Situation Calculus for Controller Synthesis in Manufacturing Systems with First-Order State Representation

Situation Calculus for Controller Synthesis in Manufacturing Systems with First-Order State Representation

Giuseppe De Giacomo[a], Paolo Felli[b,*], Brian Logan[c], Fabio Patrizi[a], Sebastian Sardiña[d]

[a] *Sapienza University of Rome, Italy*
[b] *Free University of Bozen-Bolzano, Italy*
[c] *Utrecht University, Netherlands*
[d] *RMIT University, Australia*

Abstract

Manufacturing is transitioning from a mass production model to a service model in which facilities 'bid' to produce products. To decide whether to bid for a complex, previously unseen product, a facility must be able to synthesize, on the fly, a process plan controller that delegates abstract manufacturing tasks in a supplied process recipe to the available manufacturing resources. Often manufacturing processes depend on the data and objects (parts) they produce and consume. To formalise this aspect we need to adopt a first-order representation of the state of the processes. First-order representations of the state are commonly considered in reasoning about action in AI, and here we show that we can leverage the wide literature on the Situation Calculus and ConGolog programs to formalise this kind of manufacturing. With such a formalization available, we investigate how to synthesize process plan controllers in this first-order state setting. We also identify two important decidable cases—finite domains and bounded action theories—for which we provide techniques to actually synthesize the controller.

Keywords: reasoning about actions, situation calculus, automated synthesis, smart manufacturing

1. Introduction

To be able to remain competitive in a global marketplace characterized by faster market response and demand for customization, modern manufacturing companies are transitioning from a traditional mass production model to more agile and cost-effective

*Corresponding author
Email addresses: degiacomo@diag.uniroma1.it (Giuseppe De Giacomo), pfelli@unibz.it (Paolo Felli), b.s.logan@uu.nl (Brian Logan), patrizi@diag.uniroma1.it (Fabio Patrizi), sebastian.sardina@rmit.edu.au (Sebastian Sardiña)

Preprint submitted to Elsevier. https://doi.org/10.1016/j.artint.2021.103598

manufacturing networks and supply chains. Based on service-oriented principles, in the manufacturing as a service (MaaS) paradigm, the manufacturing infrastructure is shared on-demand by potentially large numbers of different manufacturing processes, so that the products to be manufactured are not known in advance, batch sizes are often small, and a facility may produce items belonging to heterogeneous product families for several customers at the same time [1, 2]. The cost of managing and maintaining the manufacturing infrastructure is thus distributed across all customers, enhancing resource utilization and reducing unit production costs, the lead-time is decreased and the sharing of knowledge and production processes makes better products. Different manufacturing models have been proposed in the literature to achieve the MaaS vision, with a recurring emphasis on flexibility, scalability, adaptability and customization, and on an increase in collaboration, automation, data and knowledge sharing through the entire supply chain. Examples include, among others, Agile Manufacturing, Virtual Manufacturing, Application Service Providers, Manufacturing Grid, and Cloud Manufacturing.

Among these, Cloud Manufacturing [3, 4, 5] is currently seeing the most advanced trends in MaaS. Enabled by an increasing development in information technology, IoT, embedded systems and cloud computing technologies, Cloud Manufacturing is proposing an advanced MaaS paradigm and business model in which manufacturing resources, such as Computer Numerical Control (CNC) machines and robots, are packaged as abstract descriptions of manufacturing capabilities, then advertised and made available to customers through a cloud platform. Likewise, the transformations that are required to manufacture a product are assumed to be specified as abstract, system-independent processes that need to be matched against the abstract capability descriptions that are offered by the facilities participating in the manufacturing cloud. By doing so, the MaaS paradigm, as envisioned by Cloud Manufacturing, allows the creation of dynamic production lines on-demand, by *composing* the pool of configurable manufacturing capabilities according to a pay-as-you-go business model. The objective is to connect customers to those providers who can best meet their product and process specifications and requirements, while limiting unit production cost and time. This model is also regarded as a more environmentally sustainable future for the manufacturing industry as a whole [6].

Critically, *automation* is key to achieve the MaaS vision: only minimal management effort or explicit customer-provider interaction can be assumed, so the fundamental requirement for MaaS is the ability of a facility participating in the cloud to autonomously assess, in real time, whether a given product can be manufactured in that facility. If the product can be manufactured, the facility may bid for the product, taking into account the overheads in terms of logistics. In mass production, the process planning phase, which transforms a process specification in concrete and practical production schedules for the resources on the shop floor, i.e., the *process plan*, is carried out by manufacturing engineers, and is largely a manual activity. In a MaaS approach, however, manual creation of process plans is clearly uneconomic for small batch sizes, and the time required to produce a plan is too great to allow facilities to bid for products in real time. Instead, manufacturing facilities must be able to *automatically synthesize process plans* for novel products 'on the fly'. To automatically establish that a facility can manufacture a product, the abstract manufacturing tasks in the *process recipe*—the

specification of how the product is to be manufactured—must be 'matched' against the available manufacturing resources in the facility. The resulting process plan details the low-level tasks to be executed and their order, the manufacturing resources to be used, and how materials and parts move between resources [7]. The *process plan controller*, namely, the control software that delegates each operation in the plan to the appropriate manufacturing resources, is then synthesized. In doing so, no sensitive information about the resources and internal processes of the selected facilities should be exposed to customers or competitors on the cloud platform.

Research in Artificial Intelligence and Computer Science can be exploited to provide a mathematical foundation for these domain concepts, and to solve the core challenges implicit in the MaaS vision. This is confirmed by recent efforts in basing MaaS on fundamental ideas coming from the literature on service composition in CS and behavior composition [8] in AI, so as to formalize the requirements and techniques for the automated synthesis of process plan controllers in the context of manufacturing [9, 10, 11, 12, 13]. These approaches have proven fruitful for developing preliminary and 'proof-of-concept' approaches for MaaS beyond the disciplines in which they were developed [14, 15].

However these approaches are based on a *propositional description* of the states of the devices, workpieces and processes, which is too idealized and insufficient to achieve a fully-fledged solution in practice. Indeed, manufacturing processes depend, in general, on the *objects* and *data* they produce and consume, including the cases where an *unbounded* number of product items (e.g., each with a unique serial number) or basic parts (e.g., each with a unique bar code, RFID tag or MAC address) must be produced. The approaches above do not explicitly account for this dependency, as they employ a finite state representation which hides important relationships between processes and data and, more importantly, cannot deal with a potentially unbounded number of objects. While in some cases this limitation can be circumvented, the resulting discretization is unwieldy and unnatural. For instance, one needs to directly encode into the process itself all the relevant dynamic knowledge, such as the current state of a part (e.g., painted, defective, etc.) or the state of a shared resource (e.g., a conveyor belt).

Instead, a concrete and realistic approach for MaaS necessarily requires a rich, relational description of states, an *information model*, as well as advanced computational techniques that are able to manipulate such relational representations. Although some previous work exists which can be used as the basis of an unambiguous description of the manufacturing concepts [16], the scientific literature has been lacking until now.

Our work here addresses exactly this point, by offering a "data-aware" process formalization in which data and objects are treated as first-class citizens. More specifically, we propose a relational representation of the states by relying on the research on reasoning about actions in AI. We see the operations in manufacturing processes as described by an action theory in logic, and the processes as high-level programs over such action theories. In this way, we can leverage the first-order state representations of action formalisms and the second-order/fixpoint characterization of state-change as provided by programs. Critically, we do not rely on ad-hoc representations, but we choose to adopt one of the most well developed formalisms for representing and reasoning about

dynamic systems in AI, namely the Situation Calculus, to encode information models and how they change as the result of actions. In fact, we deal with multiple Situation Calculus theories simultaneously, so as to model process recipes working over both an abstract information model and a concrete, facility-level information model. Process recipes and manufacturing resources, in turn, are modeled as high-level ConGolog programs [17] (over the action theories). All together, this yields a principled, formal and declarative representation of the MaaS setting.

By exploiting this rich representation, we formally define what it means to *realize a process recipe* in a manufacturing facility and present techniques to automatically synthesize controllers that implement those realizations. We show that these techniques are actually effective, that is, that they correspond to algorithms for extracting the actual controllers, when the resulting Situation Calculus action theories are *state-bounded* [18].

In our context, state-boundedness means that, while the facility may process an infinite number of objects overall, an unbounded number of them is never "accumulated": in any given state the number of objects being processed does not exceed a given bound. Notice that this case is the natural one in practice: the number of objects handled *at a given time* by the facility is naturally bounded by the size and structure of the shop-floor.

We stress that, technically and independently of the particular manufacturing setting in which we are interested, we provide here the first decidability result for controller synthesis in a setting with a relational/first-order state representation. We also observe that while our specific technical development is based on the Situation Calculus, our results and constructions can also be applied in other frameworks for reasoning about actions in AI as well as data-aware/artifact-centric processes frameworks in databases [19, 20, 21].

2. Situation Calculus and ConGolog

The Situation Calculus [22, 23] is a sorted predicate logical language for representing and reasoning about dynamically changing worlds. Changes in the world are the result of *actions*, which are terms in the language, and world histories are represented by *situation* terms. In addition to actions and situations, the language also includes an *object* sort, used to model the other entities. We assume a *unique countably infinite set* Δ *of objects* as the object sort. For all objects in Δ we have constants denoting them, called *standard names*, together with unique name assumption and domain closure [24, 25]. This will allow us to fix a single interpretation domain for models of situation calculus formulas and blur the distinction between such standard names and objects of the domain. This way of proceeding is common in databases [26] and is convenient in our case because it makes it possible to denote each object and piece of information that is relevant for the manufacturing application.

We assume a finite number of (*simple*) action types, each of which takes a tuple of objects as arguments. For example, $\textsc{drill}(part, dmtr, speed, x, y, z)$ represents the action of drilling a hole of a certain diameter, at a certain spindle speed, in a specific position of a given part. In the manufacturing domain, we are concerned with operations

that may occur simultaneously [23, 27, 28, 29], hence we adopt the *concurrent, non-temporal* variant of the Situation Calculus, where a concurrent or *compound* action $\boldsymbol{a}$ is a possibly infinite set of simple actions, like the one above, that execute simultaneously [23, Chapter 7]. For example, $\{\text{ROTATE}(part, speed), \text{SPRAY}(part, subst)\}$ represents the joint execution of rotating a part at a given speed while spraying it with some substance. As shorthand, we denote by $\mathbf{A}(\mathbf{x})$ the compound action of type $\mathbf{A}$ with a vector $\mathbf{x}$ of arguments (of the right size, and assuming a standard ordering of simple actions so that their order can be ignored here). In the concurrent, non-temporal variant, situations denote histories that stem from performing sequences of compound actions. We denote by S^0 the initial situation, i.e., the situation where no action has been performed yet, and assume that we have complete information about S^0. The situation resulting from executing a compound action $\boldsymbol{a}$ in s is represented by the situation term $do(\boldsymbol{a}, s)$. Predicates whose value varies from situation to situation are called *fluents* and take arguments of sort object plus a situation term as their last argument. For example, we may write *painted(part,s)* to denote that a part is painted in situation s. These are the only fluents we consider, that is we deal with relational fluents only.

A *basic action theory (BAT)* [30, 23] is a collection of axioms $\mathcal{D}$ describing the *initial situation*, *preconditions* and *effects* (and non-effects) of actions on fluents, as well as axioms for unique name assumptions and domain closure (for the object sort), which we denote, respectively, as $\mathcal{D}_{una}$ and $\mathcal{D}_{dc}$. We denote by $\mathcal{D}^0$ the (complete) *initial situation description*, i.e., the set of axioms of $\mathcal{D}$ that describe the initial configuration of the world (which is unique, under complete information). Such a configuration corresponds to the extension of all the fluents of the theory in the initial situation S^0. A special predicate $Poss(a, s)$ is used to express that the simple action a is executable in situation s, and a precondition axiom specifies when the action can be legally performed. Formally precondition axioms have the form: $Poss(a, s) \equiv \varphi(a, s)$ where $\varphi(a, s)$ is a *uniform* Situation Calculus formula, that is a formula referring to only one situation s (the current one). Predicate *Poss* is extended to compound actions by writing $Poss(\boldsymbol{a}, s)$. Typically we can assume that each $a \in \boldsymbol{a}$ also needs to be possible by itself in the situation s (i.e., that $Poss(a, s)$ and hence $Poss(\{a\}, s) \equiv Poss(a, s)$), although *Poss* can be arbitrarily restricted further (as we will do in our running example). We say that a situation s is *executable*, denoted by $Executable(s)$, if every (simple or compound) action performed in reaching s is possible in the situation in which it occurs [23].

Finally, a *successor state axiom* is used to encode causal laws specifying how each fluent changes as the result of executing (simple or) compound actions in the domain, encoding causal laws. Successor state axioms have the form: $f(\mathbf{x}, do(\boldsymbol{a}, s)) \equiv \varphi(\mathbf{x}, \boldsymbol{a}, s)$ where $\varphi(\mathbf{x}, \boldsymbol{a}, s)$ is again a uniform Situation Calculus formula over the current situation, which determines the value of the fluent $f(\mathbf{x}, s')$ in the next situation $s' = do(\boldsymbol{a}, s)$ resulting from executing $\boldsymbol{a}$. Examples of BATs are provided in Section 5.

Note that, having assumed complete information on S^0, BATs are *categorical*, i.e., they essentially admit a single model [23, 25]. The adoption of standard names allows for using, in a BAT $\mathcal{D}$, object names as constants. We call *active object constants* all the object names explicitly mentioned in the initial situation description $\mathcal{D}^0$ or in some precondition or successor-state axiom. In other words, these are all the constants mentioned in $\mathcal{D}$ but not in $\mathcal{D}_{una}$ or $\mathcal{D}_{dc}$. Obviously, being standard names, active object

constants are always interpreted as themselves. The set of active object constants in a BAT $\mathcal{D}$ is denoted as $AC_{\mathcal{D}}$, possibly without subscript if no ambiguity may arise.

Complex manufacturing processes are specified using *high-level programs*. They are "high-level" in that they comprise actions and tests that belong to the domain of concern (rather than based on classical variables and assignment), and they are meant to be executed against a theory of actions. In the Situation Calculus, several such languages have been developed, such as Golog [17], which includes the usual programming constructs as well as constructs for nondeterministic choices, ConGolog [31], which extends Golog to accommodate concurrency, and IndiGolog [25], which provides means for interleaving planning and execution.

We specify programs in a variant of ConGolog without recursive procedures [31] and where the test construct yields no transition and is final when satisfied [32, 33]. This results in a *synchronous test* construct that does not allow interleaving (every transition involves the execution of an action). These are the ConGolog constructs we consider:

$\boldsymbol{a}$	simple or compound action
$\phi?$	test for a condition
$\delta_1; \delta_2$	sequence
$\delta_1 \mid \delta_2$	nondeterministic branch
$\pi x.\delta$	nondeterministic choice of argument
δ^*	nondeterministic iteration
if ϕ **then** δ_1 **else** δ_2 **endIf**	conditional
while ϕ **do** δ **endWhile**	while loop
$\delta_1 \| \delta_2$	interleaved concurrency

where $\boldsymbol{a}$ can be a simple action (as in [31]), but also a compound action (these are the atomic instructions we use most in our setting) and ϕ is a situation-suppressed (uniform) Situation Calculus formula, i.e., a formula in the language with all situation arguments in fluents suppressed. We denote by $\phi[s]$ the (uniform) Situation Calculus formula obtained from ϕ by restoring the situation argument s into all fluents in ϕ. We require that the variable x in programs of the form $\pi x.\delta$ ranges over objects, and occurs in some action term in δ, i.e., $\pi x.\delta$ acts as a construct for the nondeterministic choice of action parameters.

Programs are executed over a BAT $\mathcal{D}$ (or Situation Calculus action theory, in general). This means that the fluents mentioned in tests and conditions must be those in the BAT $\mathcal{D}$. Similarly, all constants mentioned in a program δ come from the set $AC_{\mathcal{D}}$ of $\mathcal{D}$'s active object constants.

The semantics of ConGolog is specified in terms of single-steps, using the following two predicates [31]: *Final*(δ, s), specifying that the program δ may terminate in situation s, and *Trans*(δ, s, δ', s'), specifying that one step of program δ in situation s may lead to situation s' with δ' remaining to be executed. The definitions of *Trans* and

Final for the standard ConGolog constructs are given by:

$$
\begin{array}{l}
\mathit{Final}(\boldsymbol{a}, s) \equiv \texttt{False} \\
\mathit{Final}(\phi?, s) \equiv \phi[s] \\
\mathit{Final}(\delta_1; \delta_2, s) \equiv \mathit{Final}(\delta_1, s) \wedge \mathit{Final}(\delta_2, s) \\
\mathit{Final}(\delta_1 | \delta_2, s) \equiv \mathit{Final}(\delta_1, s) \vee \mathit{Final}(\delta_2, s) \\
\mathit{Final}(\pi x.\delta, s) \equiv \exists x.\mathit{Final}(\delta, s) \\
\mathit{Final}(\delta^*, s) \equiv \texttt{True} \\
\mathit{Final}(\delta_1 \| \delta_2, s) \equiv \mathit{Final}(\delta_1, s) \wedge \mathit{Final}(\delta_2, s)
\end{array}
$$

$$
\begin{array}{l}
\mathit{Trans}(\boldsymbol{a}, s, \delta', s') \equiv s' = do(\boldsymbol{a}, s) \wedge \mathit{Poss}(\boldsymbol{a}, s) \wedge \delta' = \epsilon \\
\mathit{Trans}(\phi?, s, \delta', s') \equiv \texttt{False} \\
\mathit{Trans}(\delta_1; \delta_2, s, \delta', s') \equiv \mathit{Trans}(\delta_1, s, \delta_1', s') \wedge \delta' = \delta_1'; \delta_2 \vee \\
\qquad \mathit{Final}(\delta_1, s) \wedge \mathit{Trans}(\delta_2, s, \delta', s') \\
\mathit{Trans}(\delta_1 \mid \delta_2, s, \delta', s') \equiv \mathit{Trans}(\delta_1, s, \delta', s') \vee \mathit{Trans}(\delta_2, s, \delta', s') \\
\mathit{Trans}(\pi x.\delta, s, \delta', s') \equiv \exists x.\mathit{Trans}(\delta, s, \delta', s') \\
\mathit{Trans}(\delta^*, s, \delta', s') \equiv \mathit{Trans}(\delta, s, \delta'', s') \wedge \delta' = \delta''; \delta^* \\
\mathit{Trans}(\delta_1 \| \delta_2, s, \delta', s') \equiv \mathit{Trans}(\delta_1, s, \delta_1', s') \wedge \delta' = \delta_1' \| \delta_2 \vee \\
\qquad \mathit{Trans}(\delta_2, s, \delta_2', s') \wedge \delta' = \delta_1 \| \delta_2'
\end{array}
$$

Above, we use ϵ to denote the empty program. In fact ϵ is simply an abbreviation for `True`?. Note that the conditional and while-loop constructs are definable: **if** ϕ **then** δ_1 **else** δ_2 **endIf** $= \phi?; \delta_1 | \neg\phi?; \delta_2$ and **while** ϕ **do** δ **endWhile** $= (\phi?; \delta)^*; \neg\phi?$. Also, for convenience we denote by **loop :** δ the program δ^* to make the nondeterministic iteration more evident.

A *configuration* is a pair $\langle \delta, s \rangle$ with program δ and situation s, hence *Trans* denotes one-step transitions between configurations and *Final* denotes when a configuration is final. We use *Trans** to denote the transitive closure of *Trans*, i.e., *Trans*$^*(\delta, s, \delta', s')$ means that there exists a sequence of one-step transitions evolving the configuration $\langle \delta, s \rangle$ into the configuration $\langle \delta', s' \rangle$. By using *Trans** we can define *Do*(δ, s, s') as an abbreviation for $\exists \delta'$. *Trans*$^*(\delta, s, \delta', s') \wedge$ *Final*(δ', s') stating that the *complete execution* of the program δ from s results in the new situation s' [17, 31]. Both *Trans** and *Do* can be easily defined in second-order logic by using *Trans* and *Final*.

It is important to notice that when we have complete information on the initial situation in a BAT, *Trans*, *Final*, *Trans** and *Do* are unequivocally defined by *the* model of the BAT. That is, the BAT, together with the definition of *Trans*, *Final*, *Trans** and *Do* is, again, categorical [31]. Moreover, since in our case the object domain consists of the standard names Δ, we have a syntactic denotation of each object, situation and program (and thus configuration). This allows us to blur the distinction between the semantic and syntactic objects of our formalization and switch back and forth seamlessly between the semantic and syntactic notions of configurations. Observe that, although unique, the BAT model has an infinite object domain, as well as infinite situations and programs. This makes working with such models nontrivial and substantially different from the way they are dealt with in model checking [34, 18, 35].

3. A Variant of ConGolog for Manufacturing

The standard constructs of ConGolog are not sufficient to express 'simultaneous' execution of processes in manufacturing. For this reason, we extend ConGolog with a new construct to model the simultaneous operation of two (or more) manufacturing resources: the *synchronized concurrency* operator $\delta_1|||\delta_2$ is used to represent the synchronized concurrent execution of programs δ_1 and δ_2—their next actions take place in the *same* transition step.

Moreover, we want our theories to be able to allow the synchronous execution of a set of actions even if they are not executable by themselves. The underlying assumption here is that a number of sub-systems (in our domain, manufacturing resources) can legally perform a joint step only if this is explicitly deemed possible by a "global" BAT for compound actions. This gives us complete control when modeling manufacturing facilities, and allows to capture arbitrary constraints on the usage of the resources available. For instance, to lift a heavy object we may necessarily need two robots to perform a lifting action at the same time on the same object, while the individual action of lifting an object may not be allowed by the individual theories of these robots when they are considered in isolation.

To capture this, the semantics of $|||$ is defined as follows:

$$\mathit{Final}(\delta_1|||\delta_2, s) \equiv \mathit{Final}(\delta_1, s) \wedge \mathit{Final}(\delta_2, s);$$

$$\begin{aligned}\mathit{Trans}(\delta_1|||\delta_2, s, \delta', s') \equiv \quad & \mathit{Trans}'(\delta_1, s, \delta_1', s_1') \wedge s_1' = \mathit{do}(\boldsymbol{a}_1, s) \wedge \\ & \mathit{Trans}'(\delta_2, s, \delta_2', s_2') \wedge s_2' = \mathit{do}(\boldsymbol{a}_2, s) \wedge \\ & \mathit{Poss}(\boldsymbol{a}_1 \cup \boldsymbol{a}_2, s) \wedge \delta' = (\delta_1'|||\delta_2') \wedge s' = \mathit{do}(\boldsymbol{a}_1 \cup \boldsymbol{a}_2, s),\end{aligned}$$

where *Trans*′ has analogous axioms as *Trans* except for two modifications:

1. For simple and compound actions, $\mathit{Trans}'(\boldsymbol{a}, s, \delta', s') \equiv s' = \mathit{do}(\boldsymbol{a}, s) \wedge \delta' = \epsilon$.
2. $\mathit{Trans}'(\delta_1|||\delta_2, s, \delta', s')$ does not require that $\mathit{Poss}(\boldsymbol{a}_1 \cup \boldsymbol{a}_2, s)$ be true.

This allows us to capture executability of compound actions without necessarily requiring the executability of (subsets of) their component simple actions: in the above, we impose $\mathit{Poss}(\boldsymbol{a}_1 \cup \boldsymbol{a}_2, s)$ without requiring also $\mathit{Poss}(\boldsymbol{a}_1, s)$ and $\mathit{Poss}(\boldsymbol{a}_2, s)$, although this can be explicitly required in the BAT (above, we implicitly extended the union operator to simple actions).

Observe that $\delta_1|||\delta_2$ requires both δ_1 and δ_2 to execute one (possibly compound) action at each step. In case one program requires more steps than the other to reach a final configuration, they cannot be executed in a synchronized, concurrent fashion. However, a program may include "no-op" actions to explicitly model when the resource can remain idle.

Finally, note that synchronized concurrency is distinguished from interleaved concurrency. For example, in the program $\delta_1\|\delta_2$, it is legal to execute either δ_1 or δ_2 completely before even starting the other, and it also legal to switch back and forth after each of their primitive actions. This is not possible with the synchronized concurrency program $\delta_1|||\delta_2$, under which *both* δ_1 and δ_2 must execute an action at each step.

4. Manufacturing as a Service

In this section we describe, in abstract terms, the MaaS model which we will formally capture through the logical framework illustrated in the next section. Specifically, we consider a type of MaaS setting as those recently envisioned by Cloud Manufacturing approaches, as discussed in Section 1. In this model, we consider a set of manufacturing *facilities*, each consisting of a set of configurable manufacturing *resources* that can be rapidly provisioned and released with minimal management effort or provider interaction. Examples of resources are CNC machines, robots and tools in flexible production lines. Facilities can join a manufacturing cloud to offer their production capabilities to cloud users (i.e., product designers) who wish to have their products manufactured. In turn, a user can submit to the cloud system a product model, i.e., the (process) *recipe*, that is a representation of the activities that are to be executed in order to complete an instance of the product. For simplicity, we assume that a product is entirely described by a single process recipe, although one could consider a hierarchical subdivision in multiple sub-assemblies and, thus, process recipes.

The main objective is to assess whether the product is manufacturable, that is, it can be manufactured through the cloud and, if so, to compute exactly how. The product can be manufactured if the recipe can be manufactured in a facility, that is, if an implementation of all the possible sequences of resource-independent operations, therein prescribed, can be delegated at each step to the resources in the facility, also taking care of all the low-level additional operations that may be required (which, being dependent on the facility and its resources, are not included in the recipe). Indeed, a process recipe is a *resource-independent* process, designed by the product designer without any specific knowledge or assumption on the manufacturing system that will be selected for its implementation, and it thus assumes an information model that is common throughout the cloud. A MaaS model must then be able to bridge the gap between such abstract representation and the description of the physical manufacturing processes that each facility can execute, according to the resources that are available in the facility.

When the product is manufacturable in a facility, the module responsible for delegating actions in the recipe to resources (we refer to such delegation as either *realization* or *orchestration*) is called a process plan controller, or simply a *controller*. In contrast to mass production, where process planning is carried out by manufacturing engineers and is largely a manual activity, this setting requires facilities to be able to bid for products in real time, that is, to be able to automatically check whether a novel product is manufacturable and, if so, synthesize a controller 'on the fly' prior to submitting a bid.

Following these observations, we can provide a specific MaaS framework whose basic components match the core elements of a MaaS paradigm. This is illustrated in Figure 1, where two distinct sorts are considered:

1. An *information model* $\mathcal{D}$ describes the data and the physical objects manipulated by processes, as well as the operations used to manipulate them.

2. A *process* δ describes the sequencing of the operations and captures the specific capabilities and the dynamic behavior of a component.

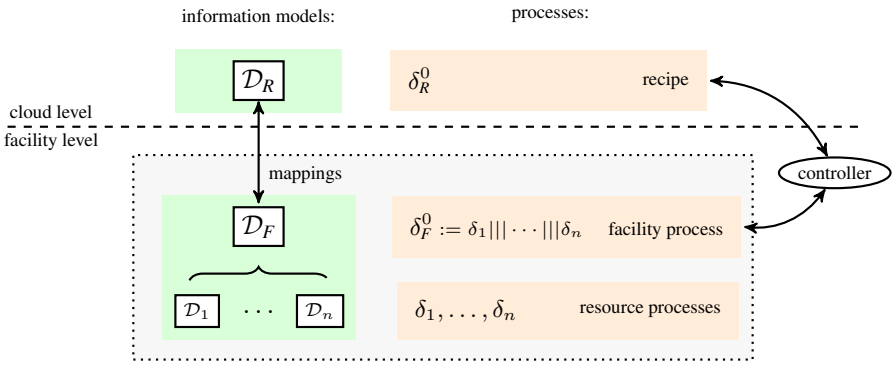

Figure 1: Framework for MaaS, divided into a cloud level and a facility level (only one facility is shown). A facility is constituted of a facility information model $\mathcal{D}_F$, a facility process δ_F^0, the mappings (and thus the cloud information model $\mathcal{D}_R$ as well).

The various components that form our MaaS framework can be described as follows. An example will be given in the next section, where these concepts are presented in detail. For now, we focus on highlighting the role that each of these components has, their internal structure, and their relationship. For simplicity, as reflected in Figure 1, we restrict ourselves to describe a single facility within the manufacturing cloud:

1. *Resources:* these are the physical manufacturing resources on the shop floor of a facility. Each resource is a couple $\langle \mathcal{D}_i, \delta_i \rangle$, with $i \in \{1, \ldots, n\}$, that has its own information model $\mathcal{D}_i$ and its own resource process δ_i, as these resources are typically sold by different companies. The actions and possible data in each $\mathcal{D}_i$ typically include low-level details.

2. *Facility information model:* it is the information model $\mathcal{D}_F$ for the facility, obtained by combining the information models $\mathcal{D}_i$ of each resource on the shop floor. It is the responsibility of the system integrator of the facility to suitably combine the information models of the resources.

3. *Facility process:* it is the process δ_F^0 resulting from the synchronous, concurrent execution of the process δ_i of each resource. Synchronous concurrency is needed to capture the fact that resources can (and may be required to) execute actions at the same time and on the same objects.

4. *Cloud information model*: it is the information model $\mathcal{D}_R$, common throughout the cloud, that is assumed by any recipe that can be designed. It is the core component of the framework whose existence precedes any other component. It represents a resource-independent information model of the data and objects that the recipes manipulate, and through which they are modeled. $\mathcal{D}_R$ is developed by the manager of the cloud infrastructure.

5. *Mappings*: a set of mappings $Maps$ represents the mechanism for relating the cloud information model $\mathcal{D}_R$, which is resource-independent, to the facility information model $\mathcal{D}_F$, that is resource-dependent. $Maps$ relates the possible abstract

executions described by the process recipe δ_R^0 (see below) with the concrete executions of the facility process δ_F^0. In fact we need two forms of mappings: the first relates the data and objects in $\mathcal{D}_R$ to those in $\mathcal{D}_F$; the second relates operations in $\mathcal{D}_R$ to possibly complex sub-processes that are required to implement them in the facility, according to $\mathcal{D}_F$. Operationally, these mappings are specified only once, namely when a facility joins the manufacturing cloud: the system integrator of the facility, who has access to both $\mathcal{D}_F$ and $\mathcal{D}_R$, is responsible for engineering them, although automation is also possible. These mappings are essential for the MaaS framework to be able to check the manufacturability of the product in a facility (and compute the controller) without exposing any internal detail, such as the number and type of resources or their processes. As depicted in Figure 1, the controller component is not at the cloud level but at the level of the facility: privacy is a critical requirement in MaaS.

6. *Facility*: by putting together some of the components above, a manufacturing facility is captured as a tuple $Fac = \langle \mathcal{D}_R, \mathcal{D}_F, \delta_F^0, Maps \rangle$, where $\mathcal{D}_R$ is the cloud information model (which is fixed), $\mathcal{D}_F$ is the facility information model, δ_F^0 is the facility program and $Maps$ is the set of mappings as above.

7. *Recipes*: a recipe is a resource-independent process δ_R^0 designed by a product designer without any specific knowledge or assumption on the manufacturing facility that will be selected for its realization. The information model describing the operations that compose the recipe *is* the cloud information model $\mathcal{D}_R$. The product designer can access $\mathcal{D}_R$ to design the recipe but cannot alter it.

8. *Controller*: when a controller exists for a given recipe and facility, i.e. when the product is manufacturable, this component is responsible for realizing the recipe by orchestrating the resources in the facility. It can be informally understood as a function relating each possible execution of the recipe δ_R^0 to one execution of the process δ_F^0 of the facility. It is not a cloud-level entity, and it is never exposed outside of the organization that owns the facility.

Given a facility $Fac = \langle \mathcal{D}_R, \mathcal{D}_F, \delta_F^0, Maps \rangle$ and a recipe δ_R^0, the *manufacturability problem* amounts to establishing whether there exists a controller to orchestrate the resources in the facility Fac to realize the recipe δ_R^0. The *synthesis task* is to automatically build the controller responsible for implementing the orchestration.

5. Manufacturing as a Service in the Situation Calculus

In this section we detail how the framework above is formally captured in our approach. We see the actions in manufacturing processes as described by an action theory in logic, and processes as high-level programs over such action theories. In this way, we can leverage the first-order state representations of action formalisms and the second-order/fixpoint characterization of state-change as provided by programs. We now formalize each framework component and provide examples.

5.1. Resources

As shown in Figure 1, a facility is composed of n manufacturing resources $\langle \mathcal{D}_i, \delta_i \rangle$, each identified by an index $i \in \{1, \ldots, n\}$. We formalize the information model $\mathcal{D}_i$ of each resource i as a BAT $\mathcal{D}_i$ (we overload symbols used in the MaaS framework itself), which specifies the resource's initial configuration and the actions it can execute (as a specification of fluent's dynamics through precondition and successor-state axioms). For convenience, we include the resource index i (a constant) as the last argument of all actions, as this will be useful when combining all these theories together. The resource process δ_i is formalized as a CongGolog program δ_i over the BAT $\mathcal{D}_i$.

Example 1 (Resources – based on the cell described in [36]). Consider a manufacturing cell consisting of five resources. Resource $\langle \mathcal{D}_1, \delta_1 \rangle$ is a robot that is able to perform different operations on parts by (autonomously) equipping the appropriate end effector ee, from a nearby rack, using the action EQUIP$(ee, 1)$. An end effector is a device or tool connected to the end of a robot arm, the nature of which depends on the intended task: by equipping a driller it can drill parts; by equipping a rivet gun it can apply rivets. The BAT $\mathcal{D}_1$ specifies when this simple action is possible for this resource:

$$\mathit{Poss}(\text{EQUIP}(ee, 1), s) \equiv \mathit{has\_effector}(ee, 1, s) \wedge \neg\exists e.\, \mathit{equipd}(e, 1, s)$$

that is, when the end effector is available on the rack and there is no end effector already equipped by the arm.

The *Poss* predicates for other actions are specified in the same way. The drilling operation is modeled as the action ROBOT_DRILL with arguments $part$, bit, $dmtr$, $speed$, $feed$, x, y, z for the ID of the part (i.e. the workpiece), the drilling bit ID, the diameter, the spindle speed, the feed rate, and hole position. For instance, a fully specified action is ROBOT_DRILL$(p, bit1, .7, 215, .2, 123, 87, 12, 1)$. The riveting action is analogous, and we represent it as RIVET$(part, rivet_type, x, y, z, 1)$. Additional operations are START_COMPRESSOR(1) for charging the compressor of the rivet gun (always possible); UNEQUIP(1) for unequipping the current end effector and place it back on the rack; SET_BIT$(bit, bit_type, dmtr, 1)$ for changing the drilling bit (for simplicity, we assume a rack of drill bit holders which can be changed autonomously, but it is also possible to use this action to model the fact that the robot returns to a default position to allow an operator to perform the tool change). For instance, we consider:

$$\begin{aligned}
&\mathit{Poss}(\text{ROBOT_DRILL}(part, bit, dmtr, speed, feed, x, y, z, 1), s) \equiv \\
&\quad \mathit{equipd}(driller, 1, s) \wedge \mathit{material}(part, m) \wedge \mathit{suitable}(bit, dmtr, m) \wedge \\
&\quad (\mathit{at}(part, 1, s) \vee \mathit{within_reach}(part, 1, s)) \wedge \mathit{part}(part, s) \wedge \mathit{safe}(1, s) \\
&\mathit{Poss}(\text{RIVET}(part, rivet_type, x, y, z, 1), s) \equiv \mathit{equipd}(rivet_gun, 1, s) \wedge \\
&\quad \mathit{drilled}(part, \mathit{hole}(x, y, z), s) \wedge \mathit{charged}(compressor, 1, s) \wedge \\
&\quad (\mathit{at}(part, 1, s) \vee \mathit{within_reach}(part, 1, s)) \wedge \mathit{part}(part, s) \wedge \mathit{safe}(1, s) \\
&\mathit{Poss}(\text{SET_BIT}(bit, bit_type, dmtr, 1), s) \equiv \mathit{tool_bits}(bit, bit_type, dmtr)
\end{aligned}$$

where $\mathit{at}(part, i, s)$ is a fluent used to specify that a part is currently allocated to the resource i, and which we assume to be included in the BAT of every resource. Similarly, $\mathit{within\_reach}(part, i, s)$ specifies that the work piece $part$ is within the envelope of the resource, so that it can either be moved on the working area, or operations can be performed on $part$ from a distance. Static relation $\mathit{tool\_bits}(bit, bit\_type, dmtr)$

represents the catalogue of tool bits available in this facility. We also use situation dependent or independent fluents, which for simplicity we assume to be available in each BAT of each resource i, to denote that a part exists in the cell, that a hole has been drilled in a part by a resource, that a part has a certain material, that it is safe for the robot to move the arm, etc. We will give examples of their successor-state axioms later.

Moreover, parts are moved between resources by a part-handling system, which is typically a collection of conveyor belts or similar pieces of equipment, although it is also possible that parts are explicitly exchanged by the resources themselves (e.g., robots). We model this by using special actions $\text{IN}(part, i)$ and $\text{OUT}(part, i)$ for each resource i, denoting that a part is moved into or out of the work area of the resource, respectively. This allows us to model either the physical movement of the part from/to the part-handling system or the acquisition/release of the exclusive access lock to the part. This approach makes sure that at most one resource is allocated a part (although other resources may be required to assist in complex machining or assembly operations, as we will show later). For the IN and OUT actions, we assume the following axioms to impose that each resource can hold at most one part at a time (larger bounds can be modeled in a similar way):

$$\mathit{Poss}(\text{IN}(part, i), s) \equiv \mathit{part}(part, s) \wedge \mathit{within\_reach}(part, i, s) \wedge \neg\exists p.\, \mathit{at}(p, i, s)$$
$$\mathit{Poss}(\text{OUT}(part, i), s) \equiv \mathit{part}(part, s) \wedge \mathit{at}(part, i, s)$$

Finally, a special action NOP, also common to all resources, can be used to keep the resource idle. The ConGolog program δ_1 (capturing the resource process) is shown in Figure 2. Note that all free parameters are implicitly existential (i.e., in the scope of a choice π).

loop :
IN$(part, 1)$
| **if** $\neg$*equipd*$(\texttt{driller}, 1)$ **then** EQUIP$(\texttt{driller}, 1)$ **endIf** ;
NOP* ;
if $\neg$*has_bit*$(bit\_type, dmtr, \texttt{driller}, 1)$ **then** SET_BIT$(bit, bit_type, dmtr, 1)$ **endIf** ;
ROBOT_DRILL$(part, bit, dmtr, speed, feed, x, y, z, 1)$
| **if** $\neg$*equipd*$(\texttt{rivet\_gun}, 1)$ **then** EQUIP$(\texttt{rivet_gun}, 1)$ **endIf** ;
NOP* ; RIVET$(part, rivet_type, x, y, z, 1)$;
if $\neg$*charged*$(\texttt{compressor}, 1)$ **then** START_COMPRESSOR(1) **endIf**
| OUT$(part, 1)$
| UNEQUIP(1)
| NOP

Figure 2: The resource program δ_1, representing a robot arm with two end effectors. The resource cannot equip an end effector and then not use it, nor attempt to equip it if already mounted on the arm. Similarly, the robot cannot release the part if in the process of drilling or riveting it. Note that these constraints are not aimed at capturing preconditions of actions, which are instead specified in the BAT $\mathcal{D}_i$ via *Poss*, but rather the control logic governing the robot, which is dictated by the resource's own design. The actual executions that are possible will depend on the interplay of the program, the BAT and the current situation, at each step.

We model the other resources in a similar way. $\langle \mathcal{D}_2, \delta_2 \rangle$ is a fixture that can perform an action HOLD_IN_PLACE$(part, force, 2)$, with a given clamping force. Resource

$\langle \mathcal{D}_3, \delta_3 \rangle$ is a robot that can move parts into and out of the cell from an external conveyor or pallet, position a part at a given location relative to another part (creating a single composite part as output), and, by equipping either a flat or hollow end effector, apply pressure to a part that is being worked on by another resource (hollow for drilling or milling, flat for riveting). These operations correspond to the actions IN_CELL$(part, weight, material, dimx, dimy, dimz, 3)$, OUT_CELL$(part, code, 3)$, POSITION$(part1, part2, part, x, y, z, 3)$, PRESSURE$(part, force, 3)$. These actions for $\langle \mathcal{D}_2, \delta_2 \rangle$ and $\langle \mathcal{D}_3, \delta_3 \rangle$ have the following preconditions:

$$\mathit{Poss}(\text{HOLD\_IN\_PLACE}(part, force, 2), s) \equiv \mathit{part}(part, s) \wedge \mathit{at}(part, 2, s)$$

$$\begin{aligned}
&\mathit{Poss}(\text{IN_CELL}(part, weight, material, dimx, dimy, dimz, 3), s) \equiv \\
&\quad \mathit{part}(part, s) \wedge \mathit{on_site}(part, s) \wedge \mathit{safe}(3, s) \\
&\mathit{Poss}(\text{OUT_CELL}(part, code, 3), s) \equiv \\
&\quad \mathit{part}(part, s) \wedge \mathit{at}(part, 3, s) \wedge \mathit{status}(part, code, s) \wedge \mathit{safe}(3, s) \\
&\mathit{Poss}(\text{POSITION}(part1, part2, part, x, y, z, 3), s) \equiv \neg\exists p.\, \mathit{holding}(p, 3, s) \wedge \neg\mathit{part}(part, s) \wedge \\
&\quad \mathit{part}(part1, s) \wedge \mathit{part}(part2, s) \wedge \mathit{at}(part1, 3, s) \wedge \mathit{within_reach}(part2, 3, s) \wedge \mathit{safe}(3, s) \\
&\mathit{Poss}(\text{PRESSURE}(part, force, type, 3), s) \equiv \mathit{part}(part, s) \wedge \\
&\quad (\mathit{at}(part, 3, s) \vee \mathit{within_reach}(part, 3, s)) \wedge ((\mathit{equipd}(\texttt{pressure_flat}, 3, s) \wedge \mathit{type}{=}\texttt{flat}) \\
&\quad \vee (\mathit{equipd}(\texttt{pressure_hollow}, 3, s) \wedge \mathit{type}{=}\texttt{hollow})) \wedge \mathit{safe}(3, s)
\end{aligned}$$

For instance, the first specifies that $\langle \mathcal{D}_2, \delta_2 \rangle$ can hold in place parts that are physically brought to the fixture. The fourth one specifies that $\langle \mathcal{D}_3, \delta_3 \rangle$ can position a part $part1$ onto a part $part2$ to create a composite part $part$ if the resource is not holding anything (with its arm), $part$ is not yet a physical part, $part1$ is allocated to the resource whereas $part2$ is within reach (e.g., held by another resource), and finally that the resource is in a safe condition to operate. The last one specifies that $\langle \mathcal{D}_3, \delta_3 \rangle$ can apply pressure to a part $part$ that is either allocated to the resource or is within reach, provided that the right type of pressure applicator is already equipped and that the resource is in a safe condition to operate.

loop :
NOP | (IN$(part, 2)$; (NOP | HOLD_IN_PLACE$(part, force, 2)$)* ; OUT$(part, 2)$)

loop :
IN$(part, 3)$
| (EQUIP$(\texttt{pressure_hollow}, 3)$ | EQUIP$(\texttt{pressure_flat}, 3)$) ;
NOP* ; PRESSURE$(part, force, type, 3)^*$; UNEQUIP(3)
| EQUIP$(\texttt{gripper}, 3)$; POSITION$(part1, part2, part, x, y, z, 3)$
| IN_CELL$(part, weight, material, dimx, dimy, dimz, 3)$
| OUT_CELL$(part, code, 3)$
| NOP

Figure 3: Resources programs δ_2 and δ_3.

Note that the information about the weight, material and size of parts loaded into the cell is made available by the arguments of IN_CELL, so that new fluents are added

to the extension to represent information about fresh workpieces (see successor-state axioms, further below). As a restriction, one is not allowed to use operators that are not axiomatized in the theories themselves (e.g., to write $\mathit{dimx}(part, x) \wedge \geq (x, 30)$ to check that a part is wider than 30 inches), but we can assume discretized intervals for each numeric parameter, and consider fluents such as $\mathit{dimx}(part, \mathtt{wide})$.

Resource $\langle \mathcal{D}_4, \delta_4 \rangle$ is an upright drilling machine for drilling parts with high precision. Finally, $\langle \mathcal{D}_5, \delta_5 \rangle$ is a human operator, who can operate the drilling machine by executing the action OPERATE_MACHINE$(4, 5)$, can physically enter/exit the cell with ENTER(5) and EXIT(5), can bring small parts into the cell with IN_CELL$(part, \cdots, 5)$, and can apply glue to parts with SPRAY_GLUE$(part, glue_type, 5)$. Hence, in $\mathcal{D}_4$ (drilling machine) we have the precondition axiom:

$$\mathit{Poss}(\text{MACHINE\_DRILL}(part, bit, dmtr, speed, feed, x, y, z, 4), s) \equiv \\ \mathit{material}(part, m) \wedge \mathit{suitable}(bit, dmtr, m) \wedge \mathit{at}(part, 4, s)$$

and in $\mathcal{D}_5$ (human operator):

$$\mathit{Poss}(\text{IN\_CELL}(part, weight, material, \cdots, 5), s) \equiv \mathit{on\_site}(part, s) \wedge \mathit{size}(part, \mathtt{small})$$
$$\mathit{Poss}(\text{SPRAY\_GLUE}(part, glue\_type, 5), s) \equiv \mathit{within\_reach}(part, 5, s) \wedge \mathit{avail}(glue\_type)$$
$$\mathit{Poss}(\text{SAFETY\_SWITCH}(5), s) \equiv \text{true}$$

while the processes for these resources are represented by the ConGolog programs δ_4 and δ_5 in Figure 4.

loop :
IN$(part, 4)$ | MACHINE_DRILL$(part, bit, dmtr, speed, feed, x, y, z, 4)$ | OUT$(part, 4)$ | NOP

loop :
if *entered*(5) **then**
IN$(part, 5)$ | SPRAY_GLUE$(part, glue_type, 5)$ | OPERATE_MACHINE$(j, 5)$ |
IN_CELL$(part, \cdots, 5)$ | OUT$(part, 5)$ | (EXIT(5) ; SAFETY_SWITCH$(\mathtt{off}, 5)$) | NOP
else (SAFETY_SWITCH$(\mathtt{on}, 5)$; ENTER(5)) | NOP **endIf**

Figure 4: The remaining resource programs δ_4 and δ_5.

According to δ_5, the human operator essentially has two distinct sets of available operations when they are inside and outside the cell (the fluent *entered*(5) is of course affected by the execution of ENTER(5) and EXIT(5)). Note how the operator is required (by safety regulations) to always operate a safety switch immediately before and after entering or exiting the cell. Nonetheless, the BATs for the robots are "unaware" of the existence of such an action: the *Poss* axioms for the robots mention a fluent $\mathit{safe}(i, s)$ that must be tested for machining or assembly actions (as the robot arm may need to move within its spatial envelope), but there is still no relation between the value of this fluent and the execution of SAFETY_SWITCH, as these belong to different BATs. An analogous consideration applies to the fluent $\mathit{within\_reach}(part, i, s)$, as this would require to talk about other resources nearby. This will be covered in the next section, when we

will show how the BATs for individual resources are merged in a semi-automated fashion into a single $\mathcal{D}_F$, namely the facility information model, as depicted in Figure 1. For now, we can assume:

$$\mathit{safe}(i, s) \equiv \mathsf{true}$$
$$\mathit{within\_reach}(\mathit{part}, i, s) \equiv \mathit{at}(\mathit{part}, i, s)$$

Finally, examples of successor-state axioms for (some of) the fluents mentioned in the BATs above are listed below (they are assumed to be the same in each BAT). For conciseness, and whenever needed, we denote by $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$ the parameters that appear as dots in the fluents within the scope of quantifiers.

$$\begin{aligned}
&\mathit{equipd}(e, i, do(a, s)) \equiv a = \textsc{equip}(e, i) \vee (\mathit{equipd}(e, i, s) \wedge a \neq \textsc{unequip}(i))\\
&\mathit{part}(\mathit{part}, do(a, s)) \equiv \exists \mathbf{y}.\ (a = \textsc{in_cell}(\mathit{part}, \cdots) \vee a = \textsc{position}(\cdot, \cdot, \mathit{part}, \cdots)) \vee\\
&\quad (\mathit{part}(\mathit{part}, s) \wedge \neg\exists \mathbf{x}.\ a = \textsc{out_cell}(\mathit{part}, \cdots)\\
&\mathit{at}(\mathit{part}, i, do(a, s)) \equiv a = \textsc{in}(\mathit{part}, i) \vee (\exists \mathbf{y}.\ a = \textsc{position}(\cdot, \mathit{part}2, \mathit{part}, \cdots) \wedge\\
&\quad \mathit{at}(\mathit{part}2, i, s)) \vee \exists \mathbf{x}.\ a = \textsc{in_cell}(\mathit{part}, \cdots, i) \vee (\mathit{at}(\mathit{part}, i, s) \wedge a \neq \textsc{out}(\mathit{part}, i) \wedge\\
&\quad \neg\exists \mathbf{z}.\ a = \textsc{out_cell}(\mathit{part}, \cdots))\\
&\mathit{material}(\mathit{part}, \mathit{material}, do(a, s)) \equiv \exists \mathbf{y}.\ a = \textsc{in_cell}(\mathit{part}, \cdots, \mathit{material}, \cdots) \vee\\
&\quad (\mathit{material}(\mathit{part}, \mathit{material}, s) \wedge \neg\exists \mathbf{x}.\ a = \textsc{out_cell}(\mathit{part}, \cdots))\\
&\mathit{drilled}(\mathit{hole}(\mathit{part}, x, y, z), do(a, s)) \equiv \exists \mathbf{y}.\ (a = \textsc{robot_drill}(\mathit{part}, \cdots, x, y, z, \cdot) \vee\\
&\quad a = \textsc{machine_drill}(\mathit{part}, \cdots, x, y, z, \cdot)) \vee\\
&\quad (\mathit{drilled}(\mathit{hole}(\mathit{part}, x, y, z), s) \wedge \neg\exists \mathbf{x}.\ a = \textsc{out_cell}(\mathit{part}, \cdots))
\end{aligned}$$

For instance, the first states that an end effector e is equipped by a resource i either if the resource executes action $\textsc{equip}(e, i)$ or e was already equipped by i and it is not removed. Similarly, a part exists after it is loaded into the cell if it is a composite part resulting from a positioning action or if already in the cell and not moved out of the cell. Analogously, the third one states that a part is allocated to resource i if it was moved to i or it is a composite part obtained by positioning another part on one that is in i.

The remaining predicates are situation-independent (such as *tool_bits*$/3$, *avail*$/1$ or *suitable*$/3$), that is, their extensions are included in the theories, but are not affected by actions as they capture instead domain knowledge (on materials, manufacturing transformations, etc). □

5.2. Facility information model

The facility information model $\mathcal{D}_F$ for the manufacturing facility is also formalized as a Situation Calculus BAT. It is in this model that a system integrator will combine, in a semi-automated manner, all information models $\mathcal{D}_1, \ldots, \mathcal{D}_n$ of the resources, e.g., by taking into account knowledge about which resources are connected by the part-handling system, which subsets of resources can work on the same parts, which compound actions are *meaningful*, etc.

To formalize the facility information model as a Situation Calculus BAT $\mathcal{D}_F$ from the BATs $\mathcal{D}_1, \ldots, \mathcal{D}_n$ of the resources we need to follow several steps:

1. Define the initial situation description for situation S_F^0 in the BAT $\mathcal{D}_F$ as the union of the descriptions $\mathcal{D}_i^0$ of the BAT of the resources, $\mathcal{D}_F^0 = \mathcal{D}_1^0 \cup \cdots \cup \mathcal{D}_n^0$.

2. Define the successor-state axioms of fluents for compound actions. This is done by trivially extending the successor-state axioms for simple actions that are in the BATs $\mathcal{D}_1, \ldots, \mathcal{D}_n$.

3. Define the *Poss* predicate for *all and only* the exact compound actions that are deemed meaningful, and establish whether each of these compound actions also requires the executability of its constituent simple actions as defined in $\mathcal{D}_1, \ldots, \mathcal{D}_n$. Recall that $\mathit{Poss}(\boldsymbol{a}, s)$ does not imply, in general, $\mathit{Poss}(a, s)$ for every $a \in \boldsymbol{a}$ (see Section 3, including the definition of $\mathit{Trans}(\delta_1 ||| \delta_2, s, \delta', s')$).

4. In doing the above, possibly introduce situation-independent fluents (with their definitions) that may be needed to relate the value of fluents in the various resources and to facilitate the writing of the axioms in the theory.

Note that the set of fluents in $\mathcal{D}_F$ is $\mathcal{F}_F = \bigcup_i \mathcal{F}_{\mathcal{D}_i}$, since we do not need to include the situation-independent fluents mentioned above (which can be seen as abbreviations). The set of active object constants mentioned in $\mathcal{D}_F$ is $AC_{\mathcal{D}_F} = \bigcup_i AC_{\mathcal{D}_i}$. Also, we denote by $\mathcal{A}_F$ the set of *action types* in $\mathcal{D}_F$.

The resulting BAT $\mathcal{D}_F$ allows for capturing any possible execution of the facility that is composed of meaningful compound actions. Note however that this does not capture any knowledge about the *manufacturing transformations* yet, i.e., the designed (sub)programs which lead to an increase of value of workpieces (such as a set of possibly complex sequences of actions that collectively implement the drilling, polishing, painting, etc. of a workpiece, and which typically include various steps and also auxiliary compound actions and tests). This notion will be modeled in Section 5.5.

Example 2 (Facility information model). Consider the manufacturing cell from the previous example. To be able to integrate all resources together, one needs to address various aspects, of which we consider only a few. First, successor state axioms have to consider compound actions. For instance, the first successor state axiom listed at the end of Example 1, in the previous section, trivially becomes:

$$\mathit{equipd}(e, i, do(\boldsymbol{a}, s)) \equiv \text{EQUIP}(e, i) \in \boldsymbol{a} \vee (\mathit{equipd}(e, i, s) \wedge \text{UNEQUIP}(i) \notin \boldsymbol{a})$$

Second, we have to take into account the layout of the shop floor, i.e., specify which resources can collaborate together for executing a compound action (the rest will need to remain idle). We do so by a special situation-independent predicate $\mathit{coopMatrix}(i, j)$ specifying that resource $\langle \mathcal{D}_i, \delta_i \rangle$ can cooperate with resource $\langle \mathcal{D}_j, \delta_j \rangle$, that is, it can perform an action on a part that is currently allocated to the other (i.e. such that $\mathit{at}(part, j, s)$ holds). Hence we axiomatize in $\mathcal{D}_F$ the *within_reach* fluent (so far assumed to be such that $\mathit{within\_reach}(part, i, s) \equiv \mathit{at}(part, i, s)$ – see Example 1 in the previous section) as follows:

$$\mathit{within\_reach}(part, i, s) \equiv \exists j.\ j \neq i \wedge \mathit{at}(part, j, s) \wedge \mathit{coopMatrix}(i, j)$$

We also connect the action SAFETY_SWITCH of the operator with the two robots in the cell by the successor-state axiom:

$$\begin{aligned} \mathit{safe}(i, do(\boldsymbol{a}, s)) \equiv (\mathit{safe}(i, s) \wedge \neg \exists j.\ \text{SAFETY\_SWITCH}(\texttt{off}, j) \in \boldsymbol{a}) \vee \\ \exists j.\ \text{SAFETY\_SWITCH}(\texttt{on}, j) \in \boldsymbol{a}) \end{aligned}$$

Third, once this step is completed (for all predicates, as needed), one needs to define the *Poss* predicate of compound actions. $\mathcal{D}_F$ includes arbitrary axioms for this purpose, however the following two are typically needed:

$$\begin{aligned}
&\mathit{Poss}(\boldsymbol{a} \cup \text{NOP}, s) \equiv \mathit{Poss}(\boldsymbol{a}, s)\\
&\mathit{Poss}(\{\text{IN}(part, i), \text{OUT}(part, j)\}, s) \equiv\\
&\quad \mathit{Poss}(\text{IN}(part, i), s) \wedge \mathit{Poss}(\text{OUT}(part, j), s) \wedge \mathit{partHandling}(j, i)
\end{aligned}$$

The first establishes that if an action $\boldsymbol{a}$ is executable, so is any compound action that is equal to $\boldsymbol{a}$ but to which NOP actions are added (note that NOP actions can be executed by resources only when their program allows it). The second specifies the possible passing of parts between resources, as allowed by the part-handling system. It assumes a situation independent predicate $\mathit{partHandling}/2$ so that $\mathit{partHandling}(j, i)$ specifies that, according to the layout of the shop floor, it is possible to move a part from resource i to resource j (typically by using fixed conveyors). This predicate can be easily replaced with a situation-dependent fluent.

Further, it remains to explicitly specify the *Poss* for the remaining compound actions that are meaningful in the facility although, in principle, it is also possible to automatically determine the preconditions for any compound action resulting from any possible combination of simple actions. For instance, in the running example we want to allow the compound action in which resource 1 (the first robot arm) performs a drilling operation on a part that is currently positioned on the fixture (resource 2), while resource 3 (the second robot arm) applies opposing pressure on the part (with a hollow pressure applicator). In this case we do not impose additional constraints, but we require the executability of these actions individually, according to the *Poss* of individual theories:

$$\begin{aligned}
&\mathit{Poss}(\{\text{ROBOT_DRILL}(part, \cdots, 1), \text{HOLD_IN_PLACE}(part, \cdots, 2), \text{PRESSURE}(part, \cdots, 3)\}, s) \equiv\\
&\quad \mathit{Poss}(\text{ROBOT_DRILL}(part, \cdots, 1), s) \wedge \mathit{Poss}(\text{HOLD_IN_PLACE}(part, \cdots, 2), s) \wedge\\
&\quad \mathit{Poss}(\text{PRESSURE}(part, \cdots, 3))
\end{aligned}$$

In turn, according to $\mathcal{D}_1$-$\mathcal{D}_3$, this requires that the part is held on the fixture, that it is within the envelope of both robots, that the right drilling bit is in the end effector, etc (see the *Poss* of these simple actions in Example 1).

According to the specification of *Trans*, by denoting the compound action above as $\{a_1, a_2, a_3\}$ for short (the order is irrelevant), in our example with five resources we have that if $\mathit{Trans}(a_1|||a_2|||a_3|||\text{NOP}|||\text{NOP}, s, \delta', s')$ then it must be that $\mathit{Poss}(\{a_1, a_2, a_3, \text{NOP}, \text{NOP}\}, s)$ and therefore $\mathit{Poss}(\{a_1, a_2, a_3\}, s)$, as a result of $\mathit{Poss}(\boldsymbol{a} \cup \text{NOP}, s) \equiv \mathit{Poss}(\boldsymbol{a}, s)$. Given the axiom above, this also implies $\mathit{Poss}(a_i)$ for $i \in \{1, 2, 3\}$, but this is not true in general (it is not required by *Trans*) and it can be determined case-by-case when generating $\mathcal{D}_F$. Also, it is not always the case that $\mathit{Poss}(\{a_1, a_2, a_3, \text{NOP}, \text{NOP}\}, s)$ implies $\mathit{Poss}(\{a_1, a_2, a_3, a_4, a_5\}, s)$ for any action a_4, a_5.

This gives the system integrator great freedom and flexibility, and it allows to encode arbitrary knowledge about the possible meaningful executions of a set of resources on the shop floor. □

5.3. Facility process

Given a set of n processes of manufacturing resources, each modeled as a ConGolog program δ_i over $\mathcal{D}_i$, $i \in \{1, \ldots, n\}$, the resulting facility process δ_F^0 is defined as the ConGolog program $\delta_F^0 := \delta_1 ||| \cdots ||| \delta_n$ over the BAT $\mathcal{D}_F$, which models the facility information model. Observe the use of the new *synchronized concurrency operator* introduced in Section 3. This operator executes all programs δ_i synchronously, generating a compound action from the actions in the various resources. In doing so, as explained in the previous section, it requires the executability of compound actions as defined in $\mathcal{D}_F$, without necessarily relying on the executability of their component actions as defined in the resources' BATs $\mathcal{D}_1, \ldots \mathcal{D}_n$.

5.4. Cloud information model

As explained in Section 4, the cloud information model $\mathcal{D}_R$ is the information model common throughout the cloud to be used by every recipe. It is the core component of our MaaS framework whose existence precedes any other component. It represents a resource-independent information model of data and objects the recipes manipulate, and on which they are expressed. The information model is specified by the manager of the cloud infrastructure.

We formalize $\mathcal{D}_R$ as an action theory $\mathcal{D}_R$, which is a variant of a Situation Calculus BAT. Importantly, $\mathcal{D}_R$ is specified *without knowing* the possible facilities in the cloud nor the resources available in each facility. For this reason, fluents in $\mathcal{D}_R$ are abstract, and they are typically affected by the resource-independent actions used in recipes. The evolution of such fluents can be specified through the initial situation description $\mathcal{D}_R^0$ of an initial situation S_R^0 of $\mathcal{D}_R$, and successor-state axioms in $\mathcal{D}_R$.

However, when specifying a recipe, we also want to use additional fluents whose interpretation is determined at runtime during the execution on an actual facility, i.e., during production. For instance, in a recipe we may need to prescribe a runtime test in order to determine how the manufacturing process should continue. We call such fluents *observations* and denote the set of observations by *Obs*. Note that the fluents in *Obs* are not determined by the initial situation description $\mathcal{D}_R^0$ and they do not have successor-state axioms. Their evolution depends on how the cloud information model is coupled with the facility information model through the *mappings*, as described in the next section.

From the Situation Calculus point of view, the resulting action theory $\mathcal{D}_R$ is a variant of a BAT (free-fluent BATs [37]), which omits successor-state axioms for the fluents in *Obs*. We denote such variants BAT$^-$ to stress this distinction. Observe that, even under complete information about the initial situation, a BAT$^-$ admits, in general, many models: without mappings relating observations to the actual facility, these fluents are free to take any extension at each situation. Nonetheless, as we formalize next, as we are capable of inspecting the facility to determine the extension of the fluent in *Obs* through the mappings, the resulting theory still admits a unique model. We will provide an example in the next section.

We denote by $\mathcal{F}_R$ the set of fluents in $\mathcal{D}_R$, with $Obs \subseteq \mathcal{F}_R$, and by $AC_{\mathcal{D}_R}$ the set of active object constants of $\mathcal{D}_R$. Finally, we denote by $\mathcal{A}_R$ the set of action types in $\mathcal{D}_R$.

5.5. Mappings

In order to establish the manufacturability of a product by a given facility, we need to define formal mappings between the abstract, resource-independent BAT$^-$ $\mathcal{D}_R$ and the BAT $\mathcal{D}_F$. These mappings are computed for each facility when it joins the manufacturing cloud, either automatically or by hand [36]. Following the ideas in [38], we define such mappings as follows:

- For each action type $\mathbf{A} \in \mathcal{A}_R$ with parameters $\mathbf{x}$, the action $\mathbf{A}(\mathbf{x})$ is mapped to a (arbitrarily complex) program $\delta_{\mathbf{A}}(\mathbf{x})$ in $\mathcal{D}_F$ composed of physical (compound) actions of the available resources, augmented with, e.g., the passing of parts through the part-handling system, the equipping of end effectors, etc.

- For each fluent $f \in \mathcal{F}_R$ with parameters $\mathbf{x}$, the atomic formula $f(\mathbf{x}, s_R)$ in the current situation s_R is mapped to a (uniform) Situation Calculus formula $\varphi_f(\mathbf{x}, s_F)$ over the fluents in $\mathcal{F}_F$ of the facility information model. The mapping establishes that $f(\mathbf{x}, s_R)$ in the current situation s_R of the BAT $\mathcal{D}_R$ on the recipe has the same extension as $\varphi_f(\mathbf{x}, s_F)$ in the current situation s_F on the BAT $\mathcal{D}_F$ of the facility.

  If f is an observation in *Obs* (i.e., without successor-state axiom, such as *precision* in Example 4) the mapping gives the extension to the observation $f(\mathbf{x}, s_R)$. If instead f is not an observation in *Obs*, then the mapping imposes a consistency requirement between the two theories as the fluent f is constrained by the initial situation description and its successor-state axiom in $\mathcal{D}_R$.

Summarizing, the mappings consist of a set $Maps$ of mapping rules of two forms:

- Mapping rules for $\mathcal{D}_R$'s actions of the form :

$$\mathbf{A}(\mathbf{x}) \leftrightarrow \delta_{\mathbf{A}}(\mathbf{x}) \qquad \text{(for each } \mathbf{A} \in \mathcal{A}_R\text{)}$$

  where $\mathbf{A}(\mathbf{x})$ is an action type in $\mathcal{A}_R$ with parameters $\mathbf{x}$, and $\delta_{\mathbf{A}}(\mathbf{x})$ is a program over $\mathcal{D}_F$ with parameters $\mathbf{x}$;

- Mapping rules for $\mathcal{D}_R$'s fluents of the form:

$$f(\mathbf{x}) \leftrightarrow \varphi_f(\mathbf{x}) \qquad \text{(for each } f \in \mathcal{F}_R\text{)}$$

  where $f(\mathbf{x})$ is a fluent of $\mathcal{D}_R$ with the situation argument suppressed and $\varphi_f(\mathbf{x})$ is a situation-suppressed (uniform) Situation Calculus formula over $\mathcal{D}_F$.

We impose two requirements on $\varphi_f(\mathbf{x})$ as well as on formulas occurring in tests of $\delta_{\mathbf{A}}(\mathbf{x})$. First of all, the active object constants mentioned in some mapping come from $AC_{\mathcal{D}_F} \cup AC_{\mathcal{D}_R}$. Second, these formulas must be *domain-independent* [26]. In our context a (uniform) Situation Calculus formula φ is domain-independent if its evaluation depends only on the objects appearing in the extension of $\mathcal{D}_F$'s fluents in the current situation, i.e., the *active domain* of the current situation, and not by other standard names in the domain Δ. One way to obtain domain-independence is to disallow negation $\neg\beta$ and instead use logical difference $\alpha \wedge \neg\beta$. Note that domain-independence is a standard requirement in databases that allows focusing on the active domain only, without loss of generality [26].

Example 3 (Mappings). To map the resource-independent DRILL action to a program specifying the possible ways in which a drilling operation can be performed in the facility of the running example, we specify:

$$\{\text{DRILL}(part, dmtr, speed, x, y, z)\} \leftrightarrow (\ \mathcal{A}_1 ||| \cdots ||| \mathcal{A}_n\ )^*\ ;$$
$$\textbf{if } \textit{size}(part, \texttt{large}) \textbf{ then } \delta_{\text{D}}^1 \textbf{ else } \delta_{\text{D}}^1 \mid \delta_{\text{D}}^2$$

$$\delta_{\text{D}}^1 = \pi\ \textit{bit}, \textit{feed}, \textit{force}, i, j, k.(\text{PRESSURE}(part, force, \texttt{hollow}, i) |||$$
$$\text{HOLD\_IN\_PLACE}(part, \texttt{3k}, j)\ |||$$
$$\text{ROBOT\_DRILL}(part, bit, dmtr, speed, feed, x, y, z, k))$$
$$\delta_{\text{D}}^2 = \pi\ \textit{bit}, \textit{feed}, i, j.(\text{OPERATE\_MACHINE}(i, j) |||$$
$$\text{MACHINE\_DRILL}(p, bit, dmtr, speed, feed, x, y, z, j))$$

where each $\mathcal{A}_i$ stands for $\pi \mathbf{x}.\ a_{i,1}(\mathbf{x}) \mid \cdots \mid a_{i,q_i}(\mathbf{x})$: i.e., each resource may perform preparatory sequences of actions before the specified compound actions. Intuitively, the rule states that, in this facility, large parts can only be drilled by using three actions for clamping (with a fixed force), drilling and applying pressure with a hollow pressure applicator (for counterbalancing the drilling pressure), whereas small parts may also be drilled by manually operating a drilling machine.

Note that these rules do not specify which resources should be used for a particular operation, hence in a large facility with many pieces of equipment this allows a high degree of flexibility in the allocation of resources to these tasks. In our simple example, with only five resources, such allocation is obvious (δ_{D}^1 can be executed by selecting i=3, j=2, k=1, and δ_{D}^2 by selecting i=5 and j=4).

We can write similar mapping rules for the fluent *precision* $\in Obs$, e.g., specifying how to observe the precision of a hole that was drilled:

$$\textit{precision}(\textit{hole}(part, x, y, z), precision) \leftrightarrow$$
$$\textit{part}(part) \wedge \textit{drilled}(\textit{hole}(part, x, y, z), i) \wedge \textit{prec\_rating}(precision, i)$$

This mapping captures the fact that the precision of drilled holes depends on the resource that was used for the drilling (in particular, on its precision rating), which is known only at runtime, i.e., during production. □

The mappings in $Maps$ combined with the two theories $\mathcal{D}_R$ and $\mathcal{D}_F$ form a new theory, defined below. This is not a traditional Situation Calculus theory since it includes two situation sorts (instead of one), which are completely independent from each other, namely: S_F for the facility information model $\mathcal{D}_F$, with initial situation $S_F^0 \in S_F$, and S_R for the cloud information system $\mathcal{D}_R$, with initial situation $S_R^0 \in S_R$.

For sort S_F we have that the value of fluents at each situation $s_F \in S_F$ is completely determined by $\mathcal{D}_F$. Defining S_R, instead, needs an additional effort, since the value of the fluents in Obs, after an action has been performed, depends only on the mappings in $Maps$. To handle this, we extend all action types $\mathbf{A} \in \mathcal{A}_R$ with an extra parameter s_F ranging over situations from S_F. Thus, every action $a(\mathbf{x})$ with parameters $\mathbf{x}$ is turned into a corresponding action $a(\mathbf{x}, s_F)$ with parameters $\mathbf{x}$ and s_F. We then make the following changes to $\mathcal{D}_R$:

- Every $\mathcal{D}_R$'s precondition axiom $Poss(\mathbf{A}(\mathbf{x}), s_R) \equiv \Phi(\mathbf{x}, s_R)$ is changed into

$$Poss(\mathbf{A}(\mathbf{x}, s_F), s_R) \equiv \Phi(\mathbf{x}, s_R)$$

that is, we ignore the newly introduced extra parameter.

- Every $\mathcal{D}_R$'s successor state axiom $f(\mathbf{x}, do(\mathbf{A}(\mathbf{y}), s_R)) \equiv \Phi(\mathbf{x}, \mathbf{y}, s_R)$, for fluent $f \notin \mathit{Obs}$ instantiated on the action $\mathbf{A}(\mathbf{y})$, is changed into

$$f(\mathbf{x}, do(\mathbf{A}(\mathbf{y}, s_F), s_R)) \equiv \Phi(\mathbf{x}, \mathbf{y}, s_R)$$

that is, as before, we ignore the extra parameter.

- For each fluent $f \in \mathit{Obs}$ with mapping $f(\mathbf{x}) \leftrightarrow \varphi_f(\mathbf{x})$, the following successor state axiom is defined:

$$f(\mathbf{x}, do(\mathbf{A}(\mathbf{y}, s_F), s_R)) \equiv \varphi_f(\mathbf{x}, s_F)$$

where the formula $\varphi_f(\mathbf{x}, s_F)$ is situation-invariant w.r.t. s_R, i.e., its value depends only on the extra parameter s_F added to action $\mathbf{A}$.

- Finally, the initial situation description $\mathcal{D}_R^0$ is changed by adding $f(\mathbf{x}, S_R^0) \equiv \varphi_f(\mathbf{x}, S_F^0)$ for each $f \in \mathit{Obs}$, using, again, the mapping $f(\mathbf{x}) \leftrightarrow \varphi_f(\mathbf{x})$.

We denote the resulting theory as $\mathcal{D}_R^{Maps}$. Note that since the initial situation description of $\mathcal{D}_F$ is complete, so is that of $\mathcal{D}_R^{Maps}$; hence, by distinguishing the two situation sorts S_R and S_F for the situations, it follows that $\mathcal{D}_R^{Maps}$ is *categorical*, i.e., admits essentially a unique model.

Among the situations of $\mathcal{D}_R^{Maps}$ in S_R, we are interested only in those that correspond to actual executions in $\mathcal{D}_F$, according to the mappings $Maps$. In other words, we need to single out the situations $s_R \in S_R$ and $s_F \in S_F$ that correspond to each other, i.e., which are *synchronized*. These can be defined by induction as the *smallest* predicate Syn such that:

- (Base case) $\quad Syn(S_R^0, S_F^0)$;
- (Inductive case) $\quad Syn(s_R, s_F) \wedge \bigwedge_{\mathbf{A} \in \mathcal{A}_R} \forall \mathbf{x}, s'_F. Do(\delta_{\mathbf{A}}(\mathbf{x}), s_F, s'_F) \wedge \bigwedge_{f \notin \mathit{Obs}} f(\mathbf{x}, do(\mathbf{A}(\mathbf{x}, s'_F), s_R)) \equiv \varphi_f(\mathbf{x}, s'_F) \supset Syn(do(\mathbf{A}(\mathbf{x}, s'_F), s_R), s'_F)$.

Intuitively, this definition says that S_R^0 and S_F^0 are synchronized, and that if s_R and s_F are synchronized and we do action $\mathbf{A}(\mathbf{x})$ in s_R and, correspondingly, we execute $\delta_{\mathbf{A}}(\mathbf{x})$ in s_F, then the resulting situations $do(\mathbf{A}(\mathbf{x}, s'_F), s_R)$ and s'_F are synchronized, as long as the successor-state axioms for non-observation fluents are satisfied. We are indeed interested in those situations $s_R \in S_R$ such that there exists a s_F such that $Syn(s_R, s_F)$, i.e., $\{s_R \in S_R \mid \exists s_F. Syn(s_R, s_F)\}$.

5.6. Facility

Given the cloud information model, a facility information model, a facility process and a set of mappings that are represented, respectively, as BAT$^-$ $\mathcal{D}_R$, a BAT $\mathcal{D}_F$, a ConGolog program δ_F^0 and mappings $Maps$, a facility is formalized by the tuple $Fac = \langle \mathcal{D}_R, \mathcal{D}_F, \delta_F^0, Maps \rangle$. We denote by AC_{Fac} the set of active object constants of Fac, that is either in $\mathcal{D}_R$ or in $\mathcal{D}_F$, i.e., $AC_{Fac} = AC_{\mathcal{D}_R} \cup AC_{\mathcal{D}_F}$.

5.7. Recipes

A recipe specifies the possible way(s) in which a product can be manufactured and is captured by a ConGolog program δ_R^0 for the action theory $\mathcal{D}_R$, i.e., the cloud information model. As such, δ_R^0 can mention only constants from $AC_{\mathcal{D}_R}$. This is compliant with the MaaS paradigm, where recipes are resource independent [39], i.e., specified using actions $\mathcal{A}_R$ and fluents $\mathcal{F}_R$ common throughout the manufacturing cloud rather than by adopting the action theory of a specific facility.

Example 4 (Recipe). A simple example of a process recipe is given below. This recipe makes use of the observation *precision* already discussed and the actions $\text{LOAD}(part, weight, material, dimx, dimy, dimz)$, $\text{DRILL}(part, dmtr, speed, x, y, z)$, $\text{APPLY\_GLUE}(part, glue\_type)$, $\text{RIVET}(part, x, y, z)$, $\text{PLACE}(part1, part2, part, x, y, z)$, $\text{REAMING}(part, dmtr, x, y, z)$ and $\text{STORE}(part, code)$, all in $\mathcal{A}_R$. As expected, these are not the same actions available in the facility (although the names and arguments are similar, for simplicity). We have commented in Example 3 on the mapping for DRILL.

According to this recipe, two steel parts denoted by `b` and `f` are loaded, a hole is drilled in `f`, then glue is applied to `b`. This is then placed on `f`, resulting in a composite part `fb`. The loading of `b` and the drilling of `f` can occur in any order, but glue must be applied to `f` before `b` is placed. If the resource used for drilling is not high-precision (which is an observation – see Example 3), a reaming operation is performed on the hole. Finally a rivet is applied to the hole and `fb` is stored away, with a `ok` code.

$$
\begin{array}{l}
\text{LOAD}(f, 4, steel, 810, 756, 29)\ ; \\
(\text{LOAD}(b, 2, steel, 312, 23, 20) \parallel \text{DRILL}(f, .3, 200, 123, 89, 21))\ ; \\
\text{APPLY\_GLUE}(b, str\_adh)\ ;\ \text{PLACE}(b, f, fb, 7, 201, 29)\ ; \\
\textbf{if}\ \neg \mathit{precision}(\mathit{hole}(f, 123, 89, 21), \texttt{high})\ \textbf{do} \\
\quad \text{REAMING}(fb, .3, 123, 89, 21) \\
\text{RIVET}(fb, 123, 89, 21)\ ; \\
\text{STORE}(fb, ok)
\end{array}
$$

□

A recipe does not specify how the manufacturing operations should be executed because it is resource-independent. The specific implementation of these actions on a facility needs to be automatically synthesized.

6. Realizability

In this section we formally characterize the conditions under which we can say that a recipe is realizable by a facility. Intuitively, this requires that each abstract action executable by the recipe can be mapped to a (sequence of) executable compound actions of the facility. In other words, no matter how the recipe may be progressed, the facility always has a way of replicating it. Moreover, whenever the recipe may be completed, the facility is in a final (i.e., safe) state.

We denote by $\langle \delta, s \rangle$ the *configuration* of an arbitrary program δ in situation s. We use $\langle \delta_F, s_F \rangle$ to denote the current program δ_F and situation $s_F \in \mathcal{S}_F$ to which the facility process δ_F^0 evolved from situation S_F^0. Similarly we use $\langle \delta_R, s_R \rangle$ to denote the current program δ_R and situation $s_R \in S_R$ to which the recipe δ_R^0 evolved from situation S_R^0. Observe that S_R is the set of situations of $\mathcal{D}_R^{Maps}$, not of $\mathcal{D}_R$; indeed we need to

use the theory $\mathcal{D}_R^{Maps}$ to evolve situations of the facility, so as to take into account the evolution of fluents $f \in \mathit{Obs}$. Initially, these configurations are $\langle \delta_F^0, S_F^0 \rangle$ and $\langle \delta_R^0, S_R^0 \rangle$.

Now we are ready to formally capture what it means for a facility to always be able to replicate the actions of a recipe, i.e., when the recipe can be realized by the facility. Formally, a recipe δ_R in situation s_R can be realized by a facility δ_F in s_F if:

- **r0** for every non-observation fluent $f \notin Obs$ in $\mathcal{D}_R$ the value in situation s_R of $\langle \delta_R, s_R \rangle$ is compatible through the mapping $f(\mathbf{x}) \leftrightarrow \varphi_f(\mathbf{x})$ with the value of $\varphi_f(\mathbf{x})$ in situation s_F of $\langle \delta_F, s_F \rangle$; note that for $f \in Obs$ this is already guaranteed by the definition of $\mathcal{D}_R^{Maps}$;
- **r1** $\mathit{Final}(\delta_R, s_R)$ implies $\mathit{Final}(\delta_F, s_F)$: if the recipe can be legally terminated, then all the resources can (safely) terminate their execution; and
- **r2** for every possible executable abstract action $\mathbf{A}(\mathbf{x})$ from $\langle \delta_R, s_R \rangle$ in the recipe, there exists a (arbitrarily complex) program $\delta_{\mathbf{A}}(\mathbf{x})$, determined through the mapping $\mathbf{A}(\mathbf{x}) \leftrightarrow \delta_{\mathbf{A}}(\mathbf{x})$, that is executable from $\langle \delta_F, s_F \rangle$ to some $\langle \delta'_F, s'_F \rangle$ and which represents the *implementation* of the action $\mathbf{A}(\mathbf{x})$ in the facility. Crucially, for at least one such $\langle \delta'_F, s'_F \rangle$, $\langle \delta'_R, do(\mathbf{A}(\mathbf{x}, s'_F), s_R) \rangle$ is realized by $\langle \delta'_F, s'_F \rangle$.

Note that in **r2**, the recipe situation $do(\mathbf{A}(\mathbf{x}, s'_F), s_R)$, resulting from the execution of the abstract action $\mathbf{A}(\mathbf{x})$ in situation s_R, depends on the situation s'_F reached by the facility after the execution of $\delta_{\mathbf{A}}(\mathbf{x})$. This captures the synchronization of the recipe and the facility situations.

Formally, we define the notion of realizability by co-induction as the largest predicate R between recipe and facility configurations such that:

$$
\begin{aligned}
&\langle \delta_R, s_R \rangle R \langle \delta_F, s_F \rangle \supset \\
&\quad \textbf{r0} \quad \textstyle\bigwedge_{f \notin Obs} \forall \mathbf{x}. f(\mathbf{x}, s_R) \equiv \varphi_f(\mathbf{x}, s_F) \ \wedge \\
&\quad \textbf{r1} \quad \mathit{Final}(\delta_R, s_R) \supset \mathit{Final}(\delta_F, s_F) \wedge \\
&\quad \textbf{r2} \quad \textstyle\bigwedge_{\mathbf{A} \in \mathcal{A}_R} \forall \delta'_R, \mathbf{x}.\ \mathit{Trans}(\delta_R, s_R, \delta'_R, do(\mathbf{A}(\mathbf{x}, S_F^0), s_R)) \supset \\
&\qquad\qquad \exists \delta'_F, s'_F.\ \mathit{Trans}^*(\delta_F, s_F, \delta'_F, s'_F) \ \wedge \mathit{Do}(\delta_{\mathbf{A}}(\mathbf{x}), s_F, s'_F) \ \wedge \\
&\qquad\qquad\qquad \langle \delta'_R, do(\mathbf{A}(\mathbf{x}, s'_F), s_R) \rangle R \langle \delta'_F, s'_F \rangle
\end{aligned}
$$

where $\mathbf{A}(\mathbf{x}) \leftrightarrow \delta_{\mathbf{A}}(\mathbf{x})$ and $f(\mathbf{x}) \leftrightarrow \varphi_f(\mathbf{x})$ are the mappings in $Maps$. A relation R satisfying the conditions above is called *realizability relation*. We say that a facility configuration $\langle \delta_F, s_F \rangle$ *realizes* a recipe configuration $\langle \delta_R, s_R \rangle$, written $\langle \delta_R, s_R \rangle \preceq \langle \delta_F, s_F \rangle$, if there exists a realizability relation R such that $\langle \delta_R, s_R \rangle R \langle \delta_F, s_F \rangle$. It is easy to see that $\preceq$ is itself a realizability relation and, in fact, the largest one.

We observe that this definition bears some similarities with the definition of *simulation* in [40], however it takes into account that the value of some fluents in the cloud information model come from the facility and that these values are not controllable by the recipe itself.

To understand the above formula, observe that from the precondition axioms in $\mathcal{D}_R^{Maps}$, which ignore the facility situation arguments, we have that if $\mathit{Poss}(\mathbf{A}(\mathbf{x}, s'_F), s_R)$ for some s'_F then $\mathit{Poss}(\mathbf{A}(\mathbf{x}, s''_F), s_R)$ for all s''_F. Now considering

that $\mathit{Trans}(\mathbf{A}(\mathbf{x}, s'_F), s_R, \delta', s'_R) \equiv s'_R = do(\mathbf{A}(\mathbf{x}, s'_F), s_R) \wedge \mathit{Poss}(\mathbf{A}(\mathbf{x}, s'_F), s_R) \wedge \delta' = \epsilon$ we can suppress the situation argument from $\mathbf{A}(\mathbf{x}, s'_F)$ in programs δ_R and use simply $\mathbf{A}(\mathbf{x})$ as action term in *Trans*, by defining:

$$\begin{aligned}&\mathit{Trans}(\mathbf{A}(\mathbf{x}), s_R, \delta', s'_R) \equiv \\ &\quad \exists s'_F \in S_F.s'_R = do(\mathbf{A}(\mathbf{x}, s'_F), s_R) \wedge \mathit{Poss}(\mathbf{A}(\mathbf{x})[s'_F], s_R) \wedge \delta' = \epsilon\end{aligned}$$

Moreover if for some s'_F we have that $\mathit{Trans}(\delta_R, s_R, \delta'_R, do(\mathbf{A}(\mathbf{x}, s'_F), s_R))$ then for every s''_F we have $\mathit{Trans}(\delta_R, s_R, \delta'_R, do(\mathbf{A}(\mathbf{x}, s''_F), s_R))$.

Based on these considerations, in $\mathcal{D}_R^{Maps}$ we are allowed to use any situation term of $\mathcal{D}_F$ (in particular, the situation constant S_F^0) as a placeholder for the situation of $\mathcal{D}_F$, since such situation term does not affect the executability of $\mathbf{A}(\mathbf{x})$ at s_R nor the program δ'_R resulting after its execution.

Definition 1 (Realizability). We say that a recipe δ_R^0 is *realizable* by a facility $Fac = \langle \mathcal{D}_R, \mathcal{D}_F, \delta_F^0, Maps \rangle$ iff $\langle \delta_R^0, S_R^0 \rangle \preceq \langle \delta_F^0, S_F^0 \rangle$. ■

When δ_R^0 is realized by δ_F^0 then, at every step, given a ground abstract action $\mathbf{A}(\mathbf{x})$ selected by δ_R^0, an execution of the corresponding program $\delta_{\mathbf{A}}(\mathbf{x})$ exists that preserves the realizability relation $\preceq$, and at the end of which control is returned to the recipe for the selection of the next action. Notice that $\delta_{\mathbf{A}}(\mathbf{x})$ is nondeterministic in general, as ConGolog programs may include choices of arguments and nondeterministic branching. Nonetheless, the existence of the realizability relation guarantees that this is possible (similarly to [40], we assume that nondeterminism in δ_F^0 is "angelic").

Note that once we have that $\langle \delta_R^0, S_R^0 \rangle \preceq \langle \delta_F^0, S_F^0 \rangle$ we can evolve (in all possible ways) the two configurations $\langle \delta_R^0, S_R^0 \rangle$ and $\langle \delta_F^0, S_F^0 \rangle$ in a synchronized way so as to maintain them in the relation $\preceq$. Formally, we can define by induction the *smallest* predicate $SynC$, such that:

- (Base case) $SynC(\delta_R^0, S_R^0, \delta_F^0, S_F^0)$;
- (Inductive case)
$$\begin{aligned}&SynC(\delta_R, s_R, \delta_F, s_F) \wedge \\ &\quad \mathit{Trans}(\delta_R, s_R, \delta'_R, s'_R) \wedge s'_R = do(\mathbf{A}(\mathbf{x}, s'_F), s_R) \wedge \\ &\quad \mathit{Trans}^*(\delta_F, s_F, \delta'_F, s'_F) \wedge \langle \delta'_R, s'_R \rangle \preceq \langle \delta'_F, s'_F \rangle \supset \\ &\qquad SynC(\delta'_R, s'_R, \delta'_F, s'_F).\end{aligned}$$

This predicate $SynC$ can be seen as a refinement of the predicate Syn introduced above, in the precise sense given by the next proposition.

Proposition 1 (Synchronization). *Let $Fac = \langle \mathcal{D}_R, \mathcal{D}_F, \delta_F^0, Maps \rangle$ be a facility and δ_R^0 a recipe realized by the facility, i.e., such that $\langle \delta_R^0, S_R^0 \rangle \preceq \langle \delta_F^0, S_F^0 \rangle$. Then for all pairs of configurations $\langle \langle \delta_R, s_R \rangle, \langle \delta_F, s_F \rangle \rangle$ we have that $SynC(\delta_R, s_R, \delta_F, s_F) \supset Syn(s_R, s_F)$.*

Proof. By induction, immediate by the definitions. □

This proposition guarantees that in the execution of recipe δ_R^0 (from S_R^0), as long as we follow the relation $\preceq$, we only generate values for the fluents in $f \in \mathit{Obs}$ that come from corresponding concrete executions in the facility.

We define a controller as follows. Given the current configuration $\langle \delta_R, s_R \rangle$ of the recipe, the current facility configuration $\langle \delta_F, s_F \rangle$, and a new configuration $\langle \delta'_R, s'_R \rangle$ specifying the next recipe configuration that may result from the *one-step* execution of δ_R through some action $\mathbf{A}(\mathbf{x})$, a controller returns a sequence of configurations $\langle \delta_F^0, s_F^0 \rangle \ldots \langle \delta_F^m, s_F^m \rangle$ of length $m \geq 0$ representing the steps that the facility must execute in order to complete $\delta_{\mathbf{A}}(\mathbf{x})$. We require that the recipe and facility configurations are in the realizability relation before and after executing $\mathbf{A}(\mathbf{x})$ and $\delta_{\mathbf{A}}(\mathbf{x})$.

Definition 2 (Controller). Given a facility $Fac = \langle \mathcal{D}_R, \mathcal{D}_F, \delta_F^0, Maps \rangle$ and a recipe δ_R^0 realizable by Fac, a *controller* for δ_F^0 that realizes δ_R^0 is a function ρ that, given two configurations $\langle \delta_R, s_R \rangle$ and $\langle \delta_F, s_F \rangle$ such that $\langle \delta_R, s_R \rangle \preceq \langle \delta_F, s_F \rangle$, an action $\mathbf{A}(\mathbf{x})$, and a program δ'_R such that $\mathit{Trans}(\delta_R, s_R, \delta'_R, \mathit{do}(\mathbf{A}(\mathbf{x}, S_F^0), s_R))$ (recall that S_F^0 is used as a placeholder, and does not affect δ'_R), returns a sequence of facility configurations $\langle \delta_F^0, s_F^0 \rangle \ldots \langle \delta_F^m, s_F^m \rangle$, such that:

- $\mathit{Trans}(\delta_F^i, s_F^i, \delta_F^{i+1}, s_F^{i+1})$ for $i \in [0, m-1]$, and $\delta_F^0 = \delta_F$ and $s_F^0 = s_F$, that is, the sequence is executable in the facility;
- $\mathit{Do}(\delta_{\mathbf{A}}(\mathbf{x}), s_F, s_F^m)$, that is, the situation s_F^m is the result of executing the concrete program $\delta_{\mathbf{A}}(\mathbf{x})$ corresponding to $\mathbf{A}(\mathbf{x})$ from s_F;
- $\langle \delta'_R, \mathit{do}(\mathbf{A}(\mathbf{x}, s_F^m), s_R) \rangle \preceq \langle \delta_F^m, s_F^m \rangle$, that is, realizability between the resulting programs is preserved. ■

In other words a controller is a function that, given a recipe and a facility configuration such that $\langle \delta_R, s_R \rangle \preceq \langle \delta_F, \delta_F \rangle$, an action $\mathbf{A}(\mathbf{x})$ and the remaining program δ'_R after action execution, returns the witnesses for the realizability, i.e., a sequence of steps to progress to next recipe and facility configurations in such a way that $\langle \delta'_R, \mathit{do}(\mathbf{A}(\mathbf{x}, s_F^m), s_R) \rangle \preceq \langle \delta_F^m, s_F^m \rangle$. As a result, given the initial recipe and facility configurations $\langle \delta_R^0, S_R^0 \rangle$ and $\langle \delta_F^0, S_F^0 \rangle$, for every possible evolution of $\langle \delta_R^0, S_R^0 \rangle$ as determined by the choice of actions $\mathbf{A}(\mathbf{x})$ at each point, a controller ρ produces a corresponding evolution of $\langle \delta_F^0, S_F^0 \rangle$ that fulfills the realizability requirements, and hence determines the sequence of concrete configurations that the facility must traverse in order to realize the recipe. Finally, observe that any evolution determined by any controller ρ defined as above is such that $SynC$ (and hence Syn) holds.

7. Controller Synthesis

To check whether a realizability relation exists and, if so, build a controller, we resort to model checking for a variant of the (modal) μ-calculus in [41, 35], which we call $\mu\mathcal{L}_c$, interpreted over *game arenas* (GA), which are special (labelled) transition systems (TSs) capturing the rules of a turn-based game between two players, ENVIRONMENT and CONTROLLER.

A *(first-order) vocabulary* is a pair $\sigma = \langle \mathcal{F}, AC \rangle$, where $\mathcal{F}$ and AC are sets of, respectively, fluents and active (object) constants. Given an interpretation domain Δ with standard names, such that $AC \subseteq \Delta$, we denote by $\mathcal{I}_\Delta^\sigma$ the set of all possible interpretations of $\mathcal{F}$ and AC over Δ that interpret all constants from AC as themselves.

Definition 3 (Game arena). Given a vocabulary $\sigma = \langle \mathcal{F}, AC \rangle$, let:

- *turnCtrl* and *turnEnv* be special 0-ary fluents (i.e., propositions) not in $\mathcal{F}$;
- $\mathcal{F}_{\mathcal{T}} = \mathcal{F} \cup \{\textit{turnCtrl}, \textit{turnEnv}\}$;
- $\sigma_{\mathcal{T}} = \langle \mathcal{F}_{\mathcal{T}}, AC \rangle$.

A *game arena* over σ is a tuple $\mathcal{T} = \langle \Delta_{\mathcal{T}}, Q, q_0, \rightarrow, \mathcal{I} \rangle$, where:

- $\Delta_{\mathcal{T}}$ is the GA object domain, with standard names, such that $AC \subseteq \Delta_{\mathcal{T}}$;
- Q is the set of GA states;
- $q_0 \in Q$ is the GA initial state;
- $\rightarrow \subseteq Q \times Q$ is the GA transition relation;
- $\mathcal{I} : Q \mapsto \mathcal{I}^{\sigma_{\mathcal{T}}}_{\Delta_{\mathcal{T}}}$ is a labeling function associating each state $q \in Q$ with an interpretation $\mathcal{I}(q) = \langle \Delta_{\mathcal{T}}, \cdot^{\mathcal{I}(q)} \rangle \in \mathcal{I}^{\sigma_{\mathcal{T}}}_{\Delta_{\mathcal{T}}}$, such that exactly one among *turnCtrl* and *turnEnv* is true (and all constants in AC are interpreted as themselves). ■

We observe that the definition of game arena is consistent with the definition of labelled transition system in [35], thus all related results in [35] are applicable here (provided the respective hypotheses hold).

Given a facility $Fac = \langle \mathcal{D}_R, \mathcal{D}_F, \delta^0_F, Maps \rangle$ and a recipe δ^0_R, we next define the GA $\mathcal{T}$ *induced by* Fac and δ^0_R. This GA essentially captures those asynchronous executions of δ^0_F over $\mathcal{D}_F$ and δ^0_R over $\mathcal{D}^{Maps}_R$ such that the fluents of $\mathcal{D}^{Maps}_R$ are correctly synchronized with those of $\mathcal{D}_F$, as per the mappings (over fluents, not actions) in $Maps$.

In defining $\mathcal{T}$, it will be convenient to use a different, yet equivalent, representation of programs, to separate the assignments of "pick variables" to domain objects (resulting from the nondeterministic choice of arguments), i.e., the *data*, from the control flow, i.e., the *program counter* [42]. This will simplify proving our results. The new representation, which we call *split representation*, has the advantage of making explicit the structure of programs and, in particular, of isolating the data as the *only* source of infiniteness of a program's closure. Indeed, since the ConGolog programs we consider here are not recursive, they yield, when executed, only finitely many program counters; on the other hand, for the finitely many program variables, there exist, in general, infinitely many possible assignments to distinct domain objects. Since programs are obtained by combining program counters with assignments, they are infinitely many only as a consequence of the infinite number of possible assignments to pick variables.

We represent a program δ^0 as the pair $\langle \delta, \mathbf{x} \rangle$, where δ denotes its current program counter (defined below) and $\mathbf{x} = \langle x_1, \ldots, x_k \rangle$ is a tuple of object terms such that each x_i is the domain object currently assigned to the i-th pick variable of δ^0 (we assume an ordering of simple actions and parameters). We call $\mathbf{x}$ the (current) *environment*. Note that the new representation is merely a syntactic variant: as shown in [42], we can reconstruct the original program δ^0 by replacing the free pick variables of δ by the object terms assigned to the variables $\mathbf{x}$. This is denoted by writing $\delta[\mathbf{x}]$. Indeed, while

the set of possible environments remains infinite along a computation, at each state the environment term, consisting of a single tuple of arity k, maintains only a bounded number of values (smaller than the size of the programs).

The set of possible program counters for a program is formalized as the program's *syntactic closure*, defined by extending the definition in [42] to the new operator $|||$.

Definition 4 (Syntactic closure of a program). The *syntactic closure of a program* δ^0 is the set $\underline{\Gamma}_{\delta^0}$ inductively defined as follows:

1. $\delta^0, \epsilon \in \underline{\Gamma}_{\delta^0}$;
2. if $\delta_1; \delta_2 \in \underline{\Gamma}_{\delta^0}$ and $\delta_1' \in \underline{\Gamma}_{\delta_1}$ then $\delta_1'; \delta_2 \in \underline{\Gamma}_{\delta^0}$ and $\underline{\Gamma}_{\delta_2} \subseteq \underline{\Gamma}_{\delta^0}$;
3. if $\delta_1 \mid \delta_2 \in \underline{\Gamma}_{\delta^0}$ then $\underline{\Gamma}_{\delta_1}, \underline{\Gamma}_{\delta_2} \subseteq \underline{\Gamma}_{\delta^0}$;
4. if $\pi z.\delta \in \underline{\Gamma}_{\delta^0}$ then $\underline{\Gamma}_{\delta} \subseteq \underline{\Gamma}_{\delta^0}$;
5. if $\delta^* \in \underline{\Gamma}_{\delta^0}$ then $\delta; \delta^* \in \underline{\Gamma}_{\delta^0}$;
6. if $\delta_1 \| \delta_2 \in \underline{\Gamma}_{\delta^0}$ and $\delta_1' \in \underline{\Gamma}_{\delta_1}$ and $\delta_2' \in \underline{\Gamma}_{\delta_2}$ then $\delta_1' \| \delta_2' \in \underline{\Gamma}_{\delta^0}$;
7. if $\delta_1 ||| \delta_2 \in \underline{\Gamma}_{\delta^0}$ and $\delta_1' \in \underline{\Gamma}_{\delta_1}$ and $\delta_2' \in \underline{\Gamma}_{\delta_2}$ then $\delta_1' ||| \delta_2' \in \underline{\Gamma}_{\delta^0}$. ■

Observe that since the environment is separated from the program counter, the syntactic closure $\underline{\Gamma}_{\delta^0}$ of a program δ^0 is always a finite set. We stress that the split representation is just a syntactic variant of program representation, equivalent to the standard one, so we can freely switch between them without affecting the results.

The GA is a turn-based game between two players, called CONTROLLER (CTRL) and ENVIRONMENT (ENV, not to be confused with the variable environment used above for programs), who take care of progressing, respectively, the recipe and the factory. Turns are not strictly alternating. The components of the GA $\mathcal{T} = \langle \Delta_{\mathcal{T}}, Q, q_0, \rightarrow, \mathcal{I} \rangle$ induced by Fac and δ_R^0 are detailed below.

Vocabulary. For the vocabulary $\sigma = \langle \mathcal{F}, AC \rangle$, we define:

- $\mathcal{F} = \mathcal{F}_R \cup \mathcal{F}_F \cup \{$*pcR*, *envR*, *Act*, *xAct*, *pcF*, *envF*, *pcA*, *envA*, *finalEnv*, *finalCtrl*, *finalA*$\}$, where fluents in $\mathcal{F}_R$ and $\mathcal{F}_F$ have the situation argument suppressed, while the remaining fluents are added for book-keeping (their role will be become clear when we consider the labeling function for states). Specifically, *pcR*, *Act*, *pcF*, and *pcA* are unary; the arity of *envF* and *envR* matches the number of pick variables in δ_F^0 and δ_R^0, respectively, the arity of *xAct* is the maximum number N_v of variables $\mathbf{x}$ among all actions $\mathbf{A} \in \mathcal{A}_R$, and the arity of *envA* is $N_v + N_p$, with N_p the maximum number of pick variables among all programs $\delta_{\mathbf{A}}(\mathbf{x}) \in P$, with P the set of programs $\delta_{\mathbf{A}}(\mathbf{x})$ such that $\mathbf{A}(\mathbf{x}) \leftrightarrow \delta_{\mathbf{A}}(\mathbf{x}) \in Maps$, for some $\mathbf{A}$; *finalEnv*, *finalCtrl*, and *finalA* are propositions;
- $AC = AC_{\mathcal{D}_F} \cup AC_{\mathcal{D}_R} \cup \mathcal{A}_R \cup \underline{\Gamma}_{\delta_R^0} \cup \underline{\Gamma}_{\delta_F^0} \cup (\bigcup_{\delta \in P} \underline{\Gamma}_{\delta})$, for P as above.

Observe that actions from $\mathcal{D}_R$ and program counters act as active constants (as well as objects). We require that *pcR*, *Act*, *pcF*, and *pcA* are sorted predicates that can take only objects from, respectively, $\Gamma_{\delta_R^0}$, $\mathcal{A}_R$, $\Gamma_{\delta_F^0}$, and $\bigcup_{\delta \in P} \underline{\Gamma}_{\delta}$, which, thus, act as sorts. No other predicate can take objects from these sorts.

Object domain. The GA object domain is $\Delta_{\mathcal{T}} = \Delta \cup \underline{\Gamma}_{\delta_R^0} \cup \mathcal{A}_R \cup \underline{\Gamma}_{\delta_F^0} \cup (\bigcup_{\delta \in P} \underline{\Gamma}_{\delta})$. Recall that $\Delta_{\mathcal{T}}$ has standard names, thus, in particular, program counters and actions act as both (active) constants and objects.

Set of states. Let $\mathrm{C}_{\delta,s}$ be the set of configurations to which an arbitrary program δ can evolve, starting from situation s. We let $\mathrm{C}_{\delta_R^0} = \mathrm{C}_{\delta_R^0, S_R^0}$, $\mathrm{C}_{\delta_F^0} = \mathrm{C}_{\delta_F^0, S_F^0}$, and $\mathrm{C}_P = \bigcup_{\delta \in P, s_F \in S_F} \mathrm{C}_{\delta, s_F}$, for P as above (recall that these sets are defined over situations of $\mathcal{D}_R^{Maps}$). Notice that programs in configurations are actual programs, with actual parameters assigned to variables, i.e., they are not split into program counter and variable environment; consequently, the configuration spaces defined above are, in general, infinite. The set of states $Q \subseteq \{\textsc{Env}, \textsc{Ctrl}\} \times \mathrm{C}_{\delta_R^0} \times \mathrm{C}_{\delta_F^0} \times (\bigcup_{\delta \in P, s \in S_F} \mathrm{C}_{\mathcal{D}_F, \delta, s})$ is defined by mutual induction with the transition relation $\rightarrow$ (see *transition relation* below). Each state $q = \langle \vartheta, \langle \delta_R, s_R \rangle, \langle \delta_F, s_F \rangle, \langle \delta, s \rangle \rangle$ is such that $s = s_F$. For notational convenience, we omit s and avoid tuple nesting, using the following equivalent representation: $q = \langle \vartheta, \delta_R, s_R, \delta_F, s_F, \delta \rangle$. Each state captures a configuration of the GA, where: ϑ represents which player is next to move; $\langle \delta_R, s_R \rangle$ stands for the current recipe configuration; $\langle \delta_F, s_F \rangle$ is the current facility configuration; and $\langle \delta, s \rangle$ is the current configuration of the program $\delta_{\mathbf{A}}(\mathbf{x}) \in P$ associated by $Maps$ with the (most recent) action $\mathbf{A}$ executed by δ_R. The definition of the transition relation specifies how these components evolve as the game proceeds. Notice that Q is in general infinite, as so are the configuration spaces over which it is defined.

Initial state. For the initial state, we let $q_0 = \langle \textsc{Env}, \delta_R^0, S_R^0, \delta_F^0, S_F^0, \epsilon \rangle$. This captures the situation where: ENVIRONMENT is to move next; the recipe has not started yet (thus no action was executed); the factory program has not started yet; and no program from P has been selected (as no action has been executed).

Transition relation. The transition relation $\rightarrow \subseteq Q \times Q$ is defined by mutual induction with the set of states Q, as follows:

- $q_0 \in Q$;
- for $q = \langle \vartheta, \delta_R, s_R, \delta_F, s_F, \delta \rangle$ and $q' = \langle \vartheta', \delta_R', s_R', \delta_F', s_F', \delta' \rangle$, if
  - $\vartheta = \textsc{Env} \wedge \mathit{Trans}(\delta_R, s_R, \delta_R', s_R') \wedge \exists \mathbf{A}, \mathbf{x}.\ s_R' = \mathit{do}(\mathbf{A}(\mathbf{x}, S_F^0), s_R) \wedge \delta' = \delta_{\mathbf{A}}(\mathbf{x}) \wedge \delta_F' = \delta_F \wedge s_F' = s_F \wedge \vartheta' = \textsc{Ctrl}$, or
  - $\vartheta = \textsc{Ctrl} \wedge \delta_R' = \delta_R \wedge \exists \mathbf{A}, \mathbf{x}, s_F'', s_R''.\ s_R = \mathit{do}(\mathbf{A}(\mathbf{x}, s_F''), s_R'') \wedge s_R' = \mathit{do}(\mathbf{A}(\mathbf{x}, s_F'), s_R'') \wedge \mathit{Trans}(\delta_F, s_F, \delta_F', s_F') \wedge \mathit{Trans}(\delta, s_F, \delta', s_F') \wedge (\vartheta' = \textsc{Env} \rightarrow \mathit{Final}(\delta', s_F'))$,

  then $q' \in Q$ and $q \rightarrow q'$.

Transitions differ based on which player moves. ENVIRONMENT selects an action $\mathbf{A}(\mathbf{x})$, together with the corresponding program $\delta_{\mathbf{A}}(\mathbf{x})$, from those made available by the recipe δ_R (initially δ_R^0) in the current configuration; ENVIRONMENT then advances the recipe configuration and the cloud situation s_R of $\mathcal{D}_R^{Maps}$, consistently with the chosen action, and finally passes the turn to CONTROLLER. Notice that s_R is advanced according

to the initial facility situation, i.e., $s'_R = do(\mathbf{A}(\mathbf{x}, S^0_F), s_R)$. Recall that $\mathcal{D}_F$'s situation argument of $\mathbf{A}$ does not affect the interpretation of $\mathcal{D}_R$'s non-observation fluents.

CONTROLLER chooses one among the actions that are currently legal for both δ_F (initially δ^0_F) and δ in their current configuration. Observe that δ is assigned to $\delta_{\mathbf{A}}(\mathbf{x})$ by ENVIRONMENT at its turn. After selecting the action, CONTROLLER advances δ, δ_F, and s_F, consistently with the chosen action. This step corresponds to one execution step of δ, i.e., one step of the implementation of action $\mathbf{A}(\mathbf{x}, \cdot)$ previously selected by ENVIRONMENT, which is actually realized by the factory. In addition, CONTROLLER aligns the current cloud situation s_R with the resulting factory situation s'_F (recall that the interpretation of $\mathcal{D}_R$'s non-observation fluents is not affected by the $\mathcal{D}_F$'s situation argument in $\mathbf{A}$). The situation s''_R where ENVIRONMENT chose the action $\mathbf{A}(\mathbf{x}, \cdot)$ currently under execution is retrieved, and the cloud situation is assigned to $s'_R = do(\mathbf{A}(\mathbf{x}, s'_F), s''_R)$. Notice that s'_R is the situation resulting from the execution of $\mathbf{A}(\mathbf{x})$ at s''_R, given the factory situation s'_F. CONTROLLER can (but does not have to) return the turn to ENVIRONMENT only when δ has reached a final configuration.

Labeling function. Let $q = \langle \vartheta, \delta_R, s_R, \delta_F, s_F, \delta \rangle \in Q$ be a generic state of $\mathcal{T}$:

- for every $f \in \mathcal{F}_F$, $f^{\mathcal{I}(q)}(\mathbf{x})$ iff $\mathcal{D}_R^{Maps} \models f(\mathbf{x}, s_F)$;
- for every $f \in \mathit{Obs}$, $f^{\mathcal{I}(q)}(\mathbf{x})$ iff $\mathcal{D}_R^{Maps} \models \varphi_f(\mathbf{x}, s_F)$;
- for every $f \in \mathcal{F}_R \setminus \mathit{Obs}$, $f^{\mathcal{I}(q)}(\mathbf{x})$ iff $\mathcal{D}_R^{Maps} \models f(\mathbf{x}, s_R)$;
- $\mathit{pcR}^{\mathcal{I}(q)} = \tilde{\delta_R}$, $\mathit{envR}^{\mathcal{I}(q)} = \{\tilde{\mathbf{x}}\}$, where $\langle \tilde{\delta_R}, \mathbf{x} \rangle$ is the split representation of δ_R into program counter and environment: $\tilde{\mathbf{x}}$ is as $\mathbf{x}$, possibly extended with trailing null values to match the arity of *envR*. For notational convenience we omit the $\tilde{}$ symbol in the following, implicitly assuming that $\mathbf{x}$ is suitably extended when needed; notice that both *pcR* and *envR* are interpreted as singletons;
- for *pcF* and *envF*, and for *pcA* and *envA*, we proceed as above but considering, respectively, δ_F and δ; the resulting interpretations are, again, singletons;
- $\mathit{Act}^{\mathcal{I}(q)} = \{\mathbf{A}\}$ and $\mathit{xAct}^{\mathcal{I}(q)} = \{\mathbf{x}\}$ if $\exists s'_R, s'_F . s_R = do(\mathbf{A}(\mathbf{x}, s'_F), s'_R)$, and $\mathit{Act}^{\mathcal{I}(q)} = \emptyset$, $\mathit{xAct}^{\mathcal{I}(q)} = \emptyset$ otherwise;
- finally, we have that $\mathit{finalEnv} \equiv \mathit{Final}(\delta_R, s_R)$, $\mathit{finalCtrl} \equiv \mathit{Final}(\delta_F, s_F)$, and $\mathit{finalA} \equiv \mathit{Final}(\delta, s_F)$.

The labeling function provides the interpretation of the fluents in $\mathcal{F}$, plus additional information about game turn, recipe action under execution, and configuration of the involved programs, associated with the current state of the game. This information is used to interpret $\mu\mathcal{L}_c$ formulas on $\mathcal{T}$ (labelings retain all the relevant information).

Observe that $\mathcal{T}$ captures the moves available to ENVIRONMENT and CONTROLLER, but not the goal of the game. Such moves essentially correspond to: *(i)* execution steps (action executions) of the recipe process δ^0_R for ENVIRONMENT; *(ii)* execution steps of the factory process δ^0_F for CONTROLLER. ENVIRONMENT moves first, and when CONTROLLER returns the turn to ENVIRONMENT, a complete execution of the program associated with the last action selected by ENVIRONMENT has been correctly completed.

7.1. Controller Synthesis from GAs

$\mu\mathcal{L}_c$ model checking can be used both to compute a realizability relation between the recipe and the facility, and to synthesize the corresponding controller. Formulas of $\mu\mathcal{L}_c$ have the following syntax:

$$\Phi := \phi \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid \langle - \rangle \Phi \mid Z \mid \mu Z.\Phi \mid \nu Z.\Phi,$$

where: ϕ is a FO sentence with predicates and (active) constants from a given vocabulary $\sigma = \langle \mathcal{F}, AC \rangle$; the modal operator $\langle - \rangle \Phi$ denotes the existence of a transition from the current state to a state where Φ holds; we use the abbreviation $[-]\Phi$ for $\neg\langle - \rangle \neg\Phi$; Z is a second-order (SO) predicate variable over sets of states, and $\mu Z.\Phi$ and $\nu Z.\Phi$ denote the least and greatest fixpoints, respectively, with Φ seen as a predicate transformer with respect to Z. By the semantics below, one can see that the only interesting formulas are those closed w.r.t. to SO (in addition to FO) variables. In fact, SO variables are needed only for technical reasons, to make the fixpoint constructs available.

Note that $\mu\mathcal{L}_c$ is a (strict) sub-language of the language $\mu\mathcal{L}_p$ defined in [35]. Specifically, $\mu\mathcal{L}_c$ disallows quantification *across-states*, i.e., the possibility of relating objects occurring in different states. As a consequence, all the results obtained for $\mu\mathcal{L}_p$ are directly applicable to $\mu\mathcal{L}_c$.

Given a GA $\mathcal{T} = \langle \Delta_{\mathcal{T}}, Q, q_0, \rightarrow, \mathcal{I} \rangle$ over a vocabulary $\sigma = \langle \mathcal{F}, AC \rangle$, $\mu\mathcal{L}_c$ formulas over $\mathcal{T}$ are defined over the vocabulary $\sigma_{\mathcal{T}}$ (which, by Definition 3, is obtained from σ by extending $\mathcal{F}$ with *turnCtrl* and *turnEnv*). The semantics of a $\mu\mathcal{L}_c$ formula Φ over $\mathcal{T}$ is inductively defined as follows, where v is an assignment from SO variables to sets of states:

$$\begin{array}{l}
(\phi)^{\mathcal{T}} = \{q \mid q \in Q \text{ and } \mathcal{I}(q) \models \phi\} \\
(\neg\Phi)^{\mathcal{T}}_v = Q \setminus (\Phi)^{\mathcal{T}}_v \\
(\Phi_1 \wedge \Phi_2)^{\mathcal{T}}_v = (\Phi_1)^{\mathcal{T}}_v \cap (\Phi_2)^{\mathcal{T}}_v \\
(\langle - \rangle \Phi)^{\mathcal{T}}_v = \{q \mid \exists q', q \rightarrow q', q' \in (\Phi)^{\mathcal{T}}_v\} \\
(Z)^{\mathcal{T}}_v = v(Z) \\
(\mu Z.\Phi)^{\mathcal{T}}_v = \bigcap\{\mathcal{E} \subseteq Q \mid (\Phi)^{\mathcal{T}}_{v[Z/\mathcal{E}]} \subseteq \mathcal{E}\} \\
(\nu Z.\Phi)^{\mathcal{T}}_v = \bigcup\{\mathcal{E} \subseteq Q \mid \mathcal{E} \subseteq (\Phi)^{\mathcal{T}}_{v[Z/\mathcal{E}]}\}
\end{array}$$

A state $q \in Q$ is said to *satisfy* a $\mu\mathcal{L}_c$ formula Φ (under a SO assignment v), if $q \in (\Phi)^{\mathcal{T}}_v$. We say that $\mathcal{T}$ satisfies Φ if $q_0 \in (\Phi)^{\mathcal{T}}_v$. Observe that when Φ is closed w.r.t. SO variables, as are formulas of practical interest, v becomes irrelevant. When not needed, we omit v from $(\cdot)^{\mathcal{T}}_v$, thus using $(\cdot)^{\mathcal{T}}$.

As we will show below, the satisfaction of the following $\mu\mathcal{L}_c$ formula by $\mathcal{T}$ implies the existence of a realizability relation between δ^0_R and δ^0_F:

$$\Phi_{Real} = \nu X.\mu Y.((\phi_{\text{OK}} \wedge [-]X) \vee (\textit{turnCtrl} \wedge \langle - \rangle Y)),$$

where $\phi_{\text{OK}} = \bigwedge_{f \in \mathcal{F}_R \setminus Obs} \forall \mathbf{x}. f(\mathbf{x}) \equiv \varphi_f(\mathbf{x}) \wedge \textit{turnEnv} \wedge (\textit{finalEnv} \supset \textit{finalCtrl})$, with $\varphi_f(\mathbf{x})$ being the situation-suppressed version of $\varphi_f(\mathbf{x}, s_F)$. Notice that fluents in the labeling preserve the same names as in $\mathcal{D}_F$ and $\mathcal{D}_R$, thus the situation-suppressed version of φ_f is defined over $\mathcal{T}$'s vocabulary σ. Intuitively, ϕ_{OK} holds in those states q of $\mathcal{T}$ where: *(i)* the interpretation of every fluent $f \in \mathcal{F}_R \setminus Obs$ in the labeling of q matches

the interpretation of the corresponding formula φ_f over the same labeling; *(ii)* it is ENVIRONMENT's turn; and *(iii)* if the recipe may terminate so can the facility. The formula Φ_{Real} is true in all those states from which CONTROLLER can force the game to visit infinitely many times a state where ϕ_{OK} holds, no matter how ENVIRONMENT moves in its turns. Φ_{Real} also requires that CONTROLLER does not pass the turn until ϕ_{OK} holds. The set *Win*(Φ_{Real}) of *winning states* is the set of states where Φ_{Real} holds.

Theorem 1. *Given a facility* $Fac = \langle \mathcal{D}_R, \mathcal{D}_F, \delta_F^0, Maps \rangle$*, a recipe* δ_R^0 *over* $\mathcal{D}_R$ *is realizable by the facility process* δ_F^0 *iff* $q_0 \in$ *Win*(Φ_{Real}).

Proof. For the *If-part*, assume that $q_0 = \langle \text{ENV}, \delta_R^0, S_R^0, \delta_F^0, S_F^0, \epsilon \rangle \in$ *Win*(Φ_{Real}). We prove that there exists a realizability relation R such that $\langle \delta_R^0, S_R^0 \rangle R \langle \delta_F^0, S_F^0 \rangle$. Since $q_0 \in$ *Win*(Φ_{Real}), for each possible ENVIRONMENT move $q_0 \rightarrow q_1$, CONTROLLER has a sequence of (legal) moves that take the GA $\mathcal{T}$ from q_1 to a state q such that $q \models \phi_{\text{OK}}$, and from which, for all possible ENVIRONMENT moves, CONTROLLER has a sequence of moves that take $\mathcal{T}$ to a state q' such that $q' \models \phi_{\text{OK}}$, and so on forever. In other words, whenever ENVIRONMENT moves, CONTROLLER can force the game to achieve a state where ϕ_{OK} holds. Notice that ϕ_{OK} implies that ENVIRONMENT is to move. By considering all possible ENVIRONMENT moves at each such state and, for each of them, a suitable CONTROLLER response sequence, we obtain a tree whose root is q_0, and where branches occur only at states where ϕ_{OK} holds, each corresponding to a possible ENVIRONMENT move. These infinite paths $q_0 q_1 \cdots$ correspond to plays of $\mathcal{T}$ and contain infinitely many states $q = \langle \text{ENV}, \delta_R, s_R, \delta_F, s_F, \delta \rangle$ such that $q \models \phi_{\text{OK}}$. For each path and each such state, we let $\langle \delta_R, s_R \rangle R \langle \delta_F, s_F \rangle$. Obviously, $\langle \delta_R^0, S_R^0 \rangle R \langle \delta_F^0, S_F^0 \rangle$.

To see that R is a realizability relation, consider an arbitrary pair of configurations from R, $\langle \delta_R, s_R \rangle$ and $\langle \delta_F, s_F \rangle$. By the definition of R, there exists a path $q_0 \cdots q_i \cdots$ containing a state $q_i = \langle \text{ENV}, \delta_R^i, s_R^i, \delta_F^i, s_F^i, \delta^i \rangle$, such that $q_i \models \phi_{\text{OK}}$, and $\delta_R^i = \delta_R$, $s_R^i = s_R$, $\delta_F^i = \delta_F$, $s_F^i = s_F$.

It is easy to see that the definition of R above satisfies requirements **r0** and **r1** of the realizability relation (p. 25). Indeed, these are consequences of the fact that $q_i \models \phi_{\text{OK}}$, which, in particular, requires that the interpretation of each non-observation fluent f of $\mathcal{D}_R$ matches that of φ_f and that *finalEnv* implies *finalCtrl*.

For requirement **r2**, consider an action $\mathbf{A}(\mathbf{x})$ legal in $\langle \delta_R, s_R \rangle$. By the definition of $\mathcal{T}$, and in particular the transition relation, ENVIRONMENT has a move $q_i \rightarrow q_{i+1}$, with $q_{i+1} = \langle \text{CTRL}, \delta_R^{i+1}, s_R^{i+1}, \delta_F^{i+1}, s_F^{i+1}, \delta^{i+1} \rangle$, where: *Trans*$(\delta_R^i, s_R^i, \delta_R^{i+1}, s_R^{i+1})$; $s_R^{i+1} =$ *do*$(\mathbf{A}(\mathbf{x}, S_F^0), s_R^i)$; $\delta_F^{i+1} = \delta_F^i$; $s_F^{i+1} = s_F^i$; and $\delta^{i+1} = \delta_{\mathbf{A}}(\mathbf{x})$. Thus, the sequence of states $q_0 \cdots q_i q_{i+1}$ is a path of the game and, since $q_0 \in$ *Win*(Φ_{Real}), CONTROLLER has a sequence of moves that extends the path with states $q_{i+2} \cdots q_{i+\ell}$, where $q_{i+\ell} \models \phi_{\text{OK}}$. Let $q_{i+j} = \langle \vartheta^{i+j}, \delta_R^{i+j}, s_R^{i+j}, \delta_F^{i+j}, s_F^{i+j}, \delta^{i+j} \rangle$ $(j = 2, \ldots, \ell)$. By the definition of $\mathcal{T}$, we have that: $\delta_R^{i+j} = \delta_R^{i+1}$; $s_R^{i+j} =$ *do*$(\mathbf{A}(\mathbf{x}, s_F^{i+j}), s_R^i)$; *Trans*$(\delta_F^{i+j-1}, s_F^{i+j-1}, \delta_F^{i+j}, s_F^{i+j})$; and *Trans*$(\delta^{i+j-1}, s_F^{i+j-1}, \delta^{i+j}, s_F^{i+j})$. Moreover, for $j = 2, \ldots, \ell - 1$, $\vartheta^{i+j} =$ CTRL, and $\vartheta^{i+\ell} =$ ENV. Thus, since the transition relation of $\mathcal{T}$ implies that $\vartheta^{i+\ell} =$ ENV only if *Final*$(\delta^{i+\ell}, s_F^{i+\ell})$, we have that *Final*$(\delta^{i+\ell}, s_F^{i+\ell})$. By the above, it follows that: *(i)* *Trans*$(\delta_R, s_R, \delta_R^{i+1}, do(\mathbf{A}(\mathbf{x}, S_F^0), s_R))$; *(ii)* *Trans*$^*(\delta_F, s_F, \delta_F^{i+\ell}, s_F^{i+\ell})$; *(iii)*

Do$(\delta_{\mathbf{A}}(\mathbf{x}), s_F, s_F^{i+\ell})$; and *(iv)* $\langle \delta_R^{i+\ell}, s_R^{i+\ell} \rangle R \langle \delta_F^{i+\ell}, s_F^{i+\ell} \rangle$, since $q^{i+\ell} \models \phi_{\text{OK}}$ and by the definition of R. Since $\mathbf{A}(\mathbf{x})$ is generic, requirement **r2** easily follows.

For the *OnlyIf-part* assume a realizability relation R such that $\langle \delta_R^0, S_R^0 \rangle R \langle \delta_F^0, S_F^0 \rangle$. Let $q_0 = \langle \text{ENV}, \delta_R^0, S_R^0, \delta_F^0, S_F^0, \delta^0 \rangle$, with $\delta^0 = \epsilon$, and consider a legal action $\mathbf{A}(\mathbf{x})$ from $\langle \delta_R^0, S_R^0 \rangle$, together with the mapped program $\delta_{\mathbf{A}}(\mathbf{x})$. Such an action defines an ENVIRONMENT move $q_0 \rightarrow q_1$ in $\mathcal{T}$, with $q_1 = \langle \text{CTRL}, \delta_R^1, s_R^1, \delta_F^1, s_F^1, \delta^1 \rangle$, such that: *(i)* *Trans*$(\delta_R^0, S_R^0, \delta_R^1, s_R^1)$; *(ii)* $s_R^1 = do(\mathbf{A}(\mathbf{x}, S_F^0), S_R^0)$; *(iii)* $\delta_F^1 = \delta_F^0$; *(iv)* $s_F^1 = S_F^0$; and *(v)* $\delta^1 = \delta_{\mathbf{A}}(\mathbf{x})$. By requirement **r2** of R, for each $\mathbf{A}(\mathbf{x})$, there exists a sequence of execution steps for the factory process δ_F that realizes the mapped program $\delta_{\mathbf{A}}(\mathbf{x})$. More formally, there exist two sequences $\langle \delta_F^1, s_F^1 \rangle \cdots \langle \delta_F^\ell, s_F^\ell \rangle$ and $\langle \delta^1, s_F^1 \rangle \cdots \langle \delta^\ell, s_F^\ell \rangle$ such that: *(i)* *Trans*$(\delta_F^i, s_F^i, \delta_F^{i+1}, s_F^{i+1})$ $(i = 1, \ldots, \ell - 1)$; *(ii)* *Trans*$(\delta^i, s_F^i, \delta^{i+1}, s_F^{i+1})$ $(i = 1, \ldots, \ell - 1)$; and *(iii)* $\langle \delta_R^1, do(\mathbf{A}(\mathbf{x}, s_F^\ell), S_R^0) \rangle R \langle \delta_F^\ell, s_F^\ell \rangle$. Using such sequences, we can extend $q_0 q_1$ to a sequence $q_0 q_1 \cdots q_\ell$ such that: *(i)* for $i = 2, \ldots, \ell - 1$, $q_i = \langle \text{CTRL}, \delta_R^1, do(\mathbf{A}(\mathbf{x}, s_F^i), S_R^0), \delta_F^i, s_F^i, \delta^i \rangle$; and *(ii)* $q_\ell = \langle \text{ENV}, \delta_R^1, do(\mathbf{A}(\mathbf{x}, s_F^\ell), S_R^0), \delta_F^\ell, s_F^\ell, \delta^\ell \rangle$. It is easy to check that the path so defined is indeed a path of $\mathcal{T}$, as it starts in the initial state q_0 and is consistent, at every step, with $\mathcal{T}$'s transition relation. The labeling of the states in the path can be obtained through the labeling function previously defined. With respect to this, by requirements **r0** and **r1** of R, it follows that $q_\ell \models \phi_{\text{OK}}$.

We have thus shown that for every move ENVIRONMENT can make in the initial state q_0, CONTROLLER has a way to force the game to reach a new state q_ℓ such that $q_\ell \models \phi_{\text{OK}}$. In proving this, we have used only the assumption that, for $q_0 = \langle \text{ENV}, \delta_R^0, S_R^0, \delta_F^0, S_F^0, \delta^0 \rangle$, $\langle \delta_R^0, S_R^0 \rangle R \langle \delta_F^0, S_F^0 \rangle$. But then, since for $q_\ell = \langle \text{ENV}, \delta_R^\ell, s_R^\ell, \delta_F^\ell, s_F^\ell, \delta^\ell \rangle$ we have that $\langle \delta_R^\ell, s_R^\ell \rangle R \langle \delta_F^\ell, s_F^\ell \rangle$, we can generalize the argument above and prove that for every ENVIRONMENT move $q_\ell \rightarrow q_{\ell+1}$, another state $q_{\ell+m}$ such that $q_{\ell+m} \models \phi_{\text{OK}}$ exists that can be reached after a suitable sequence of CONTROLLER moves, and so on forever. This ultimately proves that $q_0 \in \mathit{Win}(\Phi_{Real})$.□

Although a (possibly transfinite) fixpoint computation based on approximates provides a way to obtain $\mathit{Win}(\Phi_{Real})$, the number of approximates that we need to compute is bounded (by the size of the GA) only if $\mathcal{T}$ is finite. Since, in our case, $\mathcal{T}$ can be infinite, the fixpoint cannot be computed in general. Nonetheless, we show how a controller can be computed when the CONTROLLER player can win the game represented by $\mathcal{T}$, that is, when it has a *winning strategy*.

Let $\mathcal{T} = \langle \Delta_{\mathcal{T}}, Q, q_0, \rightarrow, \mathcal{I} \rangle$ be a GA (over a vocabulary σ). A *history* of $\mathcal{T}$ is a sequence $\tau = q_0 \cdots q_\ell \in Q^+$ such that, for $i \in [0, \ell - 1]$, $q_i \rightarrow q_{i+1}$. $\mathcal{H}$ denotes the set of histories of a GA. A CONTROLLER *strategy* is a function $\varsigma : \mathcal{H} \mapsto Q$ such that if $\varsigma(q_0 \cdots q_\ell) = q$ then $q_\ell \models$ *turnCtrl* and $q_\ell \rightarrow q$. A history $\tau = q_0 \cdots q_\ell$ is *induced* by a strategy ς if, for every $i \in [0, \ell - 1]$, whenever $q_i \models$ *turnCtrl*, $q_{i+1} = \varsigma(q_0 \cdots q_i)$. The strategies of interest are those, called *winning*, which enforce Φ_{Real}.

Definition 5 (Winning strategy). A strategy ς for player CONTROLLER is said to be *winning* for Φ_{Real} if, for every history $\tau = q_0 \cdots q_\ell$ induced by ς: $q_\ell \models$ *turnEnv* implies $q_\ell \models \phi_{\text{OK}}$; and $q_\ell \models$ *turnCtrl* implies that τ can be extended to a history $\tau' = q_0 \cdots q_\ell \cdots q_m$ induced by ς s.t. $q_m \models \phi_{\text{OK}}$ and $q_i \models$ *turnCtrl*, for $i = \ell, \ldots, m - 1$.

Intuitively, a strategy for CONTROLLER is winning for Φ_{Real} if CONTROLLER has a way to play in its turns such that, no matter how ENVIRONMENT moves in its turns, the game will end up to a state q_m where ϕ_{OK} holds, and from which this property is preserved.

While, in general, a strategy can prescribe different sequences of moves on histories that end in a same state, when a winning strategy exists, then there exists one that prescribes the same sequence on such histories. This is because we are essentially playing a model checking game over GA for a μ-calculus objective, and as a result the set of winning states does not depend on the history [41]. In what follows, we therefore focus only on *memoryless* strategies, i.e., strategies that depend only on the last state of the history. For this reason, a memoryless strategy $\varsigma : \mathcal{H} \mapsto Q$ can be represented as a function $\varsigma_m : Q \mapsto Q$, with $\varsigma(\tau) = \varsigma_m(last(\tau))$, where $last(\tau)$ denotes τ's last state.

Theorem 2. *A controller ρ for δ_F^0 that realizes δ_R^0 can be obtained from a (memoryless) winning strategy ς for Φ_{Real} and $\mathcal{T}$.*

The controller ρ can be obtained as follows. For every history $\tau = q_0 \cdots q_{\ell-1} q_\ell$ induced by ς, such that $q_{\ell-1} \models \phi_{\text{OK}}$ (hence, by $\mathcal{T}$'s transition relation, $q_\ell \models$ *turnCtrl*, $\delta_F^\ell = \delta_F^{\ell-1}$, and $s_F^\ell = s_F^{\ell-1}$), consider the sequence of states $q_{\ell+1}, \ldots, q_m$ obtained by iteratively applying ς, starting from τ, i.e., $q_{\ell+1} = \varsigma(q_0 \cdots q_\ell)$, $q_{\ell+2} = \varsigma(q_0 \cdots q_\ell q_{\ell+1})$, and so on, until a state q_m such that $q_m \models \phi_{\text{OK}}$ is obtained. Let $q_i = \langle \vartheta^i, \delta_R^i, s_R^i, \delta_F^i, s_F^i, \delta^i \rangle$ and consider the action $\mathbf{A}(\mathbf{x})$ associated with the concrete program δ^ℓ. We then define $\rho(\langle \delta_R^{\ell-1}, s_R^{\ell-1} \rangle, \langle \delta_F^\ell, s_F^\ell \rangle, \mathbf{A}(\mathbf{x}), \delta_R^\ell) := \langle \delta_F^{\ell+1}, s_F^{\ell+1} \rangle \cdots \langle \delta_F^m, s_F^m \rangle$.

It remains to establish how, and when, a memoryless winning strategy can be computed. Note that, as Q may be infinite, this may not be possible in general.

8. Bounded Case: Decidable Synthesis

ad, showing that a controller that realizes δ_R^0 can be effectively computed.

As a preliminary step, we study the problem of computing the winning set *Win*(Φ_{Real}) and a corresponding strategy for CONTROLLER on a generic GA defined over the same vocabulary σ as that of the induced GA, but with a finite set of states Q. This is a crucial step as we will reduce the problem of computing a controller for δ_F^0 that realizes δ_R^0 to that of computing *Win*(Φ_{Real}) and a strategy for CONTROLLER on a particular finite-state GA over σ. Observe that, in general, the induced GA itself cannot be finite-state, even if the object domain Δ is finite: the transition relation essentially embeds, through $\mathcal{D}_R^{Maps}$, the infinitely many situations of $\mathcal{D}_R$ and $\mathcal{D}_F$, thus yielding an infinite set of states.

On a finite-state GA $\mathcal{T}$, it is well known that μ-calculus fixpoints can be computed by iterative approximations (e.g., [41]). For a formula $\Phi = \nu X.\Psi(X)$, one starts with the initial approximate $X_0 = Q$ and then iteratively computes $X_i = (\Phi)^{\mathcal{T}}_{v[X/X_{i+1}]}$, until the fixpoint is reached, i.e., $X_n = X_{n-1}$. This is the desired fixpoint, i.e., $(\Phi)^{\mathcal{T}} = X_n$ (the SO assignment v is omitted as irrelevant). For formulas $\Phi = \mu X.\Psi(X)$, the same procedure can be used, but starting with the approximate $X_0 = \emptyset$. The winning set *Win*(Φ_{Real}) can thus be obtained by applying such procedures to the formula

$\Phi_{Real} = \nu X.\mu Y.((\phi_{\text{OK}} \wedge [-]X) \vee (\textit{turnCtrl} \wedge \langle - \rangle Y))$. During the computation, the following approximates are produced, which can be used to build the controller:

$$
\begin{array}{l}
X_i = Y_{(i-1)n_{(i-1)}} \text{ (initially, } X_0 = Q\text{)} \\
\quad Y_{i0} = \emptyset \\
\quad \cdots \\
\quad Y_{ij} = ((\phi_{\text{OK}} \wedge [-]X) \vee (\textit{turnCtrl} \wedge \langle - \rangle Y))^{\mathcal{T}}_{v[X/X_i, Y/Y_{i(j-1)}]} \\
\quad \cdots \\
X_{i+1} = Y_{in_i} \text{ (for } n_i \text{ the smallest index such that } Y_{in_i} = Y_{i(n_i+1)}\text{)} \\
\cdots \\
X_k = Y_{kn_k} \text{ (for } k \text{ the smallest index such that } X_k = X_{k+1}\text{)}
\end{array}
$$

The greatest fixpoint $\textit{Win}(\Phi_{Real}) = X_k = Y_{kn_k}$ is reached after computing a finite number k of approximates X_i, each requiring, in turn, to compute a finite number n_i of approximates Y_{ij}. Intuitively, Y_{ij} contains all those states $q \in Q$ such that either:

1. $q \models \phi_{\text{OK}}$ and no matter how ENVIRONMENT moves from q (remember ϕ_{OK} implies *turnEnv*), a state $q' \in X_i$ is reached after the move (first disjunct of Φ_{Real}); or
2. $q \models \textit{turnCtrl}$ and CONTROLLER can force the game to reach, in $m < j$ consecutive moves (and keeping the turn), a state $q' \in Q$ that satisfies property 1 (second disjunct of Φ_{Real}); in general, q may belong to many approximates $Y_{kj'}$, even with $j' < j$.

(Recall that *turnCtrl* and *turnEnv* are mutually exclusive in $\mathcal{T}$, thus so are the above properties.) Notice that, since $\textit{Win}(\Phi_{Real}) = X_k = Y_{kn_k}$, every state of $\textit{Win}(\Phi_{Real})$ fulfills either 1 or 2, for $i = k$ and $j = n_k$. In other words, for every state $q \in \textit{Win}(\Phi_{Real})$, if it is ENVIRONMENT's turn, no matter how ENVIRONMENT moves, CONTROLLER can force the game to reach, in a number $m < n_k$ of consecutive moves, a state $q' \in \textit{Win}(\Phi_{Real})$ such that ϕ_{OK} holds and from which a new state $q'' \in \textit{Win}(\Phi_{Real})$ where the same holds can be forced again, and so on and so forth. It is then easy to see that if $q_0 \in \textit{Win}(\Phi_{Real})$, CONTROLLER has a winning strategy for Φ_{Real} on $\mathcal{T}$. We now discuss how one such strategy can be obtained.

Given a state $q \in \textit{Win}(\Phi_{Real})$, we define $\varsigma_m(q)$. By construction of $\textit{Win}(\Phi_{Real})$, either 1 or 2 holds, for $i = k$ and $j = n_k$. In the former case, we leave $\varsigma_m(q)$ undefined, as $q \models \textit{turnEnv}$. In the latter case, instead, $q \models \textit{turnCtrl}$, thus $\varsigma_m(q)$ must be defined. Since $q \in \textit{Win}(\Phi_{Real}) = Y_{kn_k}$, CONTROLLER can force the game, in $m < n_k$ moves, to a state where 1 holds. Then, there must exist a transition $q \rightarrow q'$ such that $q' \in Y_{km}$ (i.e., from q', CONTROLLER can force property 1 in $m - 1$ moves). In other words, q' is one step "closer" to property 1. Thus, by defining $\varsigma_m(q) = q'$ for one such q', we guarantee progressing towards, and eventually achieve, property 1.

It remains to show how the desired q' can be selected. Indeed, from q, many possible CONTROLLER moves, i.e., successor states q' of q, are available, but only some of them progress towards 1. The question is then how to select one of them. To address this, during the fixpoint computation, we label each state $q \in \textit{Win}(\Phi_{Real})$ with its "distance from property 1", i.e., with the minimum number of moves CONTROLLER requires to force 1. To do this, we proceed as follows:

- when a new approximate Y_{i0} is initialized, all labels are removed;
- when an unlabelled state q enters an approximate Y_{ij} with $j > 0$, then we label q with $j - 1$.

Obviously, when the fixpoint is obtained, all states are left with the labeling defined during the last approximate computation. Property 1 and 2 guarantee that the so-obtained labeling corresponds to the distance of each state from property 1. Thus, given q, we can define $\varsigma_m(q) = q'$, for q' any state with minimal labeling among those such that $q \rightarrow q'$. Thus, for a finite-state GA, we have a technique to actually build a winning controller strategy. This will be useful later on.

Now, we relax the state-finiteness assumption and consider an induced, infinite-state, GA $\mathcal{T}$. In this case, constructing a controller is not possible in general. We show, however, how this can be done when the information kept in each state of the GA is "bounded" ([42]). Informally, a GA is state-bounded if all of its states are labelled by interpretations containing only a bounded number of objects (or, equivalently, a bounded number of distinct tuples). That is, in every state, the number of objects in the interpretation of all fluents is bounded by a given bound. We recall the corresponding formal definition below. Given a FO interpretation $\mathcal{I}$, the *active domain of* $\mathcal{I}$ is the set $adom(\mathcal{I})$ of all the objects that occur in the interpretation of some fluent in $\mathcal{I}$.

Definition 6 (State-boundedness). A GA $\mathcal{T} = \langle \Delta_{\mathcal{T}}, Q, q_0, \rightarrow, \mathcal{I} \rangle$ is said to be *state-bounded by* $b \in \mathbb{N}$ if $|adom(\mathcal{I}(q))| \leq b$, for every $q \in Q$. $\mathcal{T}$ is said to be *state-bounded* if it is state-bounded by b, for some b. ■

We can show that the induced GA $\mathcal{T}$ is state-bounded whenever $\mathcal{D}_F$ and $\mathcal{D}_R$ are "bounded", in the sense of the following definitions.

Following [18], for $b \in \mathbb{N}$ and a fluent f, we can write a FO formula $Bounded_{f,b}(s)$, to express that fluent f contains fewer than b distinct tuples at situation s. We then say that f is *bounded by* b in situation s (of a BAT $\mathcal{D}$), if $\mathcal{D} \models Bounded_{f,b}(s)$.

Definition 7 (Bounded BAT [18]). Let $Bounded_b(s) \doteq \bigwedge_{f \in \mathcal{F}} Bounded_{f,b}(s)$. An action theory $\mathcal{D}$ is *bounded* if there exists $b \in \mathbb{N}$ such that:

$$\mathcal{D} \models \forall s. \textit{Executable}(s) \supset Bounded_b(s).$$ ■

Intuitively, the definition requires that the number of objects contained in the interpretation of the fluents at situation s be less than some $b \in \mathbb{N}$.

We generalize the notion of boundedness to BAT$^-$s. In order to do this, we need to specifically handle observations, which are not constrained in any way in BAT$^-$s. The actual property of interest is whether a BAT$^-$ is bounded "provided its observations are". This assumption can be captured by the FO formula $Bounded_b^{Obs}(s) \doteq \bigwedge_{f \in Obs} Bounded_{f,b}(s)$, which expresses that, at situation s, the interpretations of all observations contain fewer than b distinct objects.

Definition 8 (Bounded BAT$^-$ modulo observations). A BAT$^-$ $\mathcal{D}$ is *bounded modulo observations* if there exist $b, b' \in \mathbb{N}$ such that:

$$\mathcal{D} \models \forall s. (\textit{Executable}(s) \wedge Bounded_{b'}^{Obs}(s)) \supset Bounded_b(s).$$ ■

Observe that Definition 8 generalizes 7: if $\mathcal{D}$ contains no observations, the former trivially reduces to the latter. For simplicity, when no ambiguity arises, we refer to BAT$^-$s which are bounded modulo observations simply as "bounded".

We can now prove state-boundedness for every GA induced by a facility that includes a bounded $\mathcal{D}_F$ and a bounded $\mathcal{D}_R$.

Theorem 3. *If a facility $Fac = \langle \mathcal{D}_R, \mathcal{D}_F, \delta_F^0, Maps \rangle$ is such that $\mathcal{D}_R$ is bounded (modulo observations) and $\mathcal{D}_F$ is bounded, then for any recipe δ_R^0 the induced GA $\mathcal{T}$ is state-bounded.*

Proof. First observe that, by the definition of $\mathcal{T}$'s transition relation, all $\mathcal{D}_R$ and $\mathcal{D}_F$ situations, respectively s_R and s_F, occurring in each state q of $\mathcal{T}$ are executable. Thus, because $\mathcal{D}_F$ is bounded by hypothesis, so are all of its fluents at every such s_F. Consider a fluent $f \in \mathcal{F}_F$. By the labeling function, we have that $f^{\mathcal{I}(q)}(\mathbf{x})$ iff $\mathcal{D}_R^{Maps} \models f(\mathbf{x}, s_F)$, thus, by $\mathcal{D}_F$'s boundedness, it follows that $f^{\mathcal{I}(q)}$ contains a bounded number of distinct tuples; to express this, we simply say that $f^{\mathcal{I}(q)}$ is bounded. Similarly, for fluents $f \in \mathit{Obs}$, since $f^{\mathcal{I}(q)}(\mathbf{x})$ iff $\mathcal{D}_R^{Maps} \models \varphi_f(\mathbf{x}, s_F)$, because $\varphi_f(\mathbf{x}, s_F)$ is domain-independent, $\mathcal{D}_F$ bounded, and s_F executable, $f^{\mathcal{I}(q)}$ is bounded.

For fluents $f \in \mathcal{F}_R \setminus \mathit{Obs}$, by the labeling function, we have that $f^{\mathcal{I}(q)}(\mathbf{x})$ iff $\mathcal{D}_R^{Maps} \models f(\mathbf{x}, s_R)$. Boundedness of $f^{\mathcal{I}(q)}$ follows from boundedness modulo observations of the BAT$^-$ $\mathcal{D}_R$ and executability of s_F and s_R. Indeed, by state-boundedness of $\mathcal{D}_F$ and executability of s_F, all observation fluents in $\mathcal{D}_R^{Maps}$ are bounded at situation s_F (regardless of s_R). But then, for such s_F, by boundedness modulo observations of $\mathcal{D}_R$ and executability of s_R, it follows that all non-observation fluents of $\mathcal{D}_R^{Maps}$ are bounded at situations s_F and s_R.

Finally, it is immediate to see that all other fluents in the labeling are bounded, as being either propositions or singletons. □

Infinite, state-bounded GAs are the norm when BATs are used to model manufacturing facilities. Typically, fresh parts arrive continuously in a facility for processing. This yields, in general, an infinite number of distinct states (if we consider the parts currently processed as part of the state). However, when resources have bounded capacity, a recipe operates on finitely many parts at a time, and requires only finitely many operations, thus the number of objects processed at any point in time does not exceed the capacity bound. As an example of state-bounded BAT, consider the running example of previous sections. Note that fluents such as $\mathit{part}(part, s)$, $\mathit{at}(part, i, s)$, or $\mathit{material}(part, m, s)$, disappear from the extension as soon as the part in question is stored away, i.e., it exits the cell through an action OUT_CELL. This reflects the fact that, once processed, a part is no longer involved in the process and all the corresponding information can be safely forgotten. In other words, the fluents carrying information about a given part eventually disappear and do not "accumulate". Thus, since only a bounded number of parts are processed at a time and since the information about each of these is bounded, it follows that, at any situation, the corresponding interpretation contains only a bounded number of facts, i.e., a bounded number of objects. This implies a state-bounded induced GA.

Later, we will prove that the induced GA $\mathcal{T}$ is also "generic", a notion which intuitively captures that $\mathcal{T}$'s transitions do not depend on the actual objects contained in

the labeling of the relevant states (apart from finitely many active constants) but only on the mutual relationships among objects. This notion is formally stated as follows.

Given two FO interpretations $\mathcal{I}, \mathcal{I}' \in \mathcal{I}_\Delta^{\mathcal{F},\mathcal{K}}$, an *isomorphism* h between $\mathcal{I}$ and $\mathcal{I}'$ is a function $h : \Delta \mapsto \Delta$ such that *(i)* for every fluent $f \in \mathcal{F}$, $h(\mathbf{y}) \in f^{\mathcal{I}}$ if and only if $\mathbf{y} \in f^{\mathcal{I}'}$ and *(ii)* for every active constant $k \in \mathcal{K}$, $k^{\mathcal{I}} = h(k^{\mathcal{I}'}) = k$. We say that $\mathcal{I}$ and $\mathcal{I}'$ are *isomorphic (under h)*, written $\mathcal{I} \sim_h \mathcal{I}'$, if there exists an isomorphism h between $\mathcal{I}$ and $\mathcal{I}'$. Intuitively, $\mathcal{I}$ and $\mathcal{I}'$ are isomorphic if they are the same interpretation modulo an object renaming that preserves the identity of active constants.

Definition 9 (Genericity). A GA $\mathcal{T} = \langle \Delta_\mathcal{T}, Q, q_0, \rightarrow, \mathcal{I} \rangle$ is said to be *generic* if: for every $q_1, q_1', q_2 \in Q$ and every function $h : \Delta_\mathcal{T} \mapsto \Delta_\mathcal{T}$ such that $\mathcal{I}(q_1) \sim_h \mathcal{I}(q_2)$, if $q_1 \rightarrow q_1'$, then there exists $q_2' \in Q$ such that $q_2 \rightarrow q_2'$ and $\mathcal{I}(q_1') \sim_h \mathcal{I}(q_2')$.

Intuitively, this says that a GA $\mathcal{T}$ is generic if, whenever two states are isomorphic under h, they yield the same transitions modulo the same object renaming prescribed by h. Differently put, a GA is generic if states that are identical modulo object renaming are involved in exactly the same transitions (still modulo object renaming).

We thus have that $\mathcal{T}$ is generic (proven later), and that if $\mathcal{D}_R$ and $\mathcal{D}_F$ are bounded, then $\mathcal{T}$ is state-bounded. This allows the application of the results in [42, 35] which, in turn, allow us to prove the following central result:

Theorem 4. *Given a facility $Fac = \langle \mathcal{D}_R, \mathcal{D}_F, \delta_F^0, Maps \rangle$ such that $\mathcal{D}_R$ and $\mathcal{D}_F$ are bounded, and a recipe δ_R^0 that is realizable by Fac, there exists a controller for δ_F^0 that realizes δ_R^0 and is effectively computable.*

In the next two sections we prove this theorem and show how to actually build and execute a controller.

9. Verifying Realizability for the Bounded Case

In [35] (see Theorems 5 and 6 therein), the problem of checking whether a Transition System that is state-bounded and generic satisfies a formula Φ was proven decidable for Φ belonging to $\mu\mathcal{L}_p$. In the context of this paper, a corresponding result can be formally stated as follows (recall that $\mu\mathcal{L}_c$ is a sub-language of $\mu\mathcal{L}_p$):

Theorem 5. *Given a generic, state-bounded GA $\mathcal{T}$, there exists a finite-state GA $\bar{\mathcal{T}}$ such that, for every $\mu\mathcal{L}_c$ formula Φ, $\mathcal{T} \models \Phi$ iff $\bar{\mathcal{T}} \models \Phi$.*

When it exists, we call such a $\bar{\mathcal{T}}$ a *faithful abstraction of* $\mathcal{T}$. The theorem says that if state-boundedness and genericity hold for $\mathcal{T}$ then one can check whether it satisfies Φ by simply checking whether (one of) its faithful abstraction(s) $\bar{\mathcal{T}}$ satisfies it.

In this section we prove Theorem 4. To this end, we first prove in Section 9.1 that the induced GA $\mathcal{T}$ is generic. This, together with Theorem 3, enables the application of Theorem 5 to $\mathcal{T}$ (under the assumption that $\mathcal{D}_F$ is bounded), which, in turn, implies the existence of a faithful abstraction $\bar{\mathcal{T}}$. Then, in Section 9.2 we prove that:

Theorem 6. *There exists a strategy ς on $\mathcal{T}$ for player* CONTROLLER *if and only if a winning strategy $\bar{\varsigma}$ exists on a faithful abstraction $\bar{\mathcal{T}}$ of $\mathcal{T}$.*

Finally, in Section 10 we show how a controller for $\mathcal{T}$ can be extracted from $\bar{\varsigma}$.

Together, these results guarantee that *(i)* there exists a finite GA $\bar{\mathcal{T}}$ that is a faithful abstraction of $\mathcal{T}$ and that *(ii)* a controller for $\mathcal{T}$ can be effectively extracted from $\bar{\mathcal{T}}$. From these, Theorem 4 follows (as the results hold also for memoryless strategies).

9.1. Faithful abstraction

We start by proving the following result.

Lemma 1. *The GA $\mathcal{T}$ induced by a facility $Fac = \langle \mathcal{D}_R, \mathcal{D}_F, \delta_F^0, Maps \rangle$ and a recipe δ_R^0 is generic.*

Proof. Consider three states $q_1, q_1', q_2 \in Q$ of $\mathcal{T}$ such that $q_1 \rightarrow q_1'$ and $\mathcal{I}(q_1) \sim_h \mathcal{I}(q_2)$, for some isomorphism $h : \Delta_{\mathcal{T}} \mapsto \Delta_{\mathcal{T}}$. Let $q_i = \langle \vartheta^i, \delta_R^i, s_R^i, \delta_F^i, s_F^i, \delta^i \rangle$ (possibly primed). Since $\mathcal{I}(q_1) \sim_h \mathcal{I}(q_2)$, the interpretations of *pcR* at q_1 and q_2 match, i.e., $pcR^{\mathcal{I}(q_1)} = pcR^{\mathcal{I}(q_2)}$ (recall that program counters act as active constants, thus h is the identity on them) and the interpretations of *envR* at the same states are the same modulo the object renaming defined by h (modulo h-renaming, for short). Thus, by considering the relationship established by the labeling function between state components and state labeling, we have that δ_R^1 and δ_R^2 have the same program counter but environments that are isomorphic under h. By the same argument, the above also holds for *pcF*, *envF*, *pcA*, and *envA*. Thus, δ_F^1 and δ_F^2 are the same program modulo h-renaming of their respective environments, and δ^1 and δ^2 are the same program modulo h-renaming of their respective environments. For the same reason, we have that: $Final(\delta_R^1, s_R^1)$ iff $Final(\delta_R^2, s_R^2)$, $Final(\delta_F^1, s_F^1)$ iff $Final(\delta_F^2, s_F^2)$, and $Final(\delta^1, s_F^1)$ iff $Final(\delta^2, s_F^2)$. In a similar way, we can prove that $\vartheta^1 = \vartheta^2$, i.e., the same player is to move at q_1 and q_2.

Now, observe that, by the definition of $\mathcal{T}$'s labeling function, the interpretation of the fluents associated with a game state q_i is essentially that given by $\mathcal{D}_R^{Maps}$ at its (combined) situations s_F^i and s_R^i. Therefore, an action $\mathbf{A}(\mathbf{x}, s_F)$ is legal for $\langle \delta_R^1, s_R^1 \rangle$ iff action $\mathbf{A}(h(\mathbf{x}), s_F)$ is legal for $\langle \delta_R^2, s_R^2 \rangle$, and, similarly, an action $\mathbf{A}(\mathbf{x})$ is legal for $\langle \delta_F^1, s_F^1 \rangle$ and $\langle \delta^1, s_F^1 \rangle$ iff $\mathbf{A}(h(\mathbf{x}))$ is such for $\langle \delta_F^2, s_F^2 \rangle$ and $\langle \delta^2, s_F^2 \rangle$. Consequently, by the definition of $\mathcal{T}$'s transition relation, since $q_1 \rightarrow q_1'$, there must exist a state q_2' such that $q_2 \rightarrow q_2'$. The fact that $\mathcal{I}(q_1') \sim_h \mathcal{I}(q_2')$ is a consequence of the fact that: *(i)* $\mathcal{I}(q_1) \sim_h \mathcal{I}(q_2)$, i.e., q_1 and q_2 have the same labeling modulo h-renaming; *(ii)* the moves $q_1 \rightarrow q_1'$ and $q_2 \rightarrow q_2'$ are defined by the same action, modulo h-renaming, respectively $\mathbf{A}(\mathbf{x}, s_F)$ and $\mathbf{A}(h(\mathbf{x}), s_F)$, if $\vartheta^1 = \vartheta^2 = \textsc{Env}$, and $\mathbf{A}(\mathbf{x})$ and $\mathbf{A}(h(\mathbf{x}))$, if $\vartheta^1 = \vartheta^2 = \textsc{Ctrl}$; *(iii)* Successor-state axioms are FO formulas, which are invariant to isomorphisms, modulo the object renaming defined by the isomorphism itself. Thus move $q_2 \rightarrow q_2'$ is the same as $q_1 \rightarrow q_1'$, modulo h-renaming, by which it follows that $\mathcal{I}(q_1') \sim_h \mathcal{I}(q_2')$. □

We now address the issue of computing a finite, faithful abstraction of $\mathcal{T}$.

Lemma 2. *If a GA $\mathcal{T}$ is generic and state-bounded, then a finite, faithful abstraction $\bar{\mathcal{T}}$ of $\mathcal{T}$ is effectively computable.*

Proof. Existence of $\bar{\mathcal{T}}$ is an immediate application to $\mathcal{T}$ of Theorem 17 in [35]. Effective computability and finiteness follow from the proof of the same theorem, once we

observe that: *(i)* since $\mathcal{T}$ is state-bounded, the successor state q' of a generic state q is computable, hence we do not need to construct $\mathcal{T}$ explicitly to build its finite abstraction; *(ii)* state-boundedness of $\mathcal{T}$ implies that checking whether two interpretations are isomorphic is decidable; and *(iii)* there exist only finitely many equivalence classes of isomorphic interpretations, thus only finitely many equivalent states. Based on these observations the definitions used by the proof to build the abstraction are operational, thus enabling effective construction of $\bar{\mathcal{T}}$. □

Intuitively, $\bar{\mathcal{T}}$ is a GA obtained from $\mathcal{T}$ by collapsing the classes of states having isomorphic interpretations into one representative state, and by adding a transition from one class q to another class q' iff there exists one transition between the chosen representatives in $\mathcal{T}$ (this, by generality, implies that all states in q have a transition to some state in q'). In this paper we do not describe how to obtain this abstraction, but refer the reader to the procedure illustrated in [35]. The procedure is applicable to generic transition systems with first-order state representations, of which induced GAs are instances. In our setting, the procedure requires that the transition function of $\mathcal{T}$ is computable and that the existence of an isomorphism between states is decidable, which is indeed the case, as discussed in the proof of Lemma 2. Theorem 5 comes as a direct consequence of Lemma 2.

The resulting GA is $\bar{\mathcal{T}} = \langle \bar{\Delta}_{\mathcal{T}}, \bar{Q}, q_0, \bar{\to}, \bar{\mathcal{I}} \rangle$, where $\bar{\Delta}$, $\bar{Q}$, and $\bar{\to}$ are finite subsets of their counterparts in $\mathcal{T}$, and $\bar{\mathcal{I}}$ is the projection of $\mathcal{I}$ over $\bar{Q}$, with $\Delta_{\mathcal{T}}$ replaced by $\bar{\Delta}_{\mathcal{T}}$. Notice that $\mathcal{T}$ and $\bar{\mathcal{T}}$ share the same set $\mathcal{F}$ of fluents.

We conclude this section by characterizing the relationship between $\mathcal{T}$ and $\bar{\mathcal{T}}$, which will be needed when computing executable strategies for $\mathcal{T}$ from those for $\bar{\mathcal{T}}$. This result relies on a variant of the well-known notion of bisimulation, called *persistence-preserving* bisimulation, or *p-bisimulation* for short, defined in [35].

We adapt p-bisimulation to GAs. The notion is defined co-inductively over triples $\langle q_1, h, q_2 \rangle$, where q_1 and q_2 are states of two GAs and $h \in H$ is an isomorphism between their interpretations which, differently from the case of bisimulation, is restricted to the *active domains* [18].

Definition 10 (p-bisimulation). A relation $\beta \subseteq Q_1 \times H \times Q_2$ is a *p-bisimulation* between two GAs $\mathcal{T}_1$ and $\mathcal{T}_2$ if $\langle q_1, h, q_2 \rangle \in \beta$ implies that:

(i) q_1 and q_2 have isomorphic fluent extensions, according to $h : adom(\mathcal{I}_1(q_1)) \mapsto adom(\mathcal{I}_2(q_2))$ (objects not occurring in fluent extensions are neglected). We denote this by writing $\tilde{\mathcal{I}}_1(q_1) \sim_h \tilde{\mathcal{I}}_2(q_2)$;

(ii) for every successor q_1' of q_1 there exists a successor q_2' of q_2 and a bijection $b : adom(\mathcal{I}_1(q_1)) \cup adom(\mathcal{I}_1(q_1')) \mapsto adom(\mathcal{I}_2(q_2)) \cup adom(\mathcal{I}_2(q_2'))$ that extends h to $adom(\mathcal{I}_1(q_1'))$ such that, for the restriction h' of b to $adom(\mathcal{I}_2(q_2'))$, $\langle q_1', h', q_2' \rangle$ is in β;

(iii) the analogue of (ii) holds for every successor q_2' of q_2. ■

This property intuitively captures the fact that two states of two GAs $\mathcal{T}_1$ and $\mathcal{T}_2$ are persistence-preserving bisimilar if there is an isomorphism between them that can be

extended in successor states, *while preserving bisimulation*. In other words, the identity of objects is preserved as long as they persist in the active domain or if they have just disappeared from it. Two GAs are p-bisimilar if their respective initial states are in some p-bisimulation. Since from [35] we have that every transition system is p-bisimilar to its faithful abstractions, then we directly have that:

Lemma 3. *$\mathcal{T}$ is p-bisimilar to $\bar{\mathcal{T}}$.*

The notion of p-bisimilarity will be essential to relate (winning) strategies for $\bar{\mathcal{T}}$ to those of the infinite-state $\mathcal{T}$. This will allow us to prove Theorem 6, in the next section.

9.2. Strategy existence

In this section we address Theorem 6, namely showing that there exists a strategy ς on $\mathcal{T}$ for player CONTROLLER iff a winning strategy $\bar{\varsigma}$ exists on $\bar{\mathcal{T}}$, relying on the notion of p-bisimilarity introduced above. First, we need a way to relate the strategies of two different GAs, and in particular two that are p-bisimilar. This is done through the following definition, which we will then apply to $\mathcal{T}$ and $\bar{\mathcal{T}}$.

Definition 11 (*p*-bisimilar strategy transformation). Consider two GAs $\mathcal{T}$ and $\mathcal{T}'$ that are p-bisimilar, and let ς be a CONTROLLER strategy for $\mathcal{T}$. A strategy ς' for $\mathcal{T}'$ is said to be a *p-bisimilar transformation of ς to $\mathcal{T}$*, if there exists a p-bisimulation β such that for every history $\tau = q_0 \cdots q_\ell$ of $\mathcal{T}$ induced by ς, there exists a history $\tau' = q'_0 \cdots q'_\ell$ of $\mathcal{T}'$ induced by ς' and a sequence of bijections $h_i : adom(\mathcal{I}(q_i)) \rightarrow adom(\mathcal{I}(q'_i))$, with $i \in [0, \ell]$, such that for every i:

- $\langle q_i, h_i, q'_i \rangle \in \beta$ and
- if $\mathcal{I}(q_i) \sim_{h_i} \mathcal{I}'(q'_i)$ and $\mathcal{I}(q_{i+1}) \sim_{h_{i+1}} \mathcal{I}'(q'_{i+1})$ then there exists a bijection $b : adom(\mathcal{I}(q_i)) \cup adom(\mathcal{I}(q_{i+1})) \rightarrow adom(\mathcal{I}(q'_i)) \cup adom(\mathcal{I}(q'_{i+1}))$ so that $h_i = b\,|_{adom(\mathcal{I}(q_i))}$ and $h_{i+1} = b\,|_{adom(\mathcal{I}(q_{i+1}))}$.[1] We call the isomorphism h_{i+1} the *update of h_i with respect to q_{i+1} and q'_{i+1}*.

For p-bisimilar GAs, the following result holds.

Theorem 7. *If two GAs $\mathcal{T}$ and $\mathcal{T}'$ are p-bisimilar then there exists a* CONTROLLER *strategy ς on $\mathcal{T}$ iff there exists a* CONTROLLER *strategy ς' on $\mathcal{T}'$ that is a p-bisimilar transformation of ς.*

Proof. By p-bisimilarity, there exists a bisimulation β such that for every history $\tau = q_0 \cdots q_\ell$ of $\mathcal{T}$ induced by ς, there exists a history $\tau' = q'_0 \cdots q'_\ell$ of $\mathcal{T}'$ that fulfills the requirement of τ' in Definition 11. For the if-part, we define ς' as $\varsigma'(q'_0 \cdots q'_{\ell-1}) = q'_\ell$, for every history $q_0 \cdots q_{\ell-1} q_\ell$ of $\mathcal{T}$ induced by ς, such that $q_{\ell-1} \models$ *turnCtrl*. The only-if part is analogous. □

In particular, the result holds for memoryless, winning strategies as well (as required by Theorem 2). Hence, by Lemma 3, Theorem 6 follows by applying the theorem above with $\mathcal{T}'$ replaced by $\bar{\mathcal{T}}$.

[1] The symbol | denotes projection.

10. Computing and Executing the Controller for the Bounded Case

Theorem 6 and Theorem 2 provide with a constructive way of transforming a memoryless winning strategy $\bar{\varsigma}_m$ for Φ_{Real} (cf. Section 7.1) and $\bar{\mathcal{T}}$ into an actual controller for δ_R^0 that realizes δ_F^0.

Specifically, we follow the construction of Definition 11, thanks to Theorem 7. To do so, we need to relate the states of a history $\bar{\tau}$ of $\bar{\mathcal{T}}$ to those of a history τ of $\mathcal{T}$, by applying the isomorphisms that preserve the identity of the objects that persist and of those that have just disappeared from the active domain. While the existence of a p-bisimulation β between $\mathcal{T}$ and $\bar{\mathcal{T}}$ is guaranteed by Lemma 3, we cannot represent it explicitly ($\mathcal{T}$ is infinite): the sequence of isomorphisms must be computed on the fly.

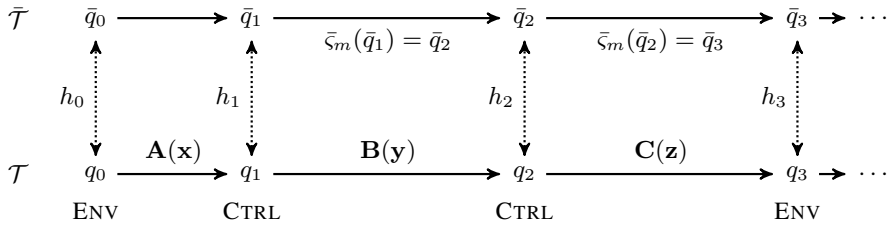

Figure 5: Executing a winning strategy $\bar{\varsigma}_m$ for the faithful abstraction $\bar{\mathcal{T}}$ on the original GA $\mathcal{T}$. Crucially, this approach does not require one to explicitly compute $\mathcal{T}$.

The procedure works as depicted in Figure 5. Initially, both $\mathcal{T}$ and $\bar{\mathcal{T}}$ are in their initial state ($q_0 = \bar{q}_0$) and h_0 is the identity function. Then, since $q_0 \models$ *turnEnv* and it is the turn of ENVIRONMENT, for any state q_1 in $\mathcal{T}$ such that $q_0 \to q_1$ (for some action $\mathbf{A}(\mathbf{x})$ in the recipe), we select an isomorphic state $\bar{q}_1$ in $\bar{\mathcal{T}}$ such that $q_1 \sim_{h_1} \bar{q}_1$, where h_1 is the update of h_0 with respect to q_1 and $\bar{q}_1$ as in Definition 11. After this (it is now the turn of CONTROLLER), assume $\bar{q}_2 = \bar{\varsigma}_m(\bar{q}_1)$ is the state selected by the strategy on the abstraction $\bar{\mathcal{T}}$. We then select a state q_2 such that $q_2 \sim_{h_2} \bar{q}_2$, which gives us the move $q_2 = \varsigma_m(q_1)$ on $\mathcal{T}$. In the inductive step, assume that $\bar{\tau} = \bar{q}_0 \dots \bar{q}_{\ell-1}$, with $\tau = q_0 \dots q_{\ell-1}$, is the bisimilar history on $\mathcal{T}$ computed so far, with $last(\tau) \models$ *turnCtrl*. Let $\bar{q}_\ell = \bar{\varsigma}_m(\bar{q}_{\ell-1})$. We proceed in the same way and obtain the move $q_\ell = \varsigma_m(q_{\ell-1})$ on $\mathcal{T}$, with $q_\ell \sim_{h_\ell} \bar{q}_\ell$, and so on. In Figure 5, $\mathbf{B}(\mathbf{y})$ and $\mathbf{C}(\mathbf{z})$ are the two compound actions which constitute a possible, complete execution of the program $\delta_{\mathbf{A}}(\mathbf{x})$ to which the action $\mathbf{A}(\mathbf{x})$ is mapped. In q_3 the turn is given back to ENVIRONMENT since $q_3 \models \phi_{\text{OK}}$.

Following the same reasoning, we can define this as a procedure which constructively *computes and executes* on-the-fly a controller returning the sequence of moves in $\mathcal{T}$ which correspond to the implementation of the last move of ENVIRONMENT, given a winning CONTROLLER strategy $\bar{\varsigma}$ for $\bar{\mathcal{T}}$. The procedure is given as Algorithm 1, which is initially called for $\bar{q} = q = q_0$ and with h initialized as the identity function. Note that the procedure computes the next compound action to execute on the fly, by restricting only to the execution of the GA $\mathcal{T}$ that is being determined at runtime.

For simplicity, differently from the formal definition of controller (but consistent with the notion of memoryless strategies), the procedure returns, at each step, a single compound action to be executed in the facility rather than returning a complete

Algorithm 1 *execute_controller*$(\bar{\mathcal{T}}, \bar{\varsigma}_m, \bar{q}, q, h)$

1: let $q = \langle \cdot, \delta_R, s_R, \delta_F, s_F, \cdot \rangle$ – we use $\cdot$ for *don't care* elements
2: **if** $q \models \phi_{\text{OK}}$ **then**
3: let $q' = \langle \mathit{turnCtrl}, \delta'_R, s'_R, \delta'_F, s'_F, \cdot \rangle$ be the state of $\mathcal{T}$ selected by the ENVIRONMENT, with $q \to q'$. This corresponds to some recipe action $\mathbf{A}(\mathbf{x}, S^0_F)$ such that $s'_R = do(\mathbf{A}(\mathbf{x}, S^0_F), s_R)$
4: let $\bar{q}'$ be a state of $\bar{\mathcal{T}}$, with $\bar{q} \to \bar{q}'$, that is isomorphic to q', i.e., such that $q' \sim_{h'} \bar{q}'$, for h' the update of h w.r.t. q' and $\bar{q}'$.
5: **else**
6: let $\bar{q}' = \bar{\varsigma}_m(\bar{q})$ be the state of $\bar{\mathcal{T}}$ selected by the winning strategy;
7: let $q' = \langle \cdot, \delta'_R, s'_R, \delta'_F, s'_F, \cdot \rangle$ in $\mathcal{T}$ be such that $q \to q'$ and $q' \sim_{h'} \bar{q}'$, for h' the update of h w.r.t. q' and $\bar{q}'$;
8: execute the action $\mathbf{B}(\mathbf{y})$ on $\mathcal{T}$, so that $s'_F = do(\mathbf{B}(\mathbf{y}), s_F)$;
9: **end if**
10: **if** $q' \not\models \phi_{\text{OK}} \wedge \mathit{finalEnv}$ **then**
11: *execute_controller*$(\bar{\mathcal{T}}, \bar{\varsigma}_m, \bar{q}', q', h')$
12: **end if**

sequence of actions. Indeed, according to Definition 2, a controller for δ^0_F that realizes δ^0_R is a function $\rho : \mathrm{C}_{\delta^0_R} \times \mathrm{C}_{\delta^0_F} \times \mathrm{C}_{\delta^0_R} \mapsto \mathrm{C}^*_{\delta^0_F}$ which, given the current configurations of the recipe and facility together with a new selected configuration for the recipe, returns a sequence of new configurations for the facility. This sequence, in turn, identifies the sequence of compound actions to execute. It is however possible to reconstruct the function ρ from the execution of Algorithm 1: given q at line 1, which specifies the current configuration $\langle \delta_R, s_R \rangle$ of the recipe and the current configuration $\langle \delta_F, s_F \rangle$ the facility process, given then a new configuration $\langle \delta'_R, s'_R \rangle$ for the recipe as in line 3, then $\rho(\langle \delta_R, s_R \rangle, \langle \delta_F, s_F \rangle, \langle \delta'_R, s'_R \rangle)$ is defined as the sequence of configurations $\langle \delta'_F, s'_F \rangle$ of the facility process as in line 7, computed by the iterative execution of the procedure, until a new state q such that $q \models \phi_{\text{OK}}$ is reached (i.e., then the turn is given back to the ENVIRONMENT player in the GA $\mathcal{T}$).

Example 5 (A controller for the running example). The controller for the facility process δ^0_F described in Section 5 that realizes the recipe δ^0_R in Example 4, corresponding to a possible winning strategy for player CONTROLLER, is depicted graphically in Figure 6. For brevity, let us refer to $\langle \mathcal{D}_i, \delta_i \rangle$, with $i \in \{1, \ldots, n\}$, by $\mathcal{R}_i$. We now show how to execute this controller: at the beginning, the recipe can only execute action LOAD(`f`, `4`, `steel`, `810`, `756`, `29`). In response, the controller prescribes the execution of two compound actions. With the first, resource $\mathcal{R}_3$ equips an end effector in order to prepare the loading of the part `b` into the cell (recall that `b` denotes a partID), while all other resources remain idle. The compound action is $\{$NOP, NOP, EQUIP(`gripper`, `3`), NOP, NOP$\}$. With the second, $\mathcal{R}_3$ loads the part while other resources remain idle: the compound action is $\{$NOP, NOP, IN_CELL(`f`, `4`, `steel`, `810`, `756`, `29`, `3`), NOP, NOP$\}$. At this point, two alternatives are possible for the recipe: the next instruction is LOAD(`b`, `2`, `steel`, `312`, `23`, `20`) $\|$ DRILL(`f`, `.3`, `200`, `123`, `89`, `21`), that is, the recipe can either first load part `b` or first drill a

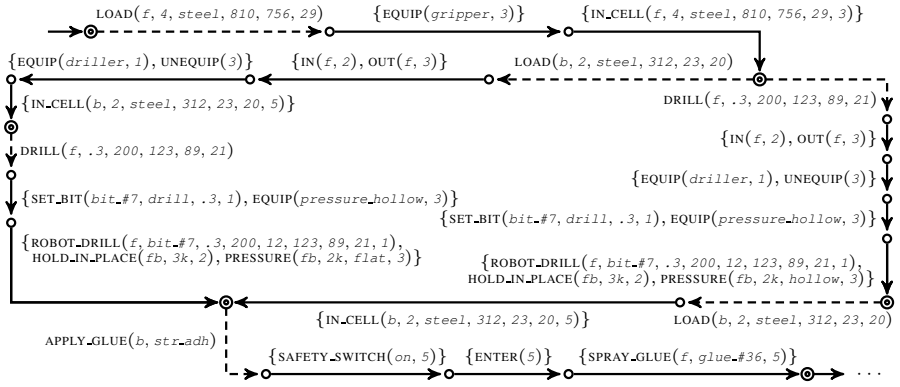

Figure 6: A fragment of a possible controller for δ_F^0 that realizes δ_R^0 as in the running example, represented as a control structure in which double-circled control states correspond to states q of $\mathcal{T}$ such that $q \models \phi_{\text{OK}}$, i.e., it is the turn of player ENVIRONMENT. Dashed transitions from these control states are labelled with the next action of the recipe, and the rest with compound actions for the facility process (we do not include NOP actions for brevity). Each control state is associated to configurations of δ_F^0 and δ_R^0, indicating their current program and situation, but these are not shown for brevity.

hole in `f`. As shown in Figure 6, in the former case this controller first makes it so that `f` is moved from $\mathcal{R}_3$ to $\mathcal{R}_2$, then $\mathcal{R}_3$ detaches the gripper while $\mathcal{R}_1$ equips a drilling end effector to prepare for the next recipe request, and finally $\mathcal{R}_5$ (the human operator) is instructed to fetch part `b` (without entering the cell). In the latter case, after the same preparatory steps, $\mathcal{R}_1$ sets the correct drill bit in the driller while $\mathcal{R}_3$ equips a hollow pressure applicator; then $\mathcal{R}_1$ is instructed to perform the drilling of the part `f` which is now currently held on $\mathcal{R}_2$ (the fixture), while $\mathcal{R}_3$ is applying pressure to balance the drilling force. Note that there are some arguments of these actions that were determined by the procedure given as Algorithm 1: for instance, the drill bit (`bit_#7`), the feed rate (`12`), the glue (`glue_#36`). These arguments corresponded to pick variables (i.e., non-deterministic choice of arguments) in the programs determined by the mappings $Maps$. Thanks to the genericity of the GA, any alternative value which is equivalent modulo isomorphism could have been selected.

The interaction between the recipe, the controller and the facility program continues in the same way until the recipe is completed. A *possible* resulting execution of this controller, for *a possible* evolution of recipe δ_R^0, is the following (NOP actions are omitted):

1 : {EQUIP(*gripper*, *3*)}
2 : {IN_CELL(*f*, *4*, *steel*, *810*, *756*, *29*, *3*)}
3 : {IN(*f*, *2*), OUT(*f*, *3*)}
4 : {EQUIP(*driller*, *1*), UNEQUIP(*3*)}
5 : {IN_CELL(*b*, *2*, *steel*, *312*, *23*, *20*, *5*)}
6 : {SET_BIT(*bit_#7*, *drill*, *.3*, *1*), EQUIP(*pressure_hollow*, *3*)}
7 : {ROBOT_DRILL(*f*, *bit_#7*, *.3*, *200*, *12*, *123*, *89*, *21*, *1*), HOLD_IN_PLACE(*f*, *3k*, *2*), PRESSURE(*f*, *2k*, *hollow*, *3*)}
8 : {SAFETY_SWITCH(*on*, *5*)}
9 : {ENTER(*5*)}
10 : {SPRAY_GLUE(*f*, *glue_#36*, *5*)}
11 : {POSITION(*f*, *b*, *fb*, *7*, *201*, *29*, *5*)}
12 : {EXIT(*5*)}
13 : {SAFETY_SWITCH(*off*, *5*)}
14 : {SET_BIT(*bit_#22*, *cntr_reaming*, *.3*, *1*)}
15 : {ROBOT_DRILL(*fb*, *bit_#22*, *.3*, *123*, *89*, *21*, *1*), HOLD_IN_PLACE(*fb*, *3k*, *2*), PRESSURE(*fb*, *2k*, *hollow*, *3*)}
16 : {UNEQUIP(*1*), UNEQUIP(*3*)}
17 : {EQUIP(*rivet_gun*, *1*), EQUIP(*pressure_flat*, *3*)}
18 : {RIVET(*fb*, *alu_rvt_.3*, *123*, *89*, *21*, *1*), HOLD_IN_PLACE(*fb*, *2k*, *2*), PRESSURE(*fb*, *1k*, *flat*, *3*)}
19 : {START_COMPRESSOR(*1*), UNEQUIP(*3*)}
20 : {UNEQUIP(*1*), OUT(*fb*, *2*), IN(*fb*, *3*)}
21 : {EQUIP(*gripper*, *3*)}
22 : {OUT_CELL(*fb*, *ok*, *3*)}

This sequence exemplifies a specific execution of the facility in which the second LOAD operation is executed before DRILL and the test on the observation represented by the fluent *precision* is false, i.e., $\mathcal{R}_3$ does not have high drilling precision. Counter-reaming is thus necessary: the situation-independent fluent *prec_rating*(*high*, *3*) is not in $\mathcal{D}_F$.

Specifically, the compound actions at lines 1-2 implement the action LOAD(*f*, *4*, *steel*, *810*, *756*, *29*) in the recipe; the compound actions in lines 3-5 implement the action LOAD(*b*, *2*, *steel*, *312*, *23*, *20*); the compound actions in lines 6-7 implement DRILL(*f*, *.3*, *200*, *123*, *89*, *21*); the compound actions in lines 8-10 implement APPLY_GLUE(*b*, *str_adh*); the compound action in line 11 implements PLACE(*b*, *f*, *fb*, *7*, *201*, *29*); those in lines 12-15 implement REAMING(*fb*, *.3*, *123*, *89*, *21*); those in lines 16-18 implement RIVET(*fb*, *123*, *89*, *21*) and finally the remaining lines implement STORE(*fb*, *ok*).

Notice, however, that each of these segments does not exactly correspond to the mapped program for each action in the recipe: additional low-level operations are added to take care of movements of parts and other preparatory steps. These are automatically computed by our procedure. □

11. Conclusions

The ability of manufacturing providers to *automatically* assess the manufacturability of products and synthesize process plan controllers is essential to realize any real-world MaaS application. Research in Artificial Intelligence and Computer Science can be exploited to provide *mathematical foundations* for the manufacturing concepts

and to solve the core challenges of MaaS, as shown by recent efforts in basing MaaS on fundamental ideas from CS and advancements in AI.

On the one hand, this has allowed previous approaches to formalize the requirements and techniques for the automated synthesis of process plan controllers [9, 11, 12, 13, 43, 44] and offer a formal foundation to practical manufacturing approaches [14, 15, 45]. On the other hand, however, these previous approaches are based on a *propositional description* of the states of the devices, workpieces, and processes, and such representations are too idealized for implementing fully-fledged applications. While in some simpler scenarios a propositional approach may be sufficient, the resulting discretization is unwieldy and unnatural and, more importantly, cannot deal with potentially unbounded objects. Concrete and realistic approaches for MaaS necessarily require a rich, relational description of states, as well as advanced computational techniques that are able to manipulate this relational representation: real manufacturing processes depend on the objects and data they produce and consume, which are in general unbounded.

In this paper, we have addressed these shortcomings and proposed a formal logical framework that explicitly accounts for this dependency. Our approach offers a "data-aware" process formalization where data and objects are treated as first-class citizens, addressing relational representations of the states by relying on the research on reasoning about actions in AI. Critically, we did not rely on ad-hoc representations. Our framework uses Situation Calculus action theories for capturing actions in manufacturing processes, and high-level ConGolog programs over such action theories for capturing the processes defined over these actions. This makes a whole body of related work readily available to address a number of problems arising in the context of manufacturing systems [46, 47, 48, 49, 50, 51]. In addition, we can leverage the first-order state representations of action formalisms and the second-order/fixpoint characterization of state-change provided by programs, giving a formal and declarative representation of the MaaS manufacturing setting. We have also shown that these techniques are actually *effective*, in that they correspond to actual algorithms that allow the extraction of actual controllers when the objects and data in the resulting Situation Calculus action theories are never "accumulated" (i.e., *state-bounded* theories [18]). This is the first decidability result for controller synthesis in a relational/first-order state reasoning about actions settings such as the Situation Calculus.

We have only scratched the surface of what KR formalisms like the Situation Calculus can bring to this new manufacturing paradigm. Our results and constructions can be applied in other frameworks for reasoning about actions in AI as well as data-aware/artifact-centric processes frameworks in databases [19, 20, 21]. Furthermore, it would be interesting to equip resources with autonomous deliberation capabilities [52], e.g., to react to exogenous events during execution, or to monitor streaming production data [53], to include an explicit treatment of time and other continuous value quantities [54], or to consider non-Markovian action theories [55] for manufacturing recipes. We plan to address these directions as future work. With the theory and framework in place, a next step is to devise actual tools for the synthesis of manufacturing processes, that are based on a ConGolog formalization. In this respect, possible approaches to start from are, e.g., those of [56], based on predicate abstraction, or [57, 58], where verification is addressed by resorting to First-Order BDDs, and [59], where (LTL) realizability

is addressed by compilation into safety and reachability games.

Acknowledgements

This work was supported by the Unibz CRC project "Data-aware controllers for Manufacturing" (DACoMan), by the Unibz ID project "Automated Process Planning in Cyber Physical Production Systems of Smart Factories" (SMART-APP), by the Unibz RTD project SYNCED, by the EPSRC grants "Evolvable Assembly Systems" (EP/K018205/1) and "Cloud Manufacturing" (EP/K014161/1), by the Sapienza project "Data-awaRe Automatic Process Execution" (DRAPE), by the ERC Advanced Grant WhiteMech (No. 834228) and by the EU ICT-48 2020 project TAILOR (No. 952215).

References

[1] UK Technology Strategy Board, A landscape for the future of high value manufacturing in the UK, Tech. rep. (2012).

[2] C. Rhodes, Manufacturing: Statistics and Policy. Briefing Paper, House of Commons Library, 2015.

[3] X. Xu, From cloud computing to cloud manufacturing, Robotics and Computer-Integrated Manufacturing 28 (1) (2012) 75 – 86.

[4] Y. Lu, X. Xu, J. Xu, Development of a hybrid manufacturing cloud, Journal of Manufacturing Systems 33 (4) (2014) 551–566.

[5] R. Henzel, G. Herzwurm, Cloud manufacturing: A state-of-the-art survey of current issues, Procedia CIRP 72 (2018) 947 – 952, 51st CIRP Conference on Manufacturing Systems.

[6] O. Fisher, N. Watson, L. Porcu, D. Bacon, M. Rigley, R. L. Gomes, Cloud manufacturing as a sustainable process manufacturing route, Journal of Manufacturing Systems 47 (2018) 53 – 68.

[7] M. P. Groover, Automation, production systems, and computer-integrated manufacturing, Prentice Hall Press, 2007.

[8] G. De Giacomo, F. Patrizi, S. Sardiña, Automatic behavior composition synthesis, Artificial Intelligence 196 (2013) 106–142.

[9] L. de Silva, P. Felli, J. C. Chaplin, B. Logan, D. Sanderson, S. Ratchev, Realisability of production recipes, in: Proceedings of the 22nd European Conference on Artificial Intelligence (ECAI-2016), ECCAI, IOS Press, The Hague, The Netherlands, 2016, pp. 1449–1457.

[10] P. Felli, B. Logan, S. Sardina, Parallel behavior composition for manufacturing, in: S. Kambhampati (Ed.), Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI 2016), 2016, pp. 271–278.

[11] P. Felli, L. de Silva, B. Logan, S. M. Ratchev, Process plan controllers for nondeterministic manufacturing systems, in: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017, pp. 1023–1030.

[12] G. De Giacomo, M. Vardi, P. Felli, N. Alechina, B. Logan, Synthesis of orchestrations of transducers for manufacturing, in: Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI-18), AAAI Press, New Orleans, USA, 2018, pp. 6161–6168.

[13] G. De Giacomo, N. Alechina, T. Brazdil, P. Felli, B. Logan, M. Vardi, Unbounded orchestrations of transducers for manufacturing, in: Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence (AAAI-19), AAAI Press, Honolulu, USA, 2019.

[14] O. J. Bakker, J. C. Chaplin, L. de Silva, P. Felli, D. Sanderson, B. Logan, S. Ratchev, Toward process control from formal models of transformable manufacturing systems, Procedia CIRP 63 (2017) 521 – 526, manufacturing Systems 4.0 – Proceedings of the 50th CIRP Conference on Manufacturing Systems.

[15] L. de Silva, P. Felli, D. Sanderson, J. C. Chaplin, B. Logan, S. Ratchev, Synthesising process controllers from formal models of transformable assembly systems, Robotics and Computer-Integrated Manufacturing 58 (2019) 130 – 144.

[16] M. Grüninger, C. Menzel, The process specification language (PSL) theory and applications, AI Magazine 24 (2003) 63–74.

[17] H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, R. B. Scherl, GOLOG: A logic programming language for dynamic domains, Journal of Logic Programming 31 (1997) 59–84.

[18] G. De Giacomo, Y. Lespérance, F. Patrizi, Bounded situation calculus action theories, Artif. Intell. 237 (2016) 172–203.

[19] B. B. Hariri, D. Calvanese, G. De Giacomo, A. Deutsch, M. Montali, Verification of relational data-centric dynamic systems with external services, in: Proc. of PODS, 2013, pp. 163–174.

[20] F. Belardinelli, A. Lomuscio, F. Patrizi, Verification of agent-based artifact systems, J. Artif. Intell. Res. 51 (2014) 333–376.

[21] A. Deutsch, R. Hull, Y. Li, V. Vianu, Automatic verification of database-centric systems, SIGLOG News 5 (2) (2018) 37–56.

[22] J. McCarthy, P. J. Hayes, Some philosophical problems from the standpoint of artificial intelligence, Machine Intelligence 4 (1969) 463–502.

[23] R. Reiter, Knowledge in Action. Logical Foundations for Specifying and Implementing Dynamical Systems, The MIT Press, 2001.

[24] H. J. Levesque, G. Lakemeyer, The Logic of Knowledge Bases, The MIT Press, 2001.

[25] S. Sardina, G. De Giacomo, Y. Lespérance, H. J. Levesque, On the semantics of deliberation in IndiGolog – From theory to implementation, Annals of Mathematics and Artificial Intelligence 41 (2–4) (2004) 259–299.

[26] S. Abiteboul, R. Hull, V. Vianu, Foundations of databases, Addison-Wesley Reading, 1995.

[27] S.-E. Bornscheuer, M. Thielscher, Representing concurrent action and solving conflicts, Journal of the IGPL 3 (4) (1996) 355–368.

[28] C. Baral, M. Gelfond, Representing concurrent actions in extended logic programming, in: Proceedings of the 13th International Joint Conference on Artifical Intelligence - Volume 2, IJCAI'93, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993, pp. 866–871.

[29] E. Erdem, V. Patoglu, Applications of action languages in cognitive robotics, in: E. Erdem, J. Lee, Y. Lierler, D. Pearce (Eds.), Correct Reasoning: Essays on Logic-Based AI in Honour of Vladimir Lifschitz, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 229–246.

[30] F. Pirri, R. Reiter, Some contributions to the metatheory of the situation calculus, Journal of the ACM 46 (3) (1999) 261–325.

[31] G. De Giacomo, Y. Lespérance, H. J. Levesque, ConGolog, a concurrent programming language based on the situation calculus, Artificial Intelligence 121 (1–2) (2000) 109–169.

[32] J. Claßen, G. Lakemeyer, A logic for non-terminating Golog programs, in: Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR), 2008, pp. 589–599.

[33] G. De Giacomo, Y. Lespérance, A. R. Pearce, Situation calculus based programs for representing and reasoning about game structures, in: Principles of Knowledge Representation and Reasoning: Proceedings of the Twelfth International Conference, KR 2010, Toronto, Ontario, Canada, May 9-13, 2010.

[34] E. M. Clarke, O. Grumberg, D. A. Peled, Model checking, The MIT Press, Cambridge, MA, USA, 1999.

[35] D. Calvanese, G. De Giacomo, M. Montali, F. Patrizi, First-order μ-calculus over generic transition systems and applications to the situation calculus, Inf. Comput. 259 (3) (2018) 328–347.

[36] P. Felli, L. de Silva, B. Logan, S. M. Ratchev, Composite capabilities for cloud manufacturing, in: Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2018, Stockholm, Sweden, July 10-15, 2018, IFAAMAS, 2018, pp. 1809–1811.

[37] M. Arenas, J. Baier, J. Navarro, S. Sardina, Incomplete causal laws in the situation calculus using free fluents, in: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), New York, USA, 2016, pp. 907–914.

[38] B. Banihashemi, G. De Giacomo, Y. Lespérance, Abstraction in situation calculus action theories, in: Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA., 2017, pp. 1048–1055.

[39] ANSI/ISA, Enterprise-control system integration – Part 1: Models and terminology, ANSI/ISA standard 95.01-2000 (IEC 62264-1 Mod) (2010).

[40] S. Sardiña, G. De Giacomo, Composition of ConGolog programs, in: IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009, pp. 904–910.

[41] J. C. Bradfield, I. Walukiewicz, The mu-calculus and model checking, in: Handbook of Model Checking., 2018, pp. 871–919.

[42] G. De Giacomo, Y. Lespérance, F. Patrizi, S. Sardiña, Verifying ConGolog programs on bounded situation calculus theories, in: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA., 2016, pp. 950–956.

[43] S. Borgo, A. Cesta, A. Orlandini, A. Umbrico, A planning-based architecture for a reconfigurable manufacturing system, in: ICAPS, AAAI Press, 2016, pp. 358–366.

[44] Z. G. Saribatur, V. Patoglu, E. Erdem, Finding optimal feasible global plans for multiple teams of heterogeneous robots using hybrid reasoning: an application to cognitive factories, Auton. Robots 43 (1) (2019) 213–238.

[45] A. Ciortea, S. Mayer, F. Michahelles, Repurposing manufacturing lines on the fly with multi-agent systems for the web of things, in: AAMAS, International Foundation for Autonomous Agents and Multiagent Systems Richland, SC, USA / ACM, 2018, pp. 813–822.

[46] R. F. Kelly, A. R. Pearce, Property persistence in the situation calculus, Artificial Intelligence 174 (12-13) (2010) 865–888.

[47] T. Hofmann, T. Niemueller, J. Claßen, G. Lakemeyer, Continual planning in Golog, in: AAAI, AAAI Press, 2016, pp. 3346–3353.

[48] L. Xiong, Y. Liu, Strategy representation and reasoning in the situation calculus, in: ECAI, Vol. 285 of Frontiers in Artificial Intelligence and Applications, IOS Press, 2016, pp. 982–990.

[49] C. Schwering, G. Lakemeyer, M. Pagnucco, Belief revision and projection in the epistemic situation calculus, Artif. Intell. 251 (2017) 62–97.

[50] M. Arenas, J. A. Baier, J. S. Navarro, S. Sardiña, On the progression of situation calculus universal theories with constants, in: KR, AAAI Press, 2018, pp. 484–493.

[51] A. M. MacNally, N. Lipovetzky, M. Ramírez, A. R. Pearce, Action selection for transparent planning, in: AAMAS, International Foundation for Autonomous Agents and Multiagent Systems Richland, SC, USA / ACM, 2018, pp. 1327–1335.

[52] K. E. Baldwin, Autonomous manufacturing systems, in: Proceedings of the IEEE International Symposium on Intelligent Control, 1989, pp. 214–220.

[53] J. Lee, H. D. Ardakani, S. Yang, B. Bagheri, Industrial big data analytics and cyber-physical systems for future maintenance & service innovation, Procedia CIRP 38 (2015) 3–7, proceedings of the 4th International Conference on Through-life Engineering Services.

[54] M. Behandish, S. Nelaturi, J. de Kleer, Automated process planning for hybrid manufacturing, Computer-Aided Design 102 (2018) 115–127.

[55] A. Gabaldon, Non-markovian control in the situation calculus, Artif. Intell. 175 (1) (2011) 25–48.

[56] P. Mo, N. Li, Y. Liu, Automatic verification of Golog programs via predicate abstraction, in: ECAI, Vol. 285 of Frontiers in Artificial Intelligence and Applications, IOS Press, 2016, pp. 760–768.

[57] J. Claßen, M. Liebenberg, G. Lakemeyer, B. Zarrieß, Exploring the boundaries of decidable verification of non-terminating Golog programs, in: Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada., 2014, pp. 1012–1019.

[58] J. Claßen, Symbolic verification of Golog programs with First-Order BDDs, in: KR, AAAI Press, 2018, pp. 524–529.

[59] A. Camacho, C. J. Muise, J. A. Baier, S. A. McIlraith, LTL realizability via safety and reachability games, in: Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI 2018), 2018, pp. 4683–4691.