Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence (IJCAI-22)
Journal Track

Situation Calculus for Controller Synthesis in Manufacturing Systems with First-Order State Representation (Extended Abstract)*

Giuseppe De Giacomo[1] , **Paolo Felli**[2] , **Brian Logan**[3] , **Fabio Patrizi**[1] and **Sebastian Sardiña**[4]
[1]Sapienza University of Rome, Italy
[2]Free University of Bozen-Bolzano, Italy
[3]Utrecht University, Netherlands, University of Aberdeen, UK
[4]RMIT University, Australia
degiacomo@diag.uniroma1.it, pfelli@unibz.it, b.s.logan@uu.nl, patrizi@diag.uniroma1.it, sebastian.sardina@rmit.edu.au

Abstract

Manufacturing is transitioning from a mass production model to a service model in which facilities 'bid' for products. To decide whether to bid for a previously unseen product, a facility must be able to synthesize, on the fly, a process plan controller that delegates abstract manufacturing tasks in the process recipe for the product to the available manufacturing resources. First-order state representations are commonly used in reasoning about action in AI. Here we show that we can leverage the extensive literature on the Situation Calculus to automatically synthesize such controllers. We identify two important decidable cases—finite domains and bounded action theories—for which we provide practical synthesis techniques.

1 Introduction

In the manufacturing as a service (MaaS) paradigm, manufacturing infrastructure is shared on-demand by a potentially large number of different manufacturing processes. The cost of managing and maintaining the manufacturing infrastructure is thus distributed across all customers, enhancing resource utilization and reducing unit production costs. Different manufacturing models have been proposed in the literature to achieve the MaaS vision, with an emphasis on flexibility, scalability, adaptability and customization, and the increased use of automation and data and knowledge sharing through the supply chain. In this paper, we focus on Cloud Manufacturing [Xu, 2012; Lu *et al.*, 2014]. Enabled by an recent developments in information technology, IoT, embedded systems and cloud computing technologies, Cloud Manufacturing is 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. Similarly, the abstract manufacturing tasks required to manufacture a product are specified as abstract, *system-independent* processes that need to be matched against the abstract capability descriptions offered by manufacturing facilities in the cloud. This allows the creation of dynamic production lines on-demand, in a pay-as-you-go business model.

In mass production, the process planning phase, which transforms a process specification into a *process plan* specifying concrete production schedules for the resources on the shop floor, is carried out by manufacturing engineers, and is largely a manual activity. This is not feasible in MaaS, where manufacturing facilities must be able to *automatically synthesize process plans* for novel products 'on the fly'. This requires matching the abstract manufacturing tasks in the *process recipe*—the specification of how the product is to be manufactured—against the available manufacturing resources in the facility. The resulting process plan details the low-level tasks to be executed and their order, the resources to be used and how materials and parts move between them [Groover, 2007]. The *process plan controller*, i.e., the control software that delegates each operation in the plan to the appropriate manufacturing resources, is then synthesized.

Research in AI 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. Recent work on MaaS based on fundamental ideas from the literature on service composition in CS and behavior composition in AI [De Giacomo *et al.*, 2013], has shown how the requirements and techniques for the automated synthesis of process plan controllers can be formalized [de Silva *et al.*, 2016; Felli *et al.*, 2016; Felli *et al.*, 2017; De Giacomo *et al.*, 2018; De Giacomo *et al.*, 2019]. While these approaches have proven fruitful for developing 'proof-of-concept' MaaS implementations, they are based on a *propositional description* of states, which is often too idealized. In many cases, manufacturing processes depend on the *objects* and *data* they produce and consume, and in general an *unbounded* number of product items or basic parts may be produced. This requires a rich, relational description of states, an *information model*, and computational techniques able to manipulate such representations. Although some work exists that provides a basis for an unambiguous description of the manufacturing concepts [Grüninger and Menzel, 2003], the scientific literature has been lacking.

We propose a relational representation of states by drawing on research on reasoning about actions in AI. Opera-

*This paper is an abridged version of [De Giacomo et al., 2022].

tions in manufacturing processes are described by an action theory in logic, and manufacturing processes are formalized as high-level programs over such action theories. In this way, we 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; instead we encode information models and how they change as the result of actions in the Situation Calculus. Process recipes and manufacturing resources, in turn, are modeled as high-level ConGolog programs [Levesque *et al.*, 1997] (over the action theories). Moreover, 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. 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 correspond to algorithms for extracting the actual controllers when the resulting Situation Calculus action theories are *state-bounded* [De Giacomo *et al.*, 2016a]. In our context, state-boundedness means that, while the facility may process an infinite number of objects overall, an unbounded number of objects is never "accumulated", i.e., in any given state the number of objects being processed does not exceed a given bound. This is the typical case in practice: the number of objects handled *at a given time* by a facility is naturally bounded by the size and structure of the shop-floor.

We stress that, independently of the application domain we consider, we provide here the first decidability result for controller synthesis in a setting with a relational/first-order state representation. While our approach 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 frameworks in databases [Hariri *et al.*, 2013; Deutsch *et al.*, 2018].

While this paper illustrates the main elements of our framework and the main results, the complete details can be found in the full paper [De Giacomo *et al.*, 2022].

2 Manufacturing as a Service in SitCalc

A *basic action theory (BAT)* [Reiter, 2001] 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 countably infinite object sort Δ, for which we assume unique names and domain closure [Levesque and Lakemeyer, 2001; Sardina *et al.*, 2004]. Employing *standard names* for objects, we fix a single interpretation domain for models of situation calculus formulas and blur the distinction between such names and domain objects.

Manufacturing activities are modeled as action types, each taking a tuple of objects as arguments. For example, $\text{DRILL}(part, dmtr, speed, x, y, z)$ represents the action of drilling a hole of a certain diameter, with a certain spindle speed, in a specific position of a given part. Since we are concerned with operations that may occur simultaneously [Reiter, 2001; Bornscheuer and Thielscher, 1996; Baral and Gelfond, 1993], we adopt the *concurrent, non-temporal* variant of the Situation Calculus, where a *compound* action $\boldsymbol{a}$ is a set of simple actions that execute simultaneously [Reiter, 2001, Chapter 7]. E.g., $\{\text{ROTATE}(part, speed), \text{PAINT}(part, color)\}$ represents the joint execution of rotating a part while spraying paint on it.

Axioms are used to specify preconditions and effects of *sets* of actions, thus offering complete control in modeling manufacturing facilities and allowing the expression of arbitrary constraints on the available resources. For example, two robots may be allowed to lift a heavy object only at the same time, while they might be prevented (by their respective theories) from doing so individually.

$\mathbf{A}(\mathbf{x})$ denotes the compound action type $\mathbf{A}$ with parameters $\mathbf{x}$. Action instance $\boldsymbol{a}$ has precondition axioms and successor state axioms of the form $\mathit{Poss}(\boldsymbol{a}, s) \equiv \varphi(\boldsymbol{a}, s)$ and $f(\mathbf{x}, do(\boldsymbol{a}, s)) \equiv \varphi(\mathbf{x}, \boldsymbol{a}, s)$, with $\varphi(\boldsymbol{a}, s)$ and $\varphi(\mathbf{x}, \boldsymbol{a}, s)$ *uniform* in the current situation s. We assume complete information on the initial situation S^0, which makes our BATs *categorical*, i.e., admitting a single model [Reiter, 2001; Sardina *et al.*, 2004]. Observe that, although unique, the BAT model has an infinite object domain, as well as infinite situations, which makes it nontrivial to deal with, and requires a substantially different approach to that adopted in model checking [Clarke *et al.*, 1999; De Giacomo *et al.*, 2016a; Calvanese *et al.*, 2018].

2.1 ConGolog High-level Programs

Several *high-level* programming languages have been proposed based on the Situation Calculus, including Golog [Levesque *et al.*, 1997], which supports both standard programming and nondeterministic-choice constructs, ConGolog [De Giacomo *et al.*, 2000], which extends Golog with concurrency, and IndiGolog [Sardina *et al.*, 2004], which supports interleaved planning and execution. We specify manufacturing processes as programs in a variant of ConGolog without recursive procedures [De Giacomo *et al.*, 2000] and where the test construct yields no transition and is final when satisfied [Claßen and Lakemeyer, 2008; De Giacomo *et al.*, 2010]. This results in a *synchronous test* construct in which interleaving is disallowed (every transition involves the execution of one action).

All standard ConGolog constructs are allowed: simple/compound actions $\boldsymbol{a}$, test $\phi?$, sequence $\delta_1; \delta_2$, nondeterministic branching $\delta_1 \mid \delta_2$, nondeterministic argument choice $\pi x.\delta$, nondeterministic iteration δ^*, conditional constructs, while loops, and interleaved concurrency $\delta_1 \| \delta_2$. A program δ is executed over a BAT $\mathcal{D}$, which must include the fluents and the constants mentioned in δ, with the latter coming from the set $AC_{\mathcal{D}}$ of $\mathcal{D}$'s active object constants. A *configuration* is a pair $\langle \delta, s \rangle$ with δ a program and s a situation. ConGolog's semantics is specified in terms of single-steps, using predicates $\mathit{Final}(\delta, s)$, which specifies when a configuration $\langle \delta, s \rangle$ is *final* (i.e., δ may terminate in s), and $\mathit{Trans}(\delta, s, \delta', s')$, which specifies the one-step transition from $\langle \delta, s \rangle$ to $\langle \delta', s' \rangle$, where δ' remains to be executed [De Giacomo *et al.*, 2000].

The definitions of *Trans* and *Final* for the ConGolog con-

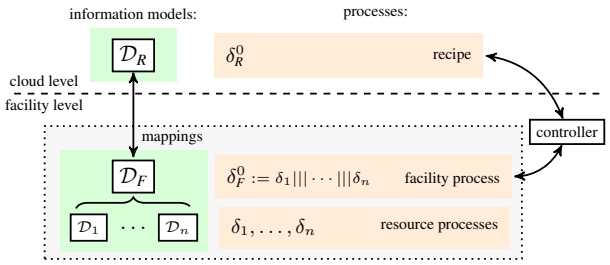

Figure 1: Framework for MaaS, divided into a cloud level and a facility level (only one facility is shown).

structs above are standard, but, as discussed earlier, cannot express simultaneous execution of compound actions. To address this, we extend ConGolog with the *synchronized concurrency* operator $\delta_1|||\delta_2$, which states that programs δ_1 and δ_2 execute concurrently and synchronously, i.e., their next actions take place in the *same* transition step. The semantics is as follows: $\mathit{Final}(\delta_1|||\delta_2, s) \equiv \mathit{Final}(\delta_1, s) \wedge \mathit{Final}(\delta_2, s)$ and $\mathit{Trans}(\delta_1|||\delta_2, s, \delta', s') \equiv \mathit{Trans}'(\delta_1, s, \delta_1', s_1') \wedge s_1' = (\boldsymbol{a}_1, s) \wedge \mathit{Trans}'(\delta_2, s, \delta_2', s_2') \wedge s_2' = (\boldsymbol{a}_2, s) \wedge \mathit{Poss}(\boldsymbol{a}_1 \cup \boldsymbol{a}_2, s) \wedge \delta' = (\delta_1'|||\delta_2') \wedge s' = (\boldsymbol{a}_1 \cup \boldsymbol{a}_2, s)$. Informally, Trans' is the same axiom as Trans, but it only requires the executability of compound actions, without requiring the executability of (subsets of) their component simple actions. Indeed, we state $\mathit{Poss}(\boldsymbol{a}_1 \cup \boldsymbol{a}_2, s)$ but not $\mathit{Poss}(\boldsymbol{a}_1, s)$ or $\mathit{Poss}(\boldsymbol{a}_2, s)$. As a result, a number of sub-systems (manufacturing resources) can legally perform a joint step only if this is explicitly stated to be possible by a "global" BAT for compound actions (the BAT for the entire facility, see the next section). This gives great flexibility when modeling manufacturing facilities. Note that synchronized concurrency is not reducible to interleaved concurrency: $\delta_1\|\delta_2$ allows either δ_1 or δ_2 to be executed completely before starting the other, and for them to be executed alternately.

2.2 Manufacturing as a Service

A product can be manufactured if an implementation of all the possible sequences of resource-independent operations prescribed by the recipe, together with any additional low-level operations (not included in the recipe) required by the implementation, can be delegated step-by-step to the facility resources. Recall that process recipes are *resource-independent*, i.e., specified without knowing the actual manufacturing system that will be used for their realization, but assuming only an information model common throughout the cloud. When a product is manufacturable in a facility, a *controller* responsible for delegating recipe actions to facility resources can be synthesized.

The resulting MaaS framework, shown in Figure 1, includes two sorts: *(i) Information models*, i.e., BATs $\mathcal{D}$ describing the data and physical objects processes manipulate and the operations to manipulate them; and *(ii) Processes*, i.e., programs δ_R^0 describing the specific capabilities and dynamics of components.

The framework comprises the following components (for simplicity, we restrict to the case of one facility in the cloud). *(i)* **Resources**: the facility manufacturing resources. Each resource is a pair $\langle \mathcal{D}_i, \delta_i \rangle$, with information model $\mathcal{D}_i$ and resource process δ_i. *(ii)* **Facility information model**: the information model $\mathcal{D}_F$ obtained by combining the BATs $\mathcal{D}_i$ of each resource on the shop floor, see the paper for details. This includes specifying the executability of compound actions (see Sec. 2.1). *(iii)* **Facility process**: the process $\delta_F^0 := \delta_1|||\cdots|||\delta_n$, specifying the synchronous execution of the resource processes (synchronous concurrency allows different resources to execute actions at the same time). *(iv)* **Cloud information model**: the common resource-independent information model $\mathcal{D}_R$ assumed by all recipes, representing the data and objects recipes manipulate. *(v)* **Mappings**: a set of mappings, Maps, relating the resource-independent cloud information model $\mathcal{D}_R$ to the resource-dependent facility information model $\mathcal{D}_F$. Maps relates the abstract executions described by the process recipe δ_R^0 (see below) to the concrete executions of the facility process δ_F^0. Adopting ideas from [Banihashemi *et al.*, 2017], two forms of mappings are used. (1) For each fluent f in $\mathcal{D}_R$ with parameters $\mathbf{x}$, the atomic formula $f(\mathbf{x}, s_R)$ is mapped to a (uniform) formula $\varphi_f(\mathbf{x}, s_F)$ over the fluents in $\mathcal{D}_R$. This formula is domain-independent: its evaluation depends only on the objects occurring in the extension of $\mathcal{D}_F$'s fluents in the current situation. Moreover, a subset *Obs* of $\mathcal{D}_R$'s recipe fluents act as *observations*: they have no successor-state axiom and their extension is provided by Maps. For fluents not in *Obs*, Maps imposes a consistency requirement between the two theories. (2) For each action type $\mathbf{A} \in \mathcal{A}_R$ with parameters $\mathbf{x}$, we map $\mathbf{A}(\mathbf{x})$ to a (arbitrarily complex) program $\delta_{\mathbf{A}}(\mathbf{x})$ for $\mathcal{D}_F$ which makes use of (compound) actions of the available resources. The combination of Maps with $\mathcal{D}_R$ and $\mathcal{D}_F$ produces a new theory $\mathcal{D}_R^{\mathit{Maps}}$, which is not a traditional Situation Calculus theory, as it includes two completely independent situation sorts (instead of one): 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 how to obtain $\mathcal{D}_R^{\mathit{Maps}}$, see the paper. *(vi)* **Facility**: a manufacturing facility is a tuple $\mathit{Fac} = \langle \mathcal{D}_R, \mathcal{D}_F, \delta_F^0, \mathit{Maps} \rangle$, with cloud information model $\mathcal{D}_R$, facility information model $\mathcal{D}_F$, facility program δ_F^0, and mappings Maps. *(vii)* **Recipe**: a resource-independent process δ_R^0 over the cloud information model $\mathcal{D}_R$, describing the process to execute to manufacture a product. *(viii)* **Controller**: when the product is manufacturable, the controller is responsible for orchestrating the resources. Informally, it is a function relating each execution of the recipe δ_R^0 to an execution of the facility process δ_F^0.

Given a facility Fac and a recipe δ_R^0, the *manufacturability problem* amounts to establishing whether there exists a controller to orchestrate the resources in the Fac to realize δ_R^0. The *controller synthesis task* is to automatically build the controller responsible for implementing the orchestration.

3 Controller Synthesis

In order to formalize when a recipe can be realized by a facility, we introduce three properties relating recipe and facility configurations $\langle \delta_R, s_R \rangle$ and $\langle \delta_F, s_F \rangle$. *(i)* **Mappings' preservation**: the value of every non-observation fluent $f \notin \mathit{Obs}$ of $\mathcal{D}_R$ in $\langle \delta_R, s_R \rangle$ is compatible, through the mapping $f(\mathbf{x}) \leftrightarrow$

$\varphi_f(\mathbf{x})$ in $Maps$, with the value of $\varphi_f(\mathbf{x})$ in $\langle \delta_F, s_F \rangle$. For every $f \in Obs$, the mapping is respected. *(ii)* **Legal termination**: $Final(\delta_R, s_R)$ implies $Final(\delta_F, s_F)$, i.e., if the recipe can legally terminate, so can the resources. *(iii)* **Recipe actions realizability**: for every abstract action $\mathbf{A}(\mathbf{x})$ in the recipe executable in $\langle \delta_R, s_R \rangle$, there exists a program $\delta_{\mathbf{A}}(\mathbf{x})$, determined through the mapping $\mathbf{A}(\mathbf{x}) \leftrightarrow \delta_{\mathbf{A}}(\mathbf{x})$, that is executable in its entirety from $\langle \delta_F, s_F \rangle$ to some $\langle \delta'_F, s'_F \rangle$, representing the complete *implementation* of $\mathbf{A}(\mathbf{x})$ in the facility. This captures the synchronization of the recipe and facility situations: the recipe situation $do(\mathbf{A}(\mathbf{x}, s'_F), s_R)$ resulting from the execution of $\mathbf{A}(\mathbf{x})$ in s_R depends on the new situation s'_F reached by the facility after the execution of $\delta_{\mathbf{A}}(\mathbf{x})$.

Realizability of a recipe by a given facility can be formalized by co-induction, by defining the *largest* realizability relation $\preceq$ between facility and recipe configurations that satisfies the three requirements above, and such that whenever the recipe executes an action and the facility executes a corresponding program through the mappings (see third point), the new situations are still in $\preceq$. Details are in the paper.

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

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 $Trans(\delta_R, s_R, \delta'_R, (\mathbf{A}(\mathbf{x}, S_F^0), s_R))$ (here S_F^0 is used as a placeholder, and does not affect δ'_R), returns a sequence of facility configurations $\langle \delta_F^0, s_F^0 \rangle \dots \langle \delta_F^m, s_F^m \rangle$, such that:

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

$Do(\delta, s, s')$ abbreviates $\exists \delta'.\, Trans^*(\delta, s, \delta', s') \wedge Final(\delta', s')$, which states that the *complete execution* of δ from s results in s' [Levesque *et al.*, 1997; De Giacomo *et al.*, 2000].

To check realizability, we define a *two-player game* between ENVIRONMENT (the antagonist) and CONTROLLER (the controller), which is played over a *game arena* (GA) $\mathcal{T}$, i.e., a labelled transition system over the vocabulary of fluents and constants from the active domain. The states of the arena are partitioned (using two propositions $turnEnv$ and $turnCtrl$) so that in each state only one player can move. The state labeling of the GA holds all the information about the (current) configurations of the recipe and facility. Technically, this requires decoupling the "data" from the control-flow, i.e., the program counter [De Giacomo *et al.*, 2016b]. Having adopted standard names, program counters and actions can then be treated as active constants and objects.

Intuitively, the game proceeds as follows: 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 $\mathcal{D}_R^{Maps}$ and finally passes the turn to CONTROLLER, which chooses one among the actions that are currently legal for both δ_F (initially δ_F^0) and $\delta_{\mathbf{A}}(\mathbf{x})$ in their current configuration. A step in $\delta_{\mathbf{A}}(\mathbf{x})$ is thus executed, and CONTROLLER aligns the current cloud situation s_R with the resulting factory situation s'_F (since the interpretation of $\mathcal{D}_R$'s non-observation fluents is not affected by the $\mathcal{D}_F$'s situation argument in $\mathbf{A}$). Then, CONTROLLER can (but does not have to) pass the turn to ENVIRONMENT only when δ has reached a final configuration.

The paper shows that a realizability relation between δ_R^0 and δ_F^0 exists iff $\mathcal{T}$ satisfies the μ-calc (in fact, $\mu\mathcal{L}_c$) formula:

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

where, informally, 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. Φ_{Real} is true in all those states from which CONTROLLER can force the game to visit infinitely often a state where ϕ_{OK} holds, no matter how ENVIRONMENT plays. Φ_{Real} also requires that CONTROLLER does not pass the turn until ϕ_{OK} holds.

The set $Win(\Phi_{Real})$ of *winning states* is the set of states where Φ_{Real} holds, so the objective of CONTROLLER is to maintain the game within such a *winning* region. When CONTROLLER has a (memoryless) *winning strategy*, i.e., a function mapping each $\mathcal{T}$ state into a new state (i.e., a game move, corresponding to a facility action execution) from the winning region, a controller can be computed.

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 δ_F^0 iff $q_0 \in Win(\Phi_{Real})$.*

The paper shows how to extract a controller from the set $Win(\Phi_{Real})$ of winning states.

4 Bounded Case: Decidable Synthesis

The paper analyzes the case of practical interest where the facility and the recipe induce a GA $\mathcal{T}$ that is both *state-bounded* and *generic*. As explained in Section 1, the former property requires state-boundedness of both $\mathcal{D}_R$ and $\mathcal{D}_F$, as well as of all $\mathcal{D}_R$'s observation fluents. The latter, which is implied by the use of BATs, requires that, whenever two states are isomorphic, they yield the same transitions modulo the same object renaming induced by the isomorphism.

Theorem 2. *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.*

We prove that, given a generic, state-bounded GA $\mathcal{T}$, there exists a finite-state GA $\bar{\mathcal{T}}$ (used as faithful abstraction) such that, for every $\mu\mathcal{L}_c$ formula Φ, $\mathcal{T} \models \Phi$ iff $\bar{\mathcal{T}} \models \Phi$. Hence we can model-check Φ_{Real} on $\bar{\mathcal{T}}$ rather than on $\mathcal{T}$. Finally, we show a constructive way of transforming a memoryless strategy as above into a controller for δ_R^0 that realizes δ_F^0.

5 Conclusions

In this paper we illustrated all the main ideas and results of our approach for the synthesis of controllers for manufacturing systems in the Situation Calculus. Details can be found in [De Giacomo *et al.*, 2022].

References

[Banihashemi *et al.*, 2017] Bita Banihashemi, Giuseppe De Giacomo, and Yves Lespérance. Abstraction in situation calculus action theories. In *Proc. of the Thirty-First AAAI Conference on Artif. Intelligence*, pages 1048–1055, 2017.

[Baral and Gelfond, 1993] Chitta Baral and Michael Gelfond. Representing concurrent actions in extended logic programming. In *Proc. of the 13th Int. Joint Conference on Artifical Intelligence*, pages 866–871, 1993.

[Bornscheuer and Thielscher, 1996] Sven-Erik Bornscheuer and Michael Thielscher. Representing concurrent action and solving conflicts. *J. of the IGPL*, 3(4):355–368, 1996.

[Calvanese *et al.*, 2018] Diego Calvanese, Giuseppe De Giacomo, Marco Montali, and Fabio Patrizi. First-order μ-calculus over generic transition systems and applications to the situation calculus. *Inf. Comput.*, 259(3):328–347, 2018.

[Clarke *et al.*, 1999] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. The MIT Press, Cambridge, MA, USA, 1999.

[Claßen and Lakemeyer, 2008] Jens Claßen and Gerhard Lakemeyer. A logic for non-terminating Golog programs. In *Proc. of KR of the Int. Conf. on Principles of Knowledge Representation and Reasoning (KR)*, pages 589–599, 2008.

[De Giacomo *et al.*, 2000] Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1–2):109–169, 2000.

[De Giacomo *et al.*, 2010] Giuseppe De Giacomo, Yves Lespérance, and Adrian R. Pearce. Situation calculus based programs for representing and reasoning about game structures. In *Proc. of KR of the Int. Conf. on Principles of Knowledge Representation and Reasoning (KR)*, 2010.

[De Giacomo *et al.*, 2013] Giuseppe De Giacomo, Fabio Patrizi, and Sebastian Sardiña. Automatic behavior composition synthesis. *Artificial Intelligence*, 196:106–142, 2013.

[De Giacomo *et al.*, 2016a] Giuseppe De Giacomo, Yves Lespérance, and Fabio Patrizi. Bounded situation calculus action theories. *Artif. Intell.*, 237:172–203, 2016.

[De Giacomo *et al.*, 2016b] Giuseppe De Giacomo, Yves Lespérance, Fabio Patrizi, and Sebastian Sardiña. Verifying ConGolog programs on bounded situation calculus theories. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pages 950–956, 2016.

[De Giacomo *et al.*, 2018] Giuseppe De Giacomo, Moshe Vardi, Paolo Felli, Natasha Alechina, and Brian Logan. Synthesis of orchestrations of transducers for manufacturing. In *Proc. of the Thirty-Second AAAI Conference on Artificial Intelligence*, pages 6161–6168. AAAI Press, 2018.

[De Giacomo *et al.*, 2019] Giuseppe De Giacomo, Natasha Alechina, Tomas Brazdil, Paolo Felli, Brian Logan, and Moshe Vardi. Unbounded orchestrations of transducers for manufacturing. In *Proc. of the Thirty-Third AAAI Conference on Artificial Intelligence*. AAAI Press, 2019.

[De Giacomo *et al.*, 2022] Giuseppe De Giacomo, Paolo Felli, Brian Logan, Fabio Patrizi, and Sebastian Sardiña. Situation calculus for controller synthesis in manufacturing systems with first-order state representation. *Artificial Intelligence*, 302:103598, 2022.

[de Silva *et al.*, 2016] Lavindra de Silva, Paolo Felli, Jack C. Chaplin, Brian Logan, David Sanderson, and Svetan Ratchev. Realisability of production recipes. In *Proc. of ECAI*, pages 1449–1457. IOS Press, 2016.

[Deutsch *et al.*, 2018] Alin Deutsch, Richard Hull, Yuliang Li, and Victor Vianu. Automatic verification of database-centric systems. *SIGLOG News*, 5(2):37–56, 2018.

[Felli *et al.*, 2016] Paolo Felli, Brian Logan, and Sebastian Sardina. Parallel behavior composition for manufacturing. In Subbarao Kambhampati, editor, *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 271–278, 2016.

[Felli *et al.*, 2017] Paolo Felli, Lavindra de Silva, Brian Logan, and Svetan M. Ratchev. Process plan controllers for non-deterministic manufacturing systems. In *Proc. of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1023–1030, 2017.

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

[Grüninger and Menzel, 2003] Michael Grüninger and Christopher Menzel. The process specification language (PSL) theory and applications. *AI Magazine*, 24:63–74, 2003.

[Hariri *et al.*, 2013] Babak Bagheri Hariri, Diego Calvanese, Giuseppe De Giacomo, Alin Deutsch, and Marco Montali. Verification of relational data-centric dynamic systems with external services. In *Proc. of PODS*, pages 163–174, 2013.

[Levesque and Lakemeyer, 2001] Hector J. Levesque and Gerhard Lakemeyer. *The Logic of Knowledge Bases*. The MIT Press, 2001.

[Levesque *et al.*, 1997] Hector J. Levesque, Ray Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31:59–84, 1997.

[Lu *et al.*, 2014] Yuqian Lu, Xun Xu, and Jenny Xu. Development of a hybrid manufacturing cloud. *Journal of Manufacturing Systems*, 33(4):551–566, 2014.

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

[Sardina *et al.*, 2004] Sebastian Sardina, Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque. On the semantics of deliberation in IndiGolog – From theory to implementation. *Annals of Mathematics and Artificial Intelligence*, 41(2–4):259–299, August 2004.

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