each "inferred" capability
13:            for all capability ∈ capabilities do
14:                sv ← addValue(sv, capability)
15:            end for
16:            // add created state variable
17:            svs ← addVariable(svs, sv)
18:        end if
19:    end for
20:    return svs
21: end function
```

Figure 12 The functional variable generation procedure

The Algorithm in Figure 12 shows the pseudo-code of the *buildFunctionalComponents* procedure. The procedure first initializes the set of functional state variables of the domain (row 3). Then, the procedure reads the taxonomy of function from the KB and, for each function, checks the available capabilities of the agent (rows 6-20). Given a function, if the KB contains at least one capability for that function, then the procedure creates a functional state variable (rows 9-11). Each capability found in the KB is modeled as a value of the state created variable (rows 12-15). The procedure ends by returning the set of obtained variables.

The *primitive state variable* generation procedure creates a set of state variables concerning the structural components of the agent. The procedure relies on a *functional interpretation* of components as elements that allow the agent to perform functions. The procedure creates a *primitive state variable* to the model for each component found in the KB. According to the rules (8), (9) and (10) the components of the agent are modeled in terms of their *capabilities*. Thus, the values of these state variables represent the *primitive functions* of the agent.

```
1: function BUILDPRIMITIVECOMPONENTS(KB, agent)
2:     svs ← ∅
3:     // get agent's operative components
4:     components ← getActiveComponents(KB, agent)
5:     for all component ∈ components do
6:         // check if component can perform some functions
7:         capabilities ← getCapabilities(KB, component)
8:         if ¬ IsEmpty(capabilities) then
9:             // create primitive variable for component
10:            sv ← createPrimitiveVariable(component)
11:            // check component's functional capabilities
12:            for all capability ∈ capabilities do
13:                sv ← addValue(sv, function)
14:            end for
15:            svs ← addVariable(svs, sv)
16:        end if
17:    end for
18:    return svs
19: end function
```

Figure 13 The primitive variable generation procedure

The Algorithm in Figure 13 shows the pseudo-code of the *buildPrimitiveComponents* procedure. The procedure first initializes the set of primitive state variables of the domain (row 2). Then, the procedure reads the set of *inferred components* from the KB (row 4). Given a component, if the KB contains at least one primitive function the agent can perform through that component, then a *primitive variable* is created (rows 5-10). The values added to the variable model the capabilities of the related component. Namely, the values model all the primitive functions the agent can perform by means of the considered component (rows 11-16). The procedure ends by returning the set of generated state variables. The *external state variable* generation procedure creates the set of external variables composing the timeline-based model. The procedure generates the set of state variables representing the *collaborators* of the agent. Specifically, a state variable is created for each *individual* found in the KB that, according to the inference rule (12), has been classified as *collaborator*. The values of these state variables represent the *operative states* the collaborators may assume over time.

```
1: function BUILDEXTERNALCOMPONENTS(KB, agent)
2:     svs ← ∅
3:     // get agent's collaborators
4:     collaborators ← getCollaborators(KB, agent)
5:     for all collaborator ∈ collaborators do
6:         // create an external variable to model the collaborator
7:         sv ← createExternalVariable(collaborator)
8:         // model the possible behaviors of collaborators
9:         states ← getOperativeStates(collaborator)
10:        for all state ∈ states do
11:            sv ← addValue(sv, state)
12:        end for
13:        svs ← addVariable(svs, sv)
14:    end for
15:    return svs
16: end function
```

Figure 14 The external variable generation procedure

The Algorithm in Figure 14 shows the pseudo-code of the *buildExternalComponents* procedure in Figure 11. The procedure first initializes the set of external variables of the domain (row 2). Then, the procedure reads the set of inferred collaborators from the KB (row 4). For each collaborator found, a state variable is created (rows 5-7) and for each operative state the collaborator may assume over time, a value is added to the created variable (rows 9-14). The procedure ends by returning the set of generated variables.

6.2.2. Building Decomposition Rules from Inference Trace

When all the state variables and their values have been generated, it is necessary to build the *synchronization rules* of the domain in order to coordinate the behaviors of the different components of the agent and achieve the desired goals. Thus, given the general procedure in Figure 11, the *buildSynchronizationRules* procedure generates the decomposition rules by leveraging the *inference trace* of the KB.

The *inference trace* represents *internal knowledge* generated by the application of the inference rules. Such knowledge manages *intermediate information* which is necessary to complete the *knowledge processing mechanism* and therefore build the KB. For instance, besides *primitive channels*, the inference rule (13) generates *cConnect* properties. These properties do not represent specific information about the agent but are necessary to generate the set of *complex channels*, as shown in rule (14). These properties encode functional dependencies among the components of a TM. In particular, they encode these dependencies in terms of *primitive channels* needed to *implement complex channels*.

The inferred *cConnect* properties can be analyzed in order to build a particular data structure, called *functional graph*, that correlates functional dependencies among components, primitive and complex channels. The graph is built according to the inferred *cConnect* properties. Thus, the possible *implementations* of complex channels can be found by traversing the functional graph. This set of information is necessary to build the set of synchronization rules specifying how the agent must *execute* complex channels. Indeed, synchronization rules are generated by analyzing the paths on the functional graph that connect the *start* with the *end* locations of complex channels. These paths can be easily expressed in terms of *precedence constraints* between primitive channels of the involved components.

```
1: function BUILDSYNCHRONIZATIONRULES(KB, agent)
2:     rules ← ∅
3:     // create the functional graph for channel functions
4:     graph ← buildChannelFunctionalGraph(KB, agent)
5:     // get inferred complex channels
6:     channels ← getChannels(KB, agent)
7:     for all channel ∈ channels do
8:         // get available implementations
9:         implementations ← getImplementation(graph, channel)
10:        for all implementation ∈ implementations do
11:            // create synchronization rule from implementation
12:            rule ← createSynchronizationRule(KB, implementation)
13:            rules ← addRule(rule)
14:        end for
15:    end for
16:    return rules
17: end function
```

Figure 15 The Synchronization rule generation procedure

The Algorithm in Figure 15 shows the pseudo-code of the *buildSynchronizationRules* procedure. The procedure first initializes the set of rules (row 2) and then analyzes the KB to build the *functional graph* concerning channel functions (row 4). For each complex channel the procedure extracts the available implementations from the functional graph (rows 6-9). Each implementation encodes a set of temporal constraints between the primitive channels of the agent. Thus, given a possible implementation of a complex channel, a new synchronization rule is created (rows 10-14). The procedure ends by returning the set of generated synchronizations.

6.2.3. The Resulting Timeline-based Control Model

The described procedures encode a model generation process which relies on a context-based characterization of the KB. According to this structure, the process generates a *hierarchical domain description* modeling the complex functions of the agent in terms of primitive functions internal components can directly handle. Figure 16 shows a partial timeline-based model generated for a TM which is endowed with a single cross-transfer unit. The model provides a functional characterization of the TM according to the functional, primitive and external hierarchical levels.

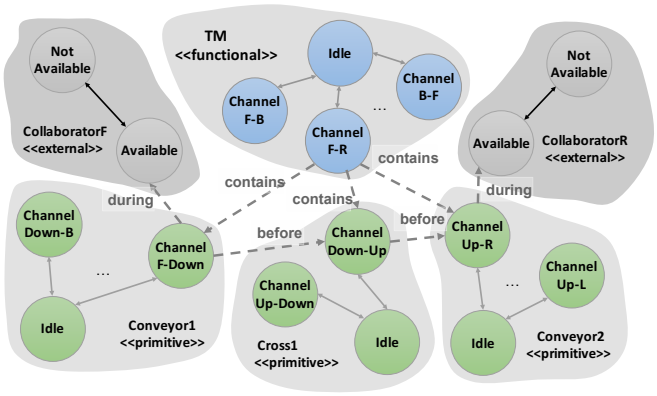

Figure 16 A (partial) timeline-based model generated for a TM equipped with one cross-transfer unit

The primitive state variables (the green ones in Figure 16) model the *active* parts of the TM that can actually perform some (primitive) functions. These state variables model the functional capabilities of the elements that compose the TM. For example, the component *Conveyor1* can perform the primitive channel *ChannelF-Down* to move a pallet between the location of component *PortF* and the location *Down* of component *Cross1*. Similarly, the component *Cross1* can perform the primitive channel *ChannelDown-Up* to move a pallet from the location *Down* to the location *Up* of the same component *Cross1*.

The external state variables (the grey ones in Figure 16) model the inferred collaborators that can directly interact with the considered TM. The values of these variables represent the operative states that collaborators may assume over time. Figure 16 shows the external state variables concerning two of four collaborators available. Specifically, the state variables model the temporal behaviors of *CollaboratorF* and *CollaboratorR i.e.*, the collaborators connected to the TM through the components *PortF* and *PortR* respectively.

The functional state variables (the blue ones in Figure 16) model the inferred channel functions the TM can perform by combining the internal (i.e., the primitive) channel functions. For example, according to this interpretation, *ChannelF-R* can be seen as the composition of the following primitive channels: *ChannelF-Down, ChannelDown-Up, ChannelUp-R.* Such a composition represents a particular *implementation* of the complex channel function *ChannelF-R*. Implementations are modeled by means of synchronization rules that specify a suited set of *temporal constraints* (see red arrows in Figure 16). These temporal constraints encode also the functional dependencies between the TM and its collaborators. Indeed, *CollaboratorF* and *CollaboratorR* must be available (i.e., operative) *during* the execution of the complex channel function *ChannelF-R.*

The generated timeline-based planning model provides a functional characterization of TMs of the plant where planning goals represent high-level functions a TM can perform. These functions are described in terms of the *atomic* operations (i.e., primitive functions) a TM is able of performing by means of its components and the available collaborators.

7. EXPERIMENTAL EVALUATION

In order to validate the architecture and the Knowledge-based Control Loop described above, the whole system has been deployed in the manufacturing case study.

A set of tests has been run for the KBCL with different TM configurations. Specifically, all the different physical configurations of a TM have been considered, i.e., from zero to three cross-transfer modules, referring to them as *simple*, *single*, *double* and *full* configuration, respectively. A different configuration also entails a different number of connected TM neighbors. Clearly, the more complex scenario is the one with the highest number of both cross-transfers (the full configuration) and neighbors. Moreover, reconfiguration scenarios have been addressed considering different external events, i.e., an increasing number of TM neighbors momentarily unable to exchange pallets, or internal failures, i.e., a cross-transfer engine failure or a local failure for a specific port.

The experiments were carried out to evaluate the performance of the following aspects of a single TM agent: (i) the knowledge processing mechanism; (ii) the planning model generation; (iii) the synthesis of plans to manage a set of pallet requests. The final aim is to show that the latency of the KBCL is compatible with execution latencies of the RMS. Figure 17 shows the timings[6] in the Setup phase for the KBCL module operation, i.e., to build the KB exploiting the classification and capability inference process, and to generate the timeline-based planning specification for the TM. On the one hand, the results show that an

[6] All the experiments have been performed on a workstation endowed with an Intel Core2 Duo 2.26GHz and 8GB RAM.

increase in the complexity of the TM configurations does not entail a performance degeneration of the knowledge processing mechanism: the inference costs are almost constant (around 1.3 secs). This behavior was expected since the number of instances/relationships in the KB is rather low independently of the physical configuration of the TM; thus, the performance of the inference engine deployed here is not particularly affected by this aspect. On the other hand, the model generation is linearly affected by the increasing complexity, spanning from 0.8 secs in the simple configuration, up to a maximum of 2.2 seconds in the full configuration.

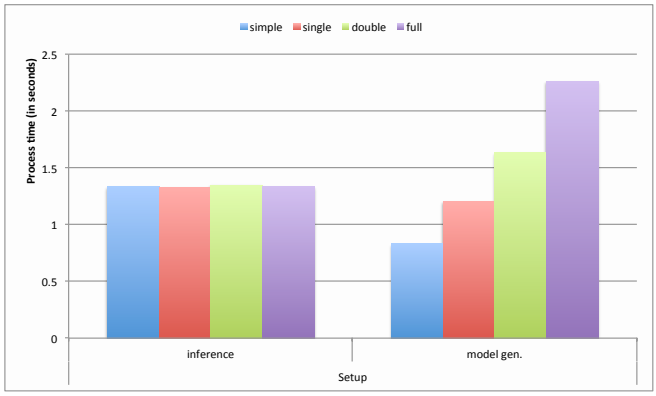

Figure 17 KB inference and planning domain generation times

The model generation process entails a combinatorial effect on the number of instances/relationships needed to generate components and synchronizations leading to larger planning models and, thus, to higher process costs. When a reconfiguration scenario occurs, the knowledge processing costs are negligible. Among all the considered reconfiguration cases, the time spent by the knowledge processing mechanism to (re)infer the enabled functionalities is just a few milliseconds. In fact, both the classification and capability inference steps are applied to a KB only slightly changed after the reconfiguration. The small changes in terms of functionalities can be quickly inferred in the system and represented in the current KB. As for the planning model generation cost, the considered reconfiguration scenarios (either external or internal) lead to a reduction of functionalities and the related costs are relatively small. For instance, in the case of the full TM configuration, the cost for the model generation is always below 0.8 seconds.

Finally, we evaluate the planning costs when facing both setup and reconfiguration scenarios with an increasing number of pallet requests (randomly generated in the specific case), i.e., planning goals, to be fulfilled. Planning costs span from few seconds up to nearly 30 seconds when planning for 10 pallet requests within a 15 minutes' time horizon. In general, the more complex the planning model, the harder the plan synthesis problem. Thus, the planning costs follow the complexity of the configurations of the specific TM agent.

7.1. Discussion

The experimental results show the practical feasibility of the KBCL approach in increasingly complex instances of a real-world manufacturing case study. The collected data for the initialization (or the update) of a generic agent's KB (considering both knowledge processing and model generation) and the cost for planning synthesis have a rather low impact on its performance during operation. In fact, in order to face production periods of 15 minutes –and the management of 10 pallet requests– no more than 5 seconds are required by the Knowledge Manager while less than 30 seconds are required by the Planner to generate a suitable plan. These performances are compatible with the system usual latency in this type of manufacturing applications [13]. It is worth reminding how the role of the KBCL is to avoid major overhauls of the control policies (e.g., control code revisions deployed too often in concrete cases) to cope with adaptation to variations or plant reconfigurations.

8. CONCLUSIONS

In most architectures for artificial agents the knowledge of the world is distributed across different parts of the system. This may lead to information redundancy (jeopardizing consistency), planning-knowledge misalignment, inconsistent knowledge update, etc. We have proposed to use a single foundational ontology to coherently organize knowledge on the entities and relationships in the agent's world (objects, functionalities, states including the agent and its components) augmented with context modules that store the factual knowledge of the agent on these entities. We then have described how the knowledge manager extracts information for the planning and execution module of the agent. With this setting, the agent can reason on its own capabilities and adapt its plans in case of failures or sudden changes in the world. To validate the approach, we provided an implementation of the knowledge control loop and showed that it has performances compatible with the settings of a realistic industrial scenario. The validation is promising but admittedly limited: a full

implementation in a large industrial scenario is needed to compare our work with other approaches.

Several issues remain to be explored. First, it is important to have a fairly amount of common knowledge on the environment to take advantage of the knowledge in the ontology and the context modules. It is also unclear how to optimize reasoning by taking advantage of the expressivity of the available languages (first-order logic and OWL), since reasoning in languages like OWL leads to ignore basic information about, e.g., activity ordering and functionality constraints. Second, we have to expand the connection between the ontology of functions and the constraints on the agents' state variables: these constraints are crucial to execute the functions and control the changes in the environment. Finally, our verification scenario is still quite limited.

ACKNOWLEDGMENTS

CNR authors are supported by MIUR/CNR within the GECKO Project - Progetto Bandiera "La Fabbrica del Futuro".

REFERENCES

[1] Yazen Al-Safi and Valeriy Vyatkin (2007). An ontology-based reconfiguration agent for intelligent mechatronic systems. In Vladimr Malik, Valeriy Vyatkin, and ArmandoW. Colombo, editors, Holonic and Multi-Agent Systems for Manufacturing, volume 4659 of Lecture Notes in Computer Science, pages 114–126. Springer Berlin Heidelberg.

[2] Stephen Balakirsky (2015). Ontology based action planning and verification for agile manufacturing. Robotics and Computer-Integrated Manufacturing, 33(0):21–28. Special Issue on Knowledge Driven Robotics and Manufacturing.

[3] Gregor Behnke, Denis Ponomaryov, Marvin Schiller, Pascal Bercher, Florian Nothdurft, Birte Glimm, and Susanne Biundo (2015). Coherence across components in cognitive systems – one ontology to rule them all. In Proc. of the 25th Int. Joint Conf. on Artificial Intelligence (IJCAI 2015). AAAI Press.

[4] S. Borgo. How Formal Ontology can help Civil Engineers. In J. Teller, J. Lee, and C. Roussey, editors, Ontologies for Urban Development, pages 37–45. Springer, Berlin, 2007.

[5] Borgo S., Cesta A., Orlandini A., Rasconi R., Suriano M., Umbrico A. (2014) Towards a cooperative knowledge-based control architecture for a reconfigurable manufacturing plant. In 19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2014). IEEE, 2014.

[6] Stefano Borgo. An ontological approach for reliable data integration in the industrial domain. Computers in Industry, 65(9):1242–1252, 2014.

[7] Borgo S., Cesta A., Orlandini A., and Umbrico A. (2015) An ontology-based domain representation for plan-based controllers in a reconfigurable manufacturing system. In Proc. of the Twenty-Eighth International Florida Artificial Intelligence Research Society Conference, pages 354–359. AAAI Press.

[8] Stefano Borgo, Maarten Franssen, Pawel Garbacz, Yoshinobu Kitamura, Riichiro Mizoguchi, and Pieter E. Vermaas. Technical artifacts: An integrated perspective. Applied Ontology Journal, 9(3-4):217–235, 2014.

[9] Stefano Borgo and Paulo Leitao. The role of foundational ontologies in manufacturing domain applications. In R. et al. Meersman, editor, Infrastructures for Virtual Enterprises - Networking Industrial Enterprises, volume LNCS 3290, pages 670–688. Springer, 2004.

[10] Stefano Borgo and Claudio Masolo. Foundational Choices in DOLCE. In S. Staab and R. Studer, editors, Handbook on Ontologies, pages 361– 381. Springer Verlag, 2nd edition, 2009.

[11] Stefano Borgo and Laure Vieu. Artifacts in Formal Ontology. In Anthonie Meijers, editor, Handbook of the Philosophy of the Technological Sciences. Technology and Engineering Sciences, volume 9, pages 273–307. Elsevier, 2009.

[12] Daniele Calisi, Luca Iocchi, Daniele Nardi, Carlo Matteo Scalzo, and Vittorio Amos Ziparo. Context-based design of robotic systems. Robotics and Autonomous Systems, 56(11):992–1003, 2008. Semantic Knowledge in Robotics.

[13] Carpanzano E., Cesta A., Orlandini A., Rasconi R., Suriano M., Umbrico A., Valente A. (2016) Design and implementation of a distributed part routing algorithm for reconfigurable transportation systems. International Journal of Computer Integrated Manufacturing.

[14] A. Ceballos, S. Bensalem, A. Cesta, L. de Silva, S. Fratini, F. Ingrand, J. Ocon, A. Orlandini, F. Py, K. Rajan, R. Rasconi, and M. van Winnendael. A Goal-Oriented Autonomous Controller for space exploration. In Proceedings of the ASTRA 2011, 11th Symposium on Advanced Space Technologies in Robotics and Automation, 2011.

[15] Cesta A. and Fratini S. (2008) The Timeline Representation Framework as a Planning and Scheduling Software Development Environment. In Proc. of the 27th Workshop of the UK Planning and Scheduling Special Interest Group, Edinburgh, UK. PlanSIG-08.

[16] B. Chandrasekaran and J.R. Josephson. Function in Device Representation. Engineering with Computers, 16(3/4):162–177, 2000.

[17] Ontologies for Robotics and Automation (ORA) Working Group. Ieee standard ontologies for robotics and automation. Technical report, IEEE Std 1872-2015, 2015.

[18] N. Guarino and C. Welty. An overview on Ontoclean. In S. Staab and R. Studer, editors, Handbook on Ontologies. Springer, 2009.

[19] Ronny Hartanto and Joachim Hertzberg. Fusing DL Reasoning with HTN Planning. In AndreasR. Dengel, Karsten Berns, ThomasM. Breuel, Frank Bomarius, and ThomasR. Roth-Berghofer, editors, KI 2008: Advances in Artificial Intelligence, volume 5243 of Lecture Notes in Computer Science, pages 62–69. Springer Berlin Heidelberg, 2008.

[20] J. Hirtz, R. B. Stone, D. A. McAdams, S. Szykman, and K. L. Wood. A functional basis for engineering design: Reconciling and evolving previous efforts. Research in Engineering Design, 13(2):65–82, 2001.

[21] Anna Hristoskova, E. Carlos Aguero, Manuela Veloso, and Filip De Turck. Heterogeneous Context-Aware Robots Providing a Personalized Building Tour. Int. J. of Advanced Robotic Systems, 2013.

[22] Yoshinobu Kitamura, Sho Segawa, Munehiko Sasajima, and Riichiro Mizoguchi. An Ontology of Classification Criteria for Functional Taxonomies. In IDETC/CIE. ASME, 2011.

[23] Y. Koren, U. Heisel, F. Jovane, T. Moriwaki, G. Pritschow, G. Ulsoy, and H. Van Brussel. Reconfigurable manufacturing systems. CIRP Annals - Manufacturing Technology, 48(2), 1999.

[24] S. Lemai and F. Ingrand. Interleaving Temporal Planning and Execution in Robotics Domains. In AAAI-04, pages 617–622, 2004.

[25] S. Lemaignan, R. Ros, L. Mosenlechner, R. Alami, and M. Beetz. ORO, a knowledge management platform for cognitive architectures in robotics. In Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on, pages 3548–3553, Oct 2010.

[26] C. Masolo, S. Borgo, A. Gangemi, N. Guarino, A. Oltramari, and L. Schneider. Wonderweb deliverable d17: The wonderweb library of foundational ontologies. Technical report, Laboratory for Applied Ontology, 2002. Technical report.

[27] G. Pahl, W. Beitz, J. Feldhusen, and K.H. Grote. Engineering Design. A Systematic Approach. Springer, London, UK, 3rd edition, 2007.

[28] P.K.Turaga, R. Chellappa, V. S. Subrahmanian, and O. Udrea. Machine Recognition of Human Activities: A Survey. IEEE Trans. Circuits Syst. Video Techn., 18(11):1473–1488, 2008.

[29] Edson Prestes, Joel Luis Carbonera, Sandro Rama Fiorini, Vitor A. M. Jorge, Mara Abel, Raj Madhavan, Angela Locoro, Paulo Goncalves, Marcos E. Barreto, Maki Habib, Abdelghani Chibani, Sébastien Ge´rard, Yacine Amirat, and Craig Schlenoff. Towards a core ontology for robotics and automation. Robotics and Autonomous Systems, 61(11):1193 – 1204, 2013. Ubiquitous Robotics.

[30] Lorenzo Solano, Pedro Rosado, and Fernando Romero. Knowledge representation for product and processes development planning in collaborative environments. International Journal of Computer Integrated Manufacturing, 27(8):787– 801, 2013.

[31] Il Hong Suh, Gi Hyun Lim, Wonil Hwang, Hyowon Suh, Jung-Hwa Choi, and Young-Tack Park. Ontology-based multi-layered robot knowledge framework (OMRKF) for robot intelligence. In Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on, pages 429–436, Oct 2007.

[32] Moritz Tenorth and Michael Beetz. Representations for robot knowledge in the KnowRob framework. Artificial Intelligence, pages –, 2015.

[33] H-P Wiendahl, Hoda A ElMaraghy, Peter Nyhuis, Michael F Zah, H-H Wiendahl, Niel Duffie, and Michael Brieke. Changeable manufacturing-classification, design and operation. CIRP Annals-Manufacturing Technology, 56(2):783– 809, 2007.

[34] Ramos, Luis. Semantic Web for manufacturing, trends and open issues: Toward a state of the art. Computers & Industrial Engineering. 90:444-460. 2015.

[35] Chandrasegaran, Senthil K, Ramani, Karthik, Sriram, Ram D, Horváth, Imré, Bernard, Alain, Harik, Ramy F, Gao, Wei. The evolution, challenges, and future of knowledge representation in product design systems. Computer-aided design. 45(2): 204:228, 2013.

[36] Umbrico A., Cesta A., Cialdea Mayer M., Orlandini A. (2017) PLATINUm: A New Framework for Planning and Acting. In: Esposito F., Basili R., Ferilli S., Lisi F. (eds) AI*IA 2017 Advances in Artificial Intelligence. AI*IA 2017. Lecture Notes in Computer Science, vol 10640. Springer, Cham.

[37] Muscettola N. (1994) HSTS: Integrating planning and scheduling. In: Zweben, M. and Fox, M.S., ed., Intelligent Scheduing. Morgan Kauffmann.

[38] J. Barreiro, M. Boyce, M. Do, J. Frank, M. Iatauro, T. Kichkaylo, P. Morris, J. Ong, E. Remolina, T. Smith, D. Smith. EUROPA: A Platform for AI Planning, Scheduling, Constraint Programming and Optimization. 2012. In ICKEPS 2012: the 4th International Competition on Knowledge Engineering for Planning and Scheduling

[39] S. Chien, D. Tran, G. Rabideau, S. Schaffer, D. Mandl, S. Frye. Timeline-based Space Operations Scheduling with External Constraints. 2010. In Proc. of the 20th International Conference on Automated Planning and Scheduling.

[40] Cesta A., Cortellessa G., Fratini S., Oddi A. (2009) Developing an end-to-end planning application from a timeline-representation framework. In Proc. of the 21st Innovative Application of Artificial Intelligence Conference. IAAI-09.

[41] F. Py, K. Rajan, C. McGann. A systematic agent framework for situated autonomous systems. 2010. In AAMAS-10. Proc. of the 9th International Conference on Autonomous Agents and Multiagent Systems.

[42] Borgo S., Cesta A., Orlandini A., Umbrico A. (2016) A planning-based architecture for a reconfigurable manufacturing system. In Proc. of the 26th International Conference on Automated Planning and Scheduling. ICAPS 2016.

[43] Monostori L, Kádár B, Bauernhansl T, Kondoh S, Kumara S, Reinhart G, Sauer O, Schuh G, Sihn W, Ueda K. Cyber-physical systems in manufacturing. 2016. CIRP Annals - Manufacturing Technology, 65(2):621-641.

[44] Tolio, T. Design of Flexible Production Systems. 2009. Milano, Italy: Springer.

[45] Flatscher M, Riel A. Stakeholder integration for the successful product–process co-design for next-generation manufacturing technologies. 2016. CIRP Annals - Manufacturing Technology, 65(1):181-184.

[46] Riichiro Mizoguchi, Yoshinobu Kitamura, and Stefano Borgo. A unifying definition for artifact and biological functions. Applied Ontology. 2016. 11(2):129–154. doi: 10.3233/AO-160165.

[47] Cialdea Mayer M., Orlandini A., Umbrico A. (2016) Planning and execution with flexible timelines: a formal account. Acta Informatica. 53(6-8). pp. 649-680.

[48] Riichiro Mizoguchi and Stefano Borgo. A Preliminary Study of Functional Parts as Roles. 2017. In Proc. of the 2nd Workshop on Foundational Ontology (FOUSTII), co-located with the 3rd Joint Ontology Workshop (JOWO2017), Bozen-Bolzano, Italy.