Situation Calculus for Synthesis of Manufacturing Controllers

Giuseppe De Giacomo
Sapienza Università di Roma, Italy
degiacomo@dis.uniroma1.it

Paolo Felli
Free University of Bozen-Bolzano, Italy
paolo.felli@unibz.it

Brian Logan
University of Nottingham, UK
bsl@cs.nott.ac.uk

Fabio Patrizi
Sapienza Università di Roma, Italy
patrizi@dis.uniroma1.it

Sebastian Sardina
RMIT University, Australia
sebastian.sardina@rmit.edu.au

Abstract

Manufacturing is transitioning from a mass production model to a manufacturing as a service model in which manufacturing facilities ‘bid’ to produce products. To determine whether to bid for a complex, previously unseen product, a manufacturing facility must be able to synthesize, ‘on the fly’, a process plan controller that delegates abstract manufacturing tasks in the supplied process recipe to the appropriate manufacturing resources, e.g., CNC machines, robots etc. Previous work in applying AI behaviour composition to synthesize process plan controllers has considered only finite state ad-hoc representations. Here, we study the problem in the relational setting of the Situation Calculus. By taking advantage of recent work on abstraction in the Situation Calculus, process recipes and available resources are represented by ConGolog programs, respectively, an abstract and a concrete action theory. This allows us to capture the problem in a formal, general framework, and show decidability for the case of bounded action theories. We also provide techniques for actually synthesizing the controller.

Introduction

Manufacturing is transitioning from a mass production model to a service model in which manufacturing facilities ‘bid’ to produce products. In contrast to mass production, where large volumes of known products are produced at a time, in manufacturing as a service, the products to be manufactured are not known in advance, batch sizes are often small, and a facility may produce products for several customers at the same time (Technology Strategy Board 2012; Rhodes 2015). This trend towards rapid provisioning of manufacturing resources, e.g., CNC machines and robots, with minimal management effort or service provider interaction has been termed ‘cloud manufacturing’ (Xu 2012; Lu, Xu, and Xu 2014).

To determine if a novel product can be manufactured, abstract manufacturing tasks in the process recipe specifying how a product is manufactured must be matched against the available manufacturing resources so as to produce a process plan detailing the low-level tasks to be executed and their order, the manufacturing resources to be used, and how materials and parts move between resources (Groover 2007).

Control software—called process plan controller—that delegates each operation in the plan to the appropriate manufacturing resources is then synthesized. In mass production, process planning is carried out by manufacturing engineers and is largely a manual process. However this is uneconomic for the small batch sizes typical of the manufacturing as a service model, and the time required to produce a plan is too great to allow facilities to bid for products in real time. To fully realize the manufacturing as a service vision, manufacturing facilities must be able to automatically synthesize process plan controllers for novel products ‘on the fly’.

There has recently been efforts to apply AI behavior composition (De Giacomo, Patrizi, and Sardina 2013) to the synthesis of process plan controllers, e.g., (de Silva et al. 2016; Felli, Logan, and Sardina 2016; Felli et al. 2017; De Giacomo et al. 2018). However, this work suffers from two important limitations. First, the approaches are restricted to finite state representations. While adequate for some manufacturing tasks and resources, the resulting discretization is unwieldy and less natural, for example, when representing and reasoning about the potentially infinite number of basic parts, each with a unique bar code or RFID tag. Secondly, existing approaches to modeling abstract manufacturing tasks and the interactions between resources result in somewhat ad-hoc and inflexible formalisms. For example, the state (and its dynamics) of a given part (e.g., painted, defective, etc.) or of a shared resource (e.g., a conveyor belt) ought to be encoded into operational representations, such as transition systems or automata. Clearly, more declarative representations would be desirable.

In this paper, we address both issues by adopting the Situation Calculus (Reiter 2001), a knowledge representation formalism to reason about action and change. We represent process recipes and available resources as high-level ConGolog programs executing over, respectively, an abstract and a concrete action theory. This yields a principled, formal, and declarative representation of the manufacturing setting. We then define, by means of a suitable simulation relation, what it means to realize a product process recipe on the available resources. Finally, by leveraging recent work on abstraction, we show how to effectively check the existence of a process plan controller (and how to compute it) for the case of bounded action theories.
Situation Calculus

The situation calculus (McCarthy and Hayes 1969; Reiter 2001) is a logical language for representing and reasoning about dynamically changing worlds in which all changes are the result of actions. We assume to have a finite number of action types, each of which takes a tuple of objects as arguments. For example, `drill(part, dntr, speed, x, y, z)` represents the (simple) action of drilling a hole of a certain diameter at a certain speed in a given part. In the manufacturing domain we are concerned with operations that may occur simultaneously.

We therefore adopt the concurrent non-temporal variant of the situation calculus (Reiter 2001) Chapter 7), where a concurrent or compound action `a` is a, possibly infinite, set of simple actions `α` that execute simultaneously. For example, `{rotate(part, speed), spray(part)}` represents the joint execution of rotating a part at a given speed while spraying it. Situations denote possible sequences of concurrent actions: the constant `S0` denotes the initial situation, on which we assume to have complete information, and the situation resulting from executing a concurrent action `a` in a situation `s` is represented as situation term `do(a, s)`. Predicates whose value varies from situation to situation are called fluents, and they take arguments of sort object plus a situation term as their last argument. For example, `painted(part, s)` may denote that a part is painted in situation `s`.

A basic action theory (BAT) (Pirri and Reiter 1999; Reiter 2001) is a collection of axioms `D` describing the preconditions and effects (and non-effects) of actions on fluents. A special predicate `Poss(a, s)` is used to state that the simple action `a` is executable in situation `s`, and, for each simple action type, a precondition axiom is given to specify when the action can be legally performed. Such a predicate is extended to compound actions `Poss(a, s)`, typically by requiring that each atomic action in `a` is possible, i.e., `Poss(a, s)`, although one can further restrict `Poss` when needed. We also assume that `Poss({a}, s) ≡ Poss(a, s)`. A successor state axiom is used to specify how each fluent changes as the result of executing (simple or) concurrent actions in the domain. Successor state axioms thus encode the causal laws of the domain being modelled, by encoding the effects of actions. Figures 1 and 2 list examples of precondition and successor state axioms for a manufacturing setting.

A Variant of ConGolog for Manufacturing

High-level programs are used to specify complex processes in the domain. We specify programs in (recursion-free) ConGolog (De Giacomo, Lespérance, and Levesque 2000):

- `α` compound action
- `φ?` test for a condition
- `δ1; δ2` sequence
- `δ1 ⊤ δ2` nondeterministic branch
- `π.x.δ` nondeterministic choice of argument
- `δ*` nondeterministic iteration
- `if φ then δ1 else δ2` conditional
- `while φ do δ endWhile` while loop

where `α` is a compound action (instead of atomic action as in the original paper) and `φ` is situation-suppressed formula, i.e., a formula in the language with all situation arguments in fluents suppressed. We denote by `φ[s]` the 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 `π.x.δ` range over objects, and occurs in some action term in `δ`, i.e., `π.x.δ` acts as a construct for the nondeterministic choice of action parameters.

The semantics of ConGolog is specified in terms of single-steps, using the following two predicates (De Giacomo, Lespérance, and Levesque 2000):

- `Final(δ, s):` program `δ` may terminate in situation `s`;
- `Trans(δ, s, δ', s')`: 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:

- `Final(a, s) ≡ False`
- `Final(φ?, s) ≡ φ[s]`
- `Final(δ1; δ2, s) ≡ Final(δ1, s) ∧ Final(δ2, s)`
- `Final(δ1(δ2, s)) ≡ Final(δ1, s) ∨ Final(δ2, s)`
- `Final(π.x.δ, s) ≡ ∃x.Final(δ, s)`
- `Final(δ*, s) ≡ True`
- `Final(δ1 || δ2, s) ≡ Final(δ1, s) ∧ Final(δ2, s)`

- `Trans(a, s, δ', s') ≡ s' = do(a, s) ∧ Poss(a, s) ∧ δ' = True?`
- `Trans(φ?, s, δ', s') ≡ False`
- `Trans(δ1; δ2, s, δ', s') ≡ Trans(δ1, s, δ1, s') ∧ δ' = δ1; δ2 ∨ Final(δ1, s) ∧ Trans(δ2, s, δ', s')`
- `Trans(δ1(δ2, s, δ', s')) ≡ Final(δ1, s) ∧ Trans(δ2, s, δ', s')`
- `Trans(π.x.δ, s, δ', s') ≡ ∃x.Trans(δ, s, δ', s')`
- `Trans(δ*, s, δ', s') ≡ Trans(δ, s, δ', s') ∧ δ' = δ''; δ*`
- `Trans(δ1 || δ2, s, δ', s') ≡ Trans(δ1, s, δ1, s') ∧ δ' = δ1 || δ2 ∨ Trans(δ2, s, δ2, s') ∧ δ' = δ1 || δ2`

Note that the conditional and while-loop constructs are definable: if `φ` then `δ1 else δ2` `endIf` = `φ?; δ1; ¬φ?; δ2` and while `φ` do `δ endWhile` = `(φ?; δ); ¬φ?`.

In the manufacturing setting, high-level programs are used to model the logic of manufacturing resources in “isolation”, and synchronized concurrency is needed to represent the operation of two (or more) resources “simultaneously”. For this reason, we introduce a new construct in ConGolog, called synchronized concurrency:

`δ1 || δ2` synchronized concurrency

that represents the synchronized concurrent execution of programs `δ1` and `δ2`: their next corresponding actions take place in the same next transition step. Its semantics is defined as follows:

- `Trans(δ1 || δ2, s, δ', s') ≡ Trans(δ1(δ2, s, δ', s2)) ∧ s' = do(a1, s) ∧ Trans(δ2(δ1, s, δ1, s2)) ∧ s2 = do(a2, s) ∧ Poss(a1 ∪ a2, s) ∧ δ' = (δ1 || δ2) ∧ s' = do(a1 ∪ a2, s)`
where $\text{Trans}'$ is equivalent to $\text{Trans}$ except for the condition $\text{Trans}(a, s, \delta', s')$ which is now $\text{Trans}'(a, s, \delta', s') \equiv s' = \text{do}(a, s) \land \delta = \text{Trans}'?$, and for $\text{Trans}'(\delta || \delta_2, s, \delta', s')$ that is as above but without check of Poss.

The characterization of Final for synchronized concurrency is analogous to that of interleaved concurrency:

$$\text{Final}(\delta || \delta_2, s) \equiv \text{Final}(\delta_1, s) \land \text{Final}(\delta_2, s).$$

Note the underlying assumption here is that a number of sub-systems, manufacturing resources in our setting, can legally perform a joint step if such joint step is deemed possible by the BAT.

### Modelling Manufacturing as a Service

In this section we show how the manufacturing as a service setting can be captured by representing both process recipes and manufacturing systems as ConGolog programs.

#### Resource Programs

We consider a manufacturing system, or simply system, composed of $n$ distinct manufacturing resources, each identified by an index $i \in \{1, n\}$. Each resource is associated to a BAT $D_i$ specifying the fluents, the actions that may be performed, their preconditions and effects. For convenience we assume that actions have the resource index $i$ (a constant) as their last argument.

**Example 1.** Consider a manufacturing cell that performs operations on parts (the example is based on the cell described in [Felli et al., 2018]). Parts have an ID, and parameters such as size, weight, material, etc. The cell consists of five resources, $R_1$ that is a robot that can perform different operations on parts within its operating envelope, by (autonomously) equipping with the appropriate end effector using the action $\text{Equip}(ee, 1)$. By equipping a driller it can drill parts; by equipping a rivet gun it can apply rivets, etc. The drilling operation is modeled as the action $\text{RobotDrill}$ with arguments $\text{part}, \text{bit}, \text{dntr}, \text{speed}, \text{feed}, x, y, z$ for the part ID, the drilling bit ID, the diameter, the spindle speed, the feed rate, and hole position. Fully specified actions are of the form $\text{RobotDrill}(p, \text{bit}, 1, 125., 2, 123, 87, 12, 1)$, with $1$ as last argument. Similarly for $\text{Rivet}(\text{part}, \text{rivet}, x, y, z)$. We model the other resources in a similar way. $R_2$ is a fixture that can perform an action $\text{HoldInPlace}(\text{part}, \text{force}, 2)$, with arguments for the part ID and the clamping force. Resource $R_3$ is another robot that can move parts into and out of the cell from an external conveyor, position a part at a given location relative to another part, and, by equipping 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 $\text{InCell}(\text{part}, \text{weight}, \text{material}, \text{dimx}, \text{dimy}, \text{dimz}, 3)$, $\text{OutCell}(\text{part}, \text{code}, 3)$, $\text{Position}(\text{part}, \text{part2}, x, y, z, 3)$, $\text{ApplyPress}(\text{part}, \text{force}, 3)$, $\text{ApplyPressFlat}(\text{part}, \text{force}, 3)$. Note that information about the weight, material and size of parts loaded into the cell is made available by passing this via arguments to $\text{InCell}$. Resource $R_4$ is a upright drilling machine for drilling parts with high precision. Finally, $R_5$ is a human operator, who operates $R_4$ and who can also bring small parts into and out of the cell, and apply glue to parts with $\text{ApplyGlue}(\text{part}, \text{glue}, \text{type}, s)$. Parts are moved between resources by a part-handling system. For simplicity, we model this using additional actions $\text{IN}(\text{part}, i)$ and $\text{OUT}(\text{part}, i)$ for each resource, denoting that a part is moved into or out of the work area of the resource, respectively. We also have a special action $\text{nop}$ that specifies when a resource may remain idle.

To denote that a part is currently in the work area of a resource $i$, that a hole has been drilled in a part by a resource, or that a part has a certain material, etc. we use situation dependent or independent fluents of the form $\text{at}(\text{part}, i, s)$, $\text{drilled}(\text{part}, \text{hole}, 1)$, $\text{material}(\text{part}, m)$, etc. In Figure 1 we list some examples of preconditions axioms for theories $D_i$.

Given the BAT $D_i$ of a resource (which include the possible actions) we can describe all possible sequences of operations that the resource can execute (in isolation) as a ConGolog program $\delta_i$ with BAT $D_i$.

#### The Available Program

The set of BATs $\{D_1, \ldots, D_n\}$ is then compiled into a single BAT $D_S$ for the entire system, in a semi-automated fashion, e.g., by taking into account knowledge about which resources are connected by the part-handling system, which resources can work on the same parts, etc. To be able to represent dynamic worlds that allow the concurrent execution of multiple actions, we consider compound action terms of the form $A = \{a_1, \ldots, a_k\}$ with $k \leq n$, where each $a_i$ is a basic action term. As shorthand, we denote by $A(x)$ the compound action $A$ with a vector $x$ of arguments (of the right size, and assuming a standard ordering of basic actions). Moreover, to ensure that the resources involved can work on the same parts, we use a special situation-independent predicate $\text{coopMatrix}(i, j)$ specifying that resource $i$ can cooperate with resource $j$.

**Example 2.** The situation $\text{do}(\{\text{RobotDrill}(\text{e}, \ldots, 1), \text{HoldInPlace}(\text{e}, \ldots, 2)\}, s)$ results from the concurrent execution of two actions: $R_1$ drilling a part with ID $\varepsilon$ that is held by the fixture $R_2$. Also, Figure 2 shows a fragment of the resulting $R_5$ for the overall system.

Similarly to the case of a single resource, assuming a set of $n$ BATs for each of the available resources, and the resulting $D_S$ for the entire system, we can capture all the possible executions of the system as a ConGolog program.

**Definition 1 (Available Program).** Given a set of $n$ resources $\delta_i$, $\in \{1, n\}$, the resulting available program is the ConGolog program $\delta_0^n := \delta_1 || \cdots || \delta_n$.

### Target Program

The product recipe specifying the possible way(s) in which a product can be manufactured is a ConGolog program $\delta^\text{product}$ which we call the target program. $\delta^\text{product}$ has its own BAT $D_T$, which is distinct from $D_S$ (for any system $S$). In the manufacturing as a service model, product recipes are resource
**Figure 2:** Above: example precondition axioms for theory parts denoted by \( \text{Poss} \) (Banihashemi, De Giacomo, and Lespérance 2017):

- **Example 2:** The loading of a part, weight, material, \( x, y, z \), \( i \) is mapped to a (possibly complex) program \( \delta_\text{S} \) of the system. In practice, only cases in which fluents are affected are shown.

**Figure 3:** Example of target program \( \delta_\text{S} \).

**Example 3:** An example target program specified using the resource independent BAT \( \delta_\text{S} \) is shown in Figure 3. Two parts denoted by \( b \) and \( z \) are loaded into the cell, then glue is applied to \( b \) and it is placed on \( z \), resulting in a composite part denoted by \( zb \). The loading of \( b \) and the drilling of \( z \) can occur in any order, but glue must be applied to \( z \) before \( b \) is placed. If the resource used for drilling is not high-precision, a reaming operation is performed. Finally a rivet is applied and \( zb \) is stored.

To establish the manufacturability of a product by a given system, we must establish mappings between the resource-independent BAT \( \delta_\text{T} \) and the BAT \( \delta_\text{S} \) of the system. In practice, these mappings are computed for each manufacturing system, automatically or by hand, at the moment of joining the manufacturing cloud (Felli et al. 2018). Inspired by (Banihashemi, De Giacomo, and Lespérance 2017):

- Each \( \mathcal{A}(x) \) in \( \delta_\text{S} \) is mapped to a (possibly complex) program \( \delta_\text{S}(x) \) in \( \delta_\text{T} \), e.g., passing of parts through the part-handling system, equipping effectors etc.;
- Some fluents \( f \in \mathcal{F}_\text{T} \) correspond to formulas over the fluents in \( \mathcal{F}_\text{S} \), i.e., to establish the value of \( f \) one needs to observe the situation of the underlying theory \( \mathcal{D}_\text{S} \) of \( \delta_\text{S} \). Hence we say that these fluents model “observations”, and use of a special unary predicate \( \text{Obs} \) to distinguish them.

This gives a set of mapping rules of the form:

\[
\mathcal{A}(x) \leftrightarrow \delta_\text{S}(x) \quad f(x) \leftrightarrow \varphi(x)
\]

For example, a rule that maps the resource-independent DRILL action in Figure 3 to a program specifying the possible ways in which a drilling operation can be executed in a specific system might be:

\[
\text{DRILL}(\text{part, drill, speed, feed, x, y, z}) \leftarrow (\mathcal{A}_1 \land \cdots \land \mathcal{A}_n) \quad ;
\]

- if size(part, large) then \( \varphi^0 \) \| \( \varphi^0 \) else \( \varphi^0 \)

\[
\varphi^0 = \text{\text{feed}} \quad i, j, k \quad \text{APPLY\_PRESS\_HOLLOW(p, i)} \] \| \[ \text{HOLD\_IN\_PLACE(p, x, j)} \] \| 

\[
\varphi^0 = \text{\text{feed}} \quad i, j \quad \text{OPERATE\_MACHINE(i, j)} \] \| 

\[
\text{MACHINE\_DRILL(p, bit, diam, speed, x, y, z, k)} \]

Crucially, a number of preliminary actions are required for these loading actions to be executable (e.g. equipping the right end effectors, clearing the working space, etc.) but these are not explicitly listed, as it is one of the objectives of the composition. Each \( \mathcal{A}_i \) stands for \( \pi x. \mathcal{A}_i(x) \mid \cdots \mid \mathcal{A}_n(x) \); each resource can perform any of their actions. We can write similar mapping rules for fluents, e.g., specifying how the precision of a drilled holes is observed.

In modelling a manufacturing domain, it is often natural to consider that the target and available programs are bounded (De Giacomo et al. 2016). In practice, this means
that the information of interest in each moment, corresponding to the parts that are being manufactured, the possible operations executed, their possible parameters and the data produced, are not arbitrarily large but are bounded by a known bound. E.g., resources have bounded capacity, a product recipe consists of finitely many parts and requires finitely many operations. This assumption will be used to give a decidable technique to synthesize controllers.

**Orchestration via Simulation**

To define the conditions under which a target program $\delta^0$ can be realised by executing the available program $\delta_\gamma$, we relate their execution. Extending the definition in (Sardinha and De Giacomo 2009) to our setting, we define the notion of *simulation* between programs:

$$\left(\delta_t, s_t\right) \subseteq (\delta_s, s_s) \supseteq \text{Final}(\delta_t, s_t) \supseteq \text{Final}(\delta_s, s_s) \land \text{Final}(\delta_t, s_t) \equiv \forall e. f(x, s_t) \equiv \forall e. f(x, s_s) \land \forall \delta_t', A, x. \text{TransObs}(\delta_t, s_t, s_s, \delta_t', \text{do}(A(x), s_t)) \supseteq \exists \delta_s', s_s'. \text{Trans}^\ast(\delta_s, s_s', \delta_t, s_t', \delta_s', s_s') \land \text{Do}(\delta_A(x), s_s, s_s') \land (\delta_t', s_t') \subseteq (\delta_s', s_s')$$

where $\text{TransObs}(\delta_t, s_t, s_s, \delta_t', s_s')$ iff $\text{Trans}(\delta_t, s_t, \delta_t', s_s')$ and $\text{Final}$ is substituted with $\text{FinalObs}$, defined as:

$$\text{FinalObs}(\delta_t, s_t, s_s) \equiv \text{Final}(\delta_t, s_t) \text{ if } \delta_t = \phi \text{ with } \text{Obs}(\phi) \text{ and } \text{Final}(\phi, s_t') \text{ otherwise}.$$  

Intuitively, we use the situation for theory $D_s$ for testing the situation suppressed fluents in the theory $D_t$ which correspond to observations of the underlying situations $s_s$. Moreover, $\text{Do}(\delta, s, s') := \exists \delta_s. \text{Trans}^\ast(\delta, s, \delta', s') \land \text{Final}(\delta', s')$ is used to establish that there exists a complete execution of the program $\delta$ from $s$ to $s'_s$.

The relation above specifies the following property: for every possible step from situation $s_s$, in which the target program evolves from $\delta_t$ to $\delta_t'$ by executing $A(x)$, there exists an execution of the concurrent program in situation $s_s$, from $\delta_s$ to $\delta_s'$ (and through a complete execution $\delta_A(x)$), for which the same property holds. Also, whenever the target program can terminate, also the available program can.

Through $\text{FinalObs}$, we allow the target program to assess the values of fluents on the situation $s_s$; the process recipes can specify conditions to be checked by observing the system. E.g., the value of precision in the program in Figure 4 must be observed after the drilling operation.

Moreover, note that we could not simply replace each action $A(x)$ in the target program $\delta^0$ by its corresponding program $\delta_A(x)$ and then apply known approaches as that of (Sardinha and De Giacomo 2009): to satisfy the simulation requirement it is enough to find at least one way in which $\delta_A$ can be executed so that the simulation property is maintained, whereas a syntactical substitution we would require that all such evolutions must be possible in the system.

Essentially, the simulation captures the fact that $\delta^0$ can implement the execution of $\delta_t'$, subject to the mapping rules.

**Definition 2 (Realizability).** The target program $\delta^0_t$ is realizable by the available program $\delta^0_\gamma$ if $(\delta^0_t, S^0_t) \subseteq (\delta^0_\gamma, S^0_\gamma)$.

When $\delta^0_t$ is realized by $\delta^0_\gamma$, then, at every step, given a possible ground action $A(x)$ selected by $\delta^0_\gamma$, one can select the corresponding program $\delta_A(x)$, execute it, and then return the control to the target program for the next action selection. Notice, however, that the execution of $\delta_A(x)$ is not deterministic, as ConGolog programs include in general choices of arguments and nondeterministic branching. Nonetheless, the existence of the simulation guarantees that this is possible, but it does not detail how. Similarly to (Sardinha and De Giacomo 2009), we assume to have total control on the interpreter executing the available concurrent program $\delta^0_\gamma$, whose nondeterminism is "angelic", and define here the notion of controller: the unit responsible for orchestrating the system, hence the available resources, at each step. Intuitively, this requires to consider any possible execution of $\delta^0_t$ and $\delta^0_\gamma$, as commented above, which are infinite.

First, in order to isolate the source of such infiniteness into the program data only, for each program (target and available) we separate the assignments of pick variables to objects in the domain from the control flow of the programs, namely their *program counter*. This is the approach of (De Giacomo et al. 2016), which we adapt here to our framework. Hence, we equivalently represent a program $\delta^0$ as the couple $(\delta, x)$, where $\delta$ merely denotes its current program counter, and $x = (x_1, \ldots, x_k)$ is a tuple of object terms so that each $x_i \in \Delta$ is the current value of $i$-th pick variable of $\delta^0$. We call $x$ the (current) environment. Importantly, this is merely a syntactic manipulation: as showed in (De Giacomo et al. 2016), we can reconstruct the original program $\delta^0$ by replacing the free pick variables of $\delta$ by those object terms to which variables $x$ are assigned. This is denoted by writing $\delta^0[x]$. Nonetheless, assuming programs without recursion, this simple technique allows one to obtain a finite set of possible program counters for a given program, which we define next (the possible environments remain infinite).

**Definition 3 (Syntactic closure of a program).** Given a program $\delta^0$, it is the set $\Gamma_{\delta^0}$ inductively defined as follows: (1) $\delta^0 \in \Gamma_{\delta^0}$; (2) if $\delta_1, \delta_2 \in \Gamma_{\delta^0}$ and $\delta_1 \sqsubseteq \delta_2$, then $\delta_1, \delta_2 \in \Gamma_{\delta^0}$ and $\Gamma_{\delta_2} \subseteq \Gamma_{\delta_1}$; (3) if $\delta_1, \delta_2 \in \Gamma_{\delta^0}$ then $\Gamma_{\delta_1, \delta_2} \subseteq \Gamma_{\delta^0}$; (4) if $\pi \alpha \delta \in \Gamma_{\delta^0}$ then $\Gamma_\delta \subseteq \Gamma_{\delta^0}$; (5) if $\delta' \in \Gamma_{\delta^0}$ then $\delta, \delta' \in \Gamma_{\delta^0}$; (6) if $\delta', \delta_2 \in \Gamma_{\delta^0}$ and $\delta'_1 \in \Gamma_{\delta_1}$, then $\Gamma_{\delta_1, \delta_2} \subseteq \Gamma_{\delta_0}$; (7) if $\delta_1 \sqsubseteq \delta_2 \in \Gamma_{\delta^0}$ and $\delta_1' \in \Gamma_{\delta_1}$, then $\delta_2' \in \Gamma_{\delta_0}$.

Denoting the finite set of all possible environments of a program $\delta^0$ as $\Delta^k$, so that $k$ is the number of its pick variables, we call a triple $(\delta, x, s) \in \Gamma_{\delta^0} \times \Delta^k \times S$ a (complete) configuration of $\delta^0$. Denoting the set of possible configurations as $C_{\delta^0}$, we can finally define our notion of controller, which, intuitively, given the current configurations $C_t$ and $C_\gamma$ for the target and system programs, and a new configuration for the target, selects a sequence of configurations for the system so that the simulation relation is recovered.

**Definition 4 (Controller).** Given a target program $\delta^0_t$ realizable by an available program $\delta^0_\gamma$, a controller for $\delta^0_t$ that realizes $\delta^0_t$ is a function $\rho : C^\gamma_{\delta^0} \times C^\gamma_{\delta^0} \times C^\gamma_{\delta^0} \rightarrow C^\gamma_{\delta^0}$ s.t.:

- $\rho(\delta_t, x_t, s_t), (\delta_s, x_s, s_s), (\delta'_t, x'_t, s'_t)$ is defined whenever $(\delta_t | x_t), s_t' \subseteq (\delta_s | x_s, s_s)$ and there exist $A, x$...
Controller Synthesis

To check whether a simulation exists and, if so, build a controller, we resort to model checking for a variant of the (modal) \( \mu \)-calculus in (Calvanese et al. 2013), interpreted over game arenas (GA), i.e., special (labelled) transitions systems (TS) capturing turn-based game rules. We show that when such systems are state-bounded, computing winning strategies becomes decidable.

Model checking over game arenas

For a set of fluents \( \mathcal{F} \) and an object domain \( \Delta \), we denote by \( \mathcal{I}_\Delta^{\mathcal{F}} \) the set of all (standard) FO interpretations \( \langle \Delta, \mathcal{I}(q) \rangle \) over \( \Delta \) of the fluents in \( \mathcal{F} \).

Definition 5 (Game arena). Let \( \mathcal{F} \) be a set of fluents including the special 0-ary fluents (i.e., propositions) \( \text{turn}S \) and \( turnT \), and \( \Delta \) an object domain. A game arena over \( \mathcal{F} \) and \( \Delta \) is a tuple \( \mathcal{T} = \langle \Delta, Q, q_0, \rightarrow, I \rangle \), where:

- \( \Delta \) is the object domain;
- \( Q \) 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;
- \( I : Q \rightarrow \mathcal{I}_\Delta^{\mathcal{F}} \) is a labeling function, associating to each state \( q \in Q \) an interpretation \( I(q) = \langle \Delta, \mathcal{I}(q) \rangle \) in \( \mathcal{I}_\Delta^{\mathcal{F}} \), s.t. exactly one among \( \text{turn}S \) and \( turnT \) is true.

\( \mathcal{T} \) represents the moves available to two players, Target and System, in a game, but not the game's arena. The arena is turn-based: Target and System can move in states where, respectively, \( turnT \) and \( turnS \) hold. Turns are not strictly alternating. Wlog, we assume that in \( q_0 \) it is Target's turn.

Goals are expressed through \( \mu \)-calculus formulas. The language we use, called \( \mu \mathcal{L}_c \) (c indicates that we use only closed FO formulas), is:

\[
\Phi ::= \phi \mid \neg \Phi \mid \Phi_1 \land \Phi_2 \mid \langle \Phi \rangle \mid Z \mid \mu Z. \Phi \mid \nu Z. \Phi
\]

were \( \phi \) is a FO sentence with predicates from \( \mathcal{F} \) and constants from \( \Delta \) (under unique-name assumption, we can safely use objects from \( \Delta \) as constants); the modal operator \( \langle \cdot \rangle \) denotes the existence of a transition from the current state to a state where \( \Phi \) holds; we use the abbreviation \( [\cdot] \Phi \) for \( \neg \langle \cdot \rangle \neg \Phi \); \( Z \) 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 \( \Phi \) seen as a predicate transformer with respect to \( Z \). By the language semantics below, one can see that the only interesting formulas are those that are closed wrt to SO (in addition to FO) variables. In fact, SO variables are needed only for technical reasons, to make the fixpoint constructs available.

Given a GA \( \mathcal{T} = \langle \Delta, Q, q_0, \rightarrow, I \rangle \), the semantics of a \( \mu \mathcal{L}_c \) formula \( \Phi \) over \( \mathcal{T} \) is inductively defined as follows, where \( v \) is an assignment to SO variables:

\[
\langle \Phi \rangle^\mathcal{T} = \{ q \mid q \in Q \text{ and } I(q) \models \psi \}
\]

\[
(\neg \Phi)^\mathcal{T} = Q \setminus (\Phi)^\mathcal{T}
\]

\[
(\Phi_1 \land \Phi_2)^{\mathcal{T}} = (\Phi_1)^{\mathcal{T}} \cap (\Phi_2)^{\mathcal{T}}
\]

\[
(\langle \Phi \rangle)^v = \{ q \mid \exists q', q \in \mathcal{T}, q' \in (\Phi)^v \}
\]

\[
(Z)^v = v(z)
\]

\[
(\mu Z. \Phi)^v = \bigcap \{ \mathcal{E} \subseteq Q \mid (\Phi)^T_{v[Z/\mathcal{E}]} \subseteq \mathcal{E} \}
\]

\[
(\nu Z. \Phi)^v = \bigcup \{ \mathcal{E} \subseteq Q \mid \mathcal{E} \subseteq (\Phi)^v_{Z/[\mathcal{E}]} \}
\]

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

\( \mu \mathcal{L}_c \) model checking is the problem of checking whether a GA \( \mathcal{T} \) satisfies a \( \mu \mathcal{L}_c \) formula \( \Phi \). When the GA is finite, this can be solved by direct application of the semantics. Thus one can compute the set \( \text{Win} \), called the winning set, of states of \( \mathcal{T} \) that satisfy the formula \( \Phi \). On the other hand, the problem can be shown to be undecidable in the general case (by reduction from the halting problem). In (Calvanese et al. 2013), decidability is proven under sufficient conditions, including genericity and state-boundedness. We recall these notions and relevant results.

Given two FO interpretations \( \mathcal{I} \) and \( \mathcal{I}' \) over a set of fluents \( \mathcal{F} \) and an object domain \( \Delta \), write \( \mathcal{I} \sim_h \mathcal{I}' \) if \( h : \Delta \rightarrow \Delta \) is an isomorphism between \( \mathcal{I} \) and \( \mathcal{I}' \), in which case \( \mathcal{I} \) and \( \mathcal{I}' \) are said to be isomorphic (under \( h \)). Intuitively, isomorphic states can be obtained by one another via object renaming.

Definition 6 (Genericity). A GA \( \mathcal{T} = \langle \Delta, Q, q_0, \rightarrow, I \rangle \) is said to be generic if: for every \( q_1, q_2 \in Q \) and every bijection \( h : \Delta \rightarrow \Delta \), if \( I(q_1) \sim_h I(q_2) \), then there exists \( q'_2 \in Q \) s.t. \( q_2 \rightarrow q'_2 \) and \( I(q'_2) \sim_h I(q_2) \).

In words, a GA is generic if states identical modulo object renaming show same behaviors, in particular, the same transitions (modulo renaming).

For an interpretation \( \mathcal{I} \), denote by \( \text{adm}(\mathcal{I}) \) the active domain of \( \mathcal{I} \), i.e., the set of all objects that occur in the interpretation of some fluent in \( \mathcal{I} \).

Definition 7 (State-boundedness). A GA \( \mathcal{T} = \langle \Delta, Q, q_0, \rightarrow, I \rangle \) is said to be state-bounded by \( b \in \mathbb{N} \) if \( \text{adm}(\mathcal{I}(q)) \leq b \), for every \( q \in Q \). \( \mathcal{T} \) is said to be state-bounded if it is state bounded by \( b \), for some \( b \).

Intuitively, a GA is state-bounded if, in every state, the number of objects occurring in the interpretation of some fluent is bounded by a given \( b \). Because \( \mu \mathcal{L}_c \) is a strict sublanguage of the general FO variant \( \mu \mathcal{L} \) defined in (Calvanese et al. 2013), by Theorem 17 therein, we have the following result.

Theorem 1. Given a generic and state-bounded GA \( \mathcal{T} \) and a \( \mu \mathcal{L}_c \) formula \( \Phi \), there exists a finite-state GA \( \mathcal{T}' \) such that \( \mathcal{T} \models \Phi \iff \mathcal{T}' \models \Phi \).
Thus, we can sidestep $T$’s infiniteness by checking whether $T’ \models \Phi$, instead of $T \models \Phi$. We do not describe how to obtain $T’$, referring the reader to (Calvanese et al. 2018), where a procedure is provided, which requires that, in $T$, (i) $\rightarrow$ is computable, and (ii) the existence of an isomorphism between states is decidable. The returned $T’$ is s.t. $T’ = \langle \Delta’, Q’, q_0, \rightarrow’, D’ \rangle$, with: $\Delta’, Q’$, and $\rightarrow’$ suitable finite subsets of their $T$ counterparts, and $D’$ the projection of $D$ over $Q’$, with $\Delta$ replaced by $\Delta’$. Notice that the $T$ and $T’$ share the same $F$.

$T$ and $T’$ are related by the notion of persistence-preserving bisimulation, $p$-bisimulation for short (Calvanese et al. 2018), i.e., a lifting of standard bisimulation to the case where states are labelled by FO (instead of propositional) interpretations. $p$-bisimulation 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 $h$ is an isomorphism between their interpretations, restricted to the active domains. In details, if $\langle q_1, h, q_2 \rangle$ is in a $p$-bisimulation $R$, then: (i) $q_1$ and $q_2$ have isomorphic fluent extensions, according to $h : \text{adom}(I_1(q_1)) \rightarrow \text{adom}(I_2(q_2))$ (objects not occurring in fluent extensions are neglected) –we denote this by writing $I_1(q_1) \sim_h I_2(q_2)$; (ii) for every successor $q’_1$ of $q_1$, there exists a successor $q’_2$ of $q_2$ and a bijection $b : \text{adom}(I_1(q’_1)) \cup \text{adom}(I_1(q_1)) \rightarrow \text{adom}(I_2(q’_2)) \cup \text{adom}(I_2(q_2))$ that extends $h$ to $\text{adom}(I_1(q’_1))$ s.t. for its restriction $h’$ to $\text{adom}(I_2(q’_2))$, $\langle q’_1, h’, q’_2 \rangle$ holds; (iii) the analogous of (ii) holds for every successor $q’_2$ of $q_2$.

$p$-bisimilarity intuitively means that 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. $T$ and $T’$ are $p$-bisimilar.

**Strategies**

In this paper, we consider only formulas of the form:

$$\Phi_{GA} = \nu X, \mu Y.(\text{turn}T \land \phi \land \lnot X) \lor (\text{turn}S \land \lnot Y)$$

where $\phi$ is a FO formula over $F \setminus \{\text{turn}S, \text{turn}T\}$. Intuitively, $\Phi_{GA}$ holds in all those states where: either (i) it is Target’s turn, (ii) $\phi$ holds, and (iii) no matter how Target moves, System can reply with a sequence of moves –which, by a slight abuse of notation, we call plan, that takes the GA to a new state where $\Phi_{GA}$ holds, or (iv) it is System’s turn and (v) System has a plan to reach a state where $\Phi_{GA}$ holds. Notice that if the initial state of the GA satisfies $\Phi$, then no matter how Target moves (now or in its future turns), System will always have a plan to enforce $\Phi_{GA}$.

Through model checking, we can obtain the winning set of $\Phi_{GA}$. To this end, the following operators are needed:

- $\text{PreE}(Z) = \{ q \in Q \mid \exists q’ \rightarrow q \text{ s.t. } q’ \in Z \}$;
- $\text{PreA}(Z) = \{ q \in Q \mid \forall q’ \rightarrow q \text{ then } q’ \in Z \}$.

With these, we compute the approximants for the SO variable $X$, ending up with a greatest fixpoint. The initial approximant of $X$ is $X_0 = Q$, and the next one is computed as $X_{i+1} = Y_i \cap X_i$, where $Y_i = \{ Y_i \text{. turn}T \land \phi \land \lnot X \}$,

$$\phi_{\text{SO}}(X_{i+1}) \in \Gamma(X_i \cup \lnot Y_i)$$

$Y_i$ being a (least) fixpoint, it can be computed, as standard, through successive approximants $Y_{i_0} = \emptyset, Y_{i_1}, \ldots, Y_{i_n} = Y_i$, as $Y_{i(j+1)} = Y_{i_j} \cup \Delta \cup (\text{turn}T \land \phi) \cup \text{PreE}(X_{i_j}) \cup (\text{turn}T \land \text{PreE}(Y_{i_j}))$.

The winning set $W$ is the resulting (greatest) fixpoint, i.e., $W = (\Phi_{\text{Sim}}) \in X_0$ (for some $k$).

If a state is in the winning set, System has a plan to reach a state where $\Phi_{GA}$ holds. However, we do not know such plan. We are interested not only in computing the winning states where $\Phi_{GA}$ holds but also in finding a “strategy” showing how the System can enforce $\Phi_{GA}$.

Let $T = (\Delta, Q, q_0, \rightarrow, I)$ be a GA. A history of $T$ is a sequence $\tau = q_0 \cdots q_\ell \in Q^+$ s.t., for every $i \in [0, \ell - 1]$, $q_i \rightarrow q_{i+1}$. We denote by $H$ the set of histories of a GA. A System (Target) strategy is a function $\varsigma : H \rightarrow Q$ s.t. if $\varsigma(q_0 \cdots q_{\ell}) = q$ then $q_{\ell} \models \text{turn}S(q_\ell \models \text{turn}T)$ and $q_\ell \rightarrow q$. In this paper, we are interested only in System’s strategies, i.e., functions that, given a history terminating in a state where System moves, prescribes a legal transition, the system’s move, to perform next. Since $q$ does not have to be s.t. $q \models \text{turn}T$, System can perform move sequences.

The strategies of interest are those, called winning, which enforce $\Phi_{GA}$; these are defined next. A history $\tau = q_0 \cdots q_\ell$ is said to be induced by a strategy if, for every $i \in [0, \ell - 1]$, whenever $q_i \models \text{turn}S$, $q_{i+1} = \varsigma(q_0 \cdots q_i)$.

**Definition 8 (Winning strategy).** A System strategy $\varsigma$ is said to be winning for a formula of the form $\Phi_{GA}$ if, for every history $\tau = q_0 \cdots q_\ell$ induced by $\varsigma$, either:

1. $q_\ell \models \text{turn}S$ and there exists a history $\tau’ = q_0 \cdots q_{\ell+1}$ induced by $\varsigma$ s.t. requirement\[2\] below (with $\ell$ replaced by $m$) holds; or
2. $q_\ell \models \text{turn}T \land \phi$ and for all histories $\tau’ = q_0 \cdots q_{\ell+1}$ requirement\[1\] above (with $\ell$ replaced by $\ell + 1$) holds.

Let $W_1$ be the winning set of $\Phi_{GA}$ computed on the finite-state GA $T’$ $p$-bisimilar to $T$. From this, we extract a winning strategy. Notice we cannot simply compute a strategy prescribing a path from a winning state to any other winning state, as in the presence of loops there would be no guarantee of eventually reaching the goal, i.e., a state where $\text{turn}T \land \phi$ holds.

To obtain a winning strategy, we observe that the computation of $W_1$ amounts to computing a series of $Y$’s approximants: $Y_{00}, \ldots, Y_{0n_0}, \ldots, Y_{00}, \ldots, Y_{0n_s}$, where each $Y_{ij}$ corresponds to the $j$-th approximant of the set of states where $\text{turn}S$ holds and from which a state in $X_i$ where $\text{turn}T \land \phi$ holds can be reached through System moves only. Thus, from $Y_{ij}$ one such state is reachable with $j - 1$ System moves. Since we are interested not in a generic $X$ approximant but in the winning set, we consider $Y_{00}, \ldots, Y_{0n_s}$ only, as these approximate the set of states that lead to states of the winning set that satisfy $\text{turn}T \land \phi$. Thus, we can “stratify” $W_1$ by annotating each of its states with the index $j$ of the first approximant $Y_{ij}$ it has appeared in. We denote the annotation of a state $q$ as $\text{annot}(q) = j$. In this way, to obtain a winning strategy for $\Phi_{GA}$, it is enough to choose a transition that takes the current state to one annotated with a lower value. Thus, a winning strategy is any
function $\varsigma$ s.t. $\varsigma(q_0 \ldots q_m) = q_{m+1}$ implies that $\text{ann}(q_m) > \text{ann}(q_{m+1})$. In fact, $\varsigma$ is memoryless, in that it does not depend on any state of the input history but the last one ($q_m$).

Next, we describe how one such strategy $\varsigma'$ computed on $\mathcal{T}'$ for a goal $\Phi_{GA}$ can be actually executed on $\mathcal{T}$.

**Definition 9 (p-bisimilar strategy transformation).** Consider two p-bisimilar GAs $\mathcal{T}$ and $\mathcal{T}'$ and let $\rho$ be a System strategy for $\mathcal{T}$. A strategy $\rho'$ for $\mathcal{T}$ is said to be a p-bisimilar transformation of $\rho$ to $\mathcal{T}'$, if there exists a p-bisimulation $R$ s.t. for every history $\tau = q_0 \cdots q_\ell$ of $\mathcal{T}$ induced by $\rho$, there exists a history $\tau' = q_0' \cdots q_{\ell}'$ of $\mathcal{T}'$ induced by $\rho'$ and a sequence of bijections $h_i : \text{dom}(I(q_i)) \to \text{dom}(I(q_i'))$ ($i = 0, \ldots, \ell$), s.t. for every $i \in [0, \ell]$, (i) $\langle q_i, h_i, q_i' \rangle \in R$ and (ii) 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 $b : \text{dom}(I(q_i)) \cup \text{dom}(I(q_i')) \to \text{dom}(I(q_i')) \cup \text{dom}(I(q_{i+1}'))$ s.t. $h_i = b|_{\text{dom}(I(q_i))}$ and $h_{i+1} = b|_{\text{dom}(I(q_{i+1}))}$. □

**Theorem 2.** If two GAs $\mathcal{T}$ and $\mathcal{T}'$ are p-bisimilar then there exists a System strategy $\varsigma$ on $\mathcal{T}$ iff there exists a System strategy $\varsigma'$ on $\mathcal{T}'$ that is a p-bisimilar transformation of $\varsigma$.

**Proof.** By p-bisimilarity, there exists a bisimulation $R$ s.t. for every history $\tau = q_0 \cdots q_\ell$ of $\mathcal{T}$ induced by $\varsigma$, there exists a history $\tau' = q_0' \cdots q_{\ell}'$ of $\mathcal{T}'$ that fulfills the requirement of $\varsigma'$ in Def. 9. For the if-part, we define $\varsigma'$ as $\varsigma'(q_0 \ldots q_{\ell-1}) = q'_\ell$, for every history $q_0 \cdots q_{\ell-1}$ of $\mathcal{T}$ induced by $\varsigma$, s.t. $q_{\ell-1} \models \text{turn}$. Only-if part is analogous. □

Theorem 2 provides us with a constructive way to transform a strategy executable on $\mathcal{T}$ into one on $\mathcal{T}'$. It essentially requires to transform the states of a history $\tau$ of $\mathcal{T}$ into those of a history $\tau'$ of $\mathcal{T}'$, by applying the isomorphisms that, in $\mathcal{R}$, associate the states of the two GAs, while preserving the identity of the objects that persist and of those that have just disappeared from the active domain.

For an example of this, assume to have computed a winning strategy on $\mathcal{T}$ and want to execute a transformation of it on $\mathcal{T}$. Assume that $\mathcal{T}$ has traversed the history $\tau = q_0 \cdots q_\ell$. From this, we obtain the corresponding $\mathcal{T}'$ history $\tau' = q_0' \cdots q_{\ell}'$, compute the move $q' = \varsigma'(\tau')$, and then translate it back to a move for $\mathcal{T}$, according to any isomorphism chosen as described in the theorem above.

**Controller synthesis**

In this section, we show how we can exploit $\mu\mathcal{L}_c$ model checking to both compute a simulation between the target program and the available program, and to synthesize the corresponding controller responsible for orchestrating the available system. Given the two programs $\delta_0^x$ and $\delta_0^\bar{x}$, together with the corresponding theories $\mathcal{D}_\bar{x}$ and $\mathcal{D}_x$, we now construct a GS $\mathcal{T}$ induced by these programs, i.e., that captures the crossproduct of their execution.

It is the GA $\mathcal{T} = \langle \Delta_{\mathcal{T}}, Q, q_0, \rightarrow, I \rangle$ built as follows:

**Object domain:** $\Delta_{\mathcal{T}} = \Delta_{\mathcal{T}}(T, S) \cup \Gamma_{\delta_0^x} \cup \Gamma_{\delta_0^\bar{x}} \cup \Gamma_{s}$, where the latter set is the union of the syntactic closures of all programs $\delta_A(x)$ such that a mapping rule $A(x) \leftrightarrow \delta_A(x)$ exists;

**States:** $Q \subseteq \{T, S\} \times C_{\delta_0^x} \times C_{\delta_0^\bar{x}} \times C_{\delta_s}$ is the set of states, where $C_{\delta_s}$ is a special set of configurations, discussed below.

Each $q = \langle \theta, (\delta_\bar{x}, \bar{x}_T, s_T), (\delta_x, x_T, s_T), (\delta_s, s_T, s_s) \rangle$ is such that $\theta \in \{T, S\}$ specifies the turn, i.e., which program, target (T) or available (S), moves next. $(\delta_\bar{x}, \bar{x}_T, s_T)$ and $(\delta_x, x_T, s_T)$ are, respectively, configurations of the target and available programs. Specifically, $\delta_s$ is the remaining fragment of the target program to execute, while $\delta_s$ is the program representing all remaining possible executions of the available system; moreover, $x_T$ and $\bar{x}_T$ are the current environments for these programs, representing the current assignments of their pick variables, and finally $s_T$ and $s_s$ are the situations of $\mathcal{D}_\bar{x}$ and $\mathcal{D}_x$ resulting, respectively, from the portion of $\delta_0^x$ and $\delta_0^\bar{x}$ executed so far. Finally, the additional configuration $(\delta_s, s_T, s_s) \in \Gamma_{s} \times \Delta_{\delta_s} \times S$ represents the remaining fragment of the program which corresponds, through a mapping rule, to the last target action $A(x)$ being realised by the system: after each turn of the target program, this is precisely $\delta_A(x)$. As defined in the definition of the transition relation, its role is to make sure that the evolution of the system, represented by the sequence of evolving configurations in $C_{\delta_t}$, corresponds to a complete execution of $\delta_A(x)$.

**Initial state:** $q_0 = \langle T, (\delta_0^x, x_0^T, S_0), (\delta_0^\bar{x}, x_0^\bar{T}, S_0), (\text{True?}, x_0^T, S_0) \rangle$. Initially, it is the turn of the target program; the target program is $\delta_0^x$ with initial assignment $x_0^T$; the available program is $\delta_0^\bar{x}$ with initial assignment $x_0^\bar{T}$; and the two BATs are in their initial situations. Also, there is yet no target action to be replicated, and therefore no associated program – True? is the empty program;

**Transitions:** $\rightarrow \subseteq Q \times Q$ is the transition relation, s. and $\rightarrow$ are defined through mutual induction: $q_0 \in Q$ and if $q \in Q$ we have that $q' \in Q$ for all $q \rightarrow q'$. A transition $\langle \theta, (\delta_\bar{x}, \bar{x}_T, s_T), (\delta_x, x_T, s_T), (\delta_s, s_T, s_s) \rangle \rightarrow \langle \theta', (\delta_\bar{x}, \bar{x}_T', s_T'), (\delta_x, x_T', s_T'), (\delta_s, s_T', s_s') \rangle$ exists if and only if it is the turn of the target program and a possible next target situation $s_T' = \text{do}(A(x), s_T)$ is selected (that is, resulting from the execution of the action $A(x)$), or it is the turn of the available concurrent program and the target situation $s_T'$ can be replicated by executing $\delta_A(x)$ in the system. We also need to make sure that the resulting situation $s_T'$ is a situation in which the program $\delta_A$ corresponding to A is final (for this, we use Do). Since $\delta_A(x)$ is single-step, we use $\vartheta$ to establish a strict alternation between $\delta_0^\bar{x}$ and $\delta_0^x$. Therefore,

$\langle \theta, (\delta_\bar{x}, \bar{x}_T, s_T), (\delta_x, x_T, s_T), (\delta_s, s_T, s_s) \rangle \rightarrow \langle \theta', (\delta_\bar{x}', \bar{x}_T', s_T' ), (\delta_x', x_T', s_T'), (\delta_s', s_T', s_s') \rangle$ if either:

- $\vartheta = \text{TransObs}(\{\delta_\bar{x}[x_T], s_T, s_s, \delta_\bar{x}'[x_T'], s_T' \} \land \exists A.x. s_T' = \text{do}(A(x), s_T) \land \delta_\bar{x}' = \delta_\bar{x} \land s_T = s_T'$

- $\vartheta = \text{Trans}(\{\delta_\bar{x}[x_T], s_T, s_s, \delta_\bar{x}'[x_T'], s_T' \} \land \text{Trans}(\{\delta_\bar{x}[x_T], s_T, s_s, \delta_\bar{x}'[x_T'], s_T' \}) \land \text{Trans}(\{\delta_\bar{x}, s_T, s_s, \delta_\bar{x}' \} \land \delta_\bar{x}' = \delta_\bar{x} \land s_T' = s_T \land (\text{Final}(\delta_\bar{x}[x_T], s_T, s_s, \delta_\bar{x}' ) \land \vartheta = T \lor \vartheta = S )$.

The former case applies when it is the turn of the target program and a possible next target situation $s_T' = \text{do}(A(x), s_T)$ is selected (corresponding to an action $A(x)$). The target configuration is progressed, while the system remains idle. The last configuration registers the program to execute, with free variables replaced by $x$ (denoted here by...
iff 

The latter case applies when it is the turn of the available program, while the system remains idle. Note that we also progress the program corresponding to the last configuration \(c_b\), testing that the program remaining fragment of \(\delta_x(x)\) may terminate, and in this case we allow the turn to be "given back" to the target program.

**Labelling:** \(I : Q \rightarrow I_{\delta, \Delta}\), where \(F = F_T \cup F_S \cup \{\text{turn, progT, progS, finalT, finalS, envT, envS}\}\), for fluents in \(F_T\) and \(F_S\) with the situation argument suppressed. Formally, we "make visible" the internal state of the system through the labelling \(I\), so that we can evaluate \(\mu L_c\) formulas on \(T\). Formally, \(I\) is as follows:

(i) Fluents in \(F_T\) and \(F_S\) are interpreted according to the interpretation provided by the model \(M\) of \(D_T \cup D_S\) and the Situation Calculus and ConGolog axioms at situations \(s_T\) and \(s_s\). Specifically, for every \(q \in Q\) and \(x \in \Delta\), and for every \(f \in F_S\) we have \(f_T(q)(x)\) iff \(f^M(x, s_s)\). Similarly, for each fluent \(f \in F_T\),

- if \(\vartheta = s\) then \(f_T(q)(x)\) iff \(f^M(x, s_T)\);
- if \(\vartheta = T\) then:
  - if \(-\text{Obs}(f)\) then \(f_T(q)(x)\) iff \(f^M(x, s_T)\);
  - otherwise \(f_T(q)(x)\) iff \(\phi^M(x, s_s)\).

Intuitively, we define the labelling function \(I\) based on the model \(M\), making sure that, whenever \(\vartheta = T\), \(I\) is consistent with the mappings between \(D_T\) and \(D_S\) for all fluents representing observations of the available system.

(ii) For the remaining fluents in \(F\), assuming \(q = \langle \vartheta, (\delta_T, x_T, s_T), (\delta_S, x_s, s_s) \rangle\), we have: \(\text{turn}_T(q) = \emptyset\), \(\text{prog}_T(q) = \delta_T\), \(\text{prog}_S(q) = \delta_S\), \(\text{env}_T(q) = \{x_s\}\), and \(\text{env}_S(q) = \{x_s\}\). Finally, the 0-ary predicate \(\text{finalT}\) is true iff \(\text{final}(\delta_T[x_T], s_T)\), and analogously for \(\text{finalS}\).

The \(GA\) above is essentially the tree of executable (combinations of) configurations for \(\delta_T^0\) and \(\delta_S^0\), with the state-labelling providing an interpretation of the set of fluents \(F\), used to verify \(\mu L_c\) formulas on \(T\). Notice that labelings retain all the relevant information about states.

Satisfaction of the following \(\mu L_c\) by \(T\) implies the existence of a simulation between \(\delta_T^0\) and \(\delta_S^0\), with the state-labelling providing an interpretation of the set of fluents \(F\), used to verify \(\mu L_c\) formulas on \(T\). Notice that labelings retain all the relevant information about states.

\[\Phi_{Sim} = \nu X. \mu Y. ((\phi_{OK} \land [x]X) \lor (\text{turnS} \land [x]Y))\]

where \(\phi_{OK} = (\text{finalT} \rightarrow \text{finalS}) \land \text{turnT}\). Intuitively, \(\phi_{OK}\) holds in those states in which it is the turn of the target program and if the target program may terminate so can the available program. Therefore, the formula requires that no matter how the target program evolves to a new program \(\delta_T^\tau\) through the execution of an action \(\text{A}(x)\) from a state in which \(\phi_{OK}\) holds, from that successor state (where \(\text{turnS}\) holds) there exists a sequence of transitions corresponding to a complete execution of \(\delta_x(x)\), and from where the whole property still holds.

**Theorem 3.** Let \(\text{Win}(\Phi_{Sim})\) be the set of winning states in \(T\) wrt \(\Phi_{Sim}\). Then, \(\delta_T^\tau\) is realizable by \(\delta_S^\vartheta\) iff \(q_0 \in \text{Win}(\Phi_{Sim})\).

**Proof.** (Sketch.) It follows from the fact that for each state \((T, (\delta_T, x_T, s_T), (\delta_S, x_s, s_s), c_b) \in \text{Win}(\Phi_{Sim})\) we have \((\delta_T[x_T], s_T) \subseteq (\delta_S[x_s], s_s)\). It is immediate to see that if a state \(q \in \text{Win}(\Phi_{Sim})\) then \(q \models \phi_{OK}\) and that all fluent mappings are satisfied by definition of \(I\) in \(T\). Also, since any possible target action \(A(x)\) is captured by a successor of such \(q\), it follows that there exists a path in \(T\) corresponding to \(\delta_S(x)\), and that the resulting state is in \(\text{Win}\). With analogous reasoning, one can see that opposite holds as well, as it implies \((\delta_T[x_T], s_T) \subseteq (\delta_S[x_s], s_s)\). □

Although the fixpoint computation described earlier gives us a way of capturing the winning set \(\text{Win}(\Phi_{Sim})\), the number of approximants that we need to compute is bounded (by the size of the \(GA\)) only if \(T\) is finite. Since, in our case, \(T\) can be infinite, the fixpoint cannot be computed, in general.

**Lemma 1.** \(T\) as above is generic.

**Proof.** (Sketch) The result is a direct consequence of the fact that \(T\) is defined on two \(BA\)s and the transition relation is defined by a \(FO\) specification involving only the current state and the next one. □

Thus, by Theorem 1 if \(T\) is state-bounded, there exists a finite \(GA\) \(T'\) which we can use to verify \(\text{Win}(\Phi_{Sim})\) (instead of using the infinite-state \(T\)). Importantly, \(T'\) is effectively computable.

**Lemma 2.** If \(T\) is state-bounded, then \(T'\) is effectively computable.

**Proof.** (Sketch) Consequence of Theorem 17 of Calvanese et al. [2018] and the fact that, for a given state, if \(T\) is state-bounded, the successor state is computable. □

By applying our general technique for \(\Phi_{GA}\) formulas, we can compute a winning strategy from \(\text{Win}(\Phi_{Sim})\). While this strategy is not directly executable on \(T\), we can exploit the notion of strategy transformation introduced earlier. Therefore, a concrete inductive procedure for executing the controller corresponding to the strategy for \(T\) is provided in the proof of Theorem 2 which we can directly execute.

Finally, as a winning strategy \(\varsigma\) as above is given, we can directly obtain the corresponding controller for \(\delta_T^0\) and \(\delta_S^0\), as explained in Definition 4 as follows. For every history \(\tau = q_0 \cdots q_{m+1}\), with \(q_{m+1} = \text{turnT}\) (and thus \(q_{i+1} = \text{turnS}\)), for \(q_i = \langle \vartheta_i, c_{iT}, c_{iT}, c_{iS} \rangle\), we return the sequence \(\rho((c_{iT}, c_{iT}, c_{iT+1})) = c_{iS} \cdots c_{i0}\) with \(c_i\) is the system configuration of each state \(q_i = \varsigma(\tau q^{0} \cdots q^{i})\), for \(i \in [0, m-1]\).

**Conclusions**

In this paper, by exploiting recent results on the Situation Calculus, we have been able to effectively synthesize controllers for manufacturing-as-a-service scenarios, under the assumption of state boundedness. However, we have only scratched the surface of what KR formalisms like the Situation Calculus can bring to this new manufacturing paradigm. For instance, it would be of interest to equip resources with autonomous deliberation capabilities (Baldwin 1989), e.g., to react to exogenous events during execution, or to monitor streaming production data (Lee et al. 2015), or consider the explicit treatment of time and other continuous value quantities (Behandish, Nelaturi, and de Kleer 2018). We leave these to further work.
References

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

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

[Behandish, Nelaturi, and de Kleer 2018] Behandish, M.; Nelaturi, S.; and de Kleer, J. 2018. Automated process planning for hybrid manufacturing. Computer-Aided Design. (in press).

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

[De Giacomo et al. 2016] De Giacomo, G.; Lespérance, Y.; Patrizi, F.; and Sardiña, S. 2016. 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., 950–956.

[De Giacomo et al. 2018] De Giacomo, G.; Vardi, M.; Felli, P.; Alechina, N.; and Logan, B. 2018. Synthesis of orchestrations of transducers for manufacturing. In Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI-18). New Orleans, USA: AAAI.

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

[De Giacomo et al. 2013] De Giacomo, G.; Patrizi, F.; and Sardiña, S. 2013. Automatic behavior composition synthesis. Artificial Intelligence 196:106–142.

[de Silva et al. 2016] de Silva, L.; Felli, P.; Chaplin, J. C.; Logan, B.; Sanderson, D.; and Ratchev, S. 2016. Realisability of production recipes. In Kaminka, G. A.; Fox, M.; Bouquet, P.; E., H.; F. D.; V.; D.; and van Harmelen F., eds., Proceedings of the 22nd European Conference on Artificial Intelligence (ECAI-2016), 1449–1457. The Hague, The Netherlands: ECCAI.

[Felli et al. 2017] Felli, P.; de Silva, L.; Logan, B.; and Ratchev, S. 2017. Process plan controllers for non-deterministic manufacturing systems. In Sierra, C., ed., Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI 2017), 1023–1030. Melbourne, Australia: IJCAI.

[Felli et al. 2018] Felli, P.; de Silva, L.; Logan, B.; and Ratchev, S. 2018. Composite capabilities for cloud manufacturing (Demonstration). In Dastani, M.; Sukthankar, G.; Andre, E.; and Koenig, S., eds., Proceedings of the 17th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2018), Stockholm, Sweden: IFAAMAS. (to appear).

[Felli, Logan, and Sardina 2016] Felli, P.; Logan, B.; and Sardina, S. 2016. Parallel behavior composition for manufacturing. In Kambhampati, S., ed., Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI 2016), 271–278. IJCAI/AAAI Press.

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

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

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

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

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

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

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

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

[Technology Strategy Board 2012] Technology Strategy Board. 2012. A landscape for the future of high value manufacturing in the UK. Technical report, Technology Strategy Board.

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