Optimal Manufacturing Controller Synthesis Using Situation Calculus

Omar Adalat, Daniele Scrimieri(✉), and Savas Konur

Department of Computer Science, University of Bradford, Bradford, UK
{o.adalat,d.scrimieri,s.konur}@bradford.ac.uk

Abstract. In this paper, we discuss a framework for synthesising manufacturing process controllers using situation calculus, a well-known second-order logic for reasoning about actions in AI. Using a library of high-level ConGolog programs and logical action theories for production resources, we demonstrate how to efficiently synthesise an 'optimal' plan, i.e. the plan that minimises the number of actions for a target high-level program of a process recipe.

Keywords: Situation calculus · Controller synthesis · ConGolog · Planning · Manufacturing

1 Introduction

Traditional planning formalisms and languages are often difficult to work with for automated synthesis within the manufacturing domain in a practical sense. While many other formalisms have been proposed and developed for manufacturing controller synthesis [1,9], many of these formalisms only allow expressing and representing states propositionally instead of a data-aware first-order state representation of objects [6] and their intertwined relationships. As a direct consequence almost all approaches in prior literature yield lower expressive power. The approach presented in this paper uses situation calculus and builds upon [6,12]. Briefly, we consider the problem of synthesising a high-level controller that orchestrates a number of manufacturing resources composed of logical action theories and high-level programs. Such a controller implements a 'recipe', i.e. a manufacturing process specification for making a product, modelled as a high-level program. We provide an easily controllable progression-based search algorithm instead of relying on computationally intensive fixpoint calculations.

2 Preliminaries

2.1 Situation Calculus

Situation calculus is a many-sorted second-order logic with equality for reasoning about actions to represent dynamically changing worlds [11]. The three disjoint

M. Bramer and F. Stahl (Eds.): SGAI 2023, LNAI 14381, pp. 222–227, 2023.
https://doi.org/10.1007/978-3-031-47994-6_19

sorts are: *situations*, *actions*, and *objects* (Δ). Situations are terms derived by action histories with S_0 representing the initial situation constant.

Fluents are dynamic properties that can change across situations, where relational fluents operate on objects and return a proposition of $\{\top, \bot\}$, such as $\mathit{equipped}(\mathit{effector}, s)$. Functional fluents instead return against a range of objects. We also use the standard closed-world and domain-closure assumptions.

Additionally, we track two situation-independent static predicates, specifically for the manufacturing domain, following from [6]. The first is $\mathit{routable}(i, j)$, stating that there exists a valid route between resource i with resource j (usable with In and Out actions that model transferring parts). The second is $\mathit{coopMatrix}(i, j)$, stating that resource i can cooperate with resource j.

Actions are n-arity terms. The binary function $\mathit{do}(a, s)$ signifies that action a is performing in situation s, such that $\mathsf{do} : \mathsf{action} \times \mathsf{situation} \rightarrow \mathsf{situation}$. To determine whether an action a is possible or executable within a situation s, we use the following binary predicate $\mathsf{Poss} : \mathsf{action} \times \mathsf{situation}$, which is determined by a situation-suppressed formula of the form $\mathsf{Poss}(\mathsf{a}(\boldsymbol{x}), s) \equiv \varphi(\mathsf{a}(\boldsymbol{x}), s)$ where φ is a situation-suppressed formula with $\boldsymbol{x}$ denoting the vector of arguments. *Compound actions* (denoted by $\mathbf{a}$) [6] refer to joint or simultaneous actions, such as $\{\mathsf{LiftAndHold}(\mathsf{p}), \mathsf{Drill}(\mathsf{p})\}$.

Successor state axioms encode the causal laws of the system. A successor state axiom is of the form $F(\boldsymbol{x}, do(a, s)) \equiv \varphi_F(\boldsymbol{x}, a, s)$, one for each fluent. The update of a fluent is given by $F(\boldsymbol{x}, do(a, s)) \equiv \varphi_F^+(\boldsymbol{x}, a, s) \vee F(\boldsymbol{x}, s) \wedge \neg\varphi_F^-(\boldsymbol{x}, a, s)$ where F is a fluent, $\varphi_F^+(\boldsymbol{x}, a, s)$ and $\varphi_F^-(\boldsymbol{x}, a, s)$ are formulae that make F true or false respectively.

Each manufacturing production resource that we model comes with its own *basic action theory* (BAT), which is a set of sentences $\mathcal{D}$, formed from

$$\mathcal{D} = \mathcal{D}_{\mathsf{S}_0} \cup \mathcal{D}_{ap} \cup \mathcal{D}_{ss} \cup \mathcal{D}_{una} \cup \Sigma$$

where $\mathcal{D}_{\mathsf{S}_0}$ initial situation description, Σ is the set of domain-independent foundational axioms [11], $\mathcal{D}_{ss}$ is the set of successor state axioms, $\mathcal{D}_{ap}$ represents the set of action precondition axioms, and $\mathcal{D}_{una}$ provides the unique name axioms for actions. $\phi[s]$ denotes substituting situation term s in the situation-uniform formula ϕ.

2.2 ConGolog

ConGolog is a high-level programming language for situation calculus that supports a rich notion of concurrency and non-determinism [8]. ConGolog uses transition-step semantics, with two predicates, $\mathit{Trans}(\delta, s, \delta', s')$ denoting that a program transition exists from δ to δ' starting from situation s and resulting in situation s', and $\mathit{Final}(\delta, s)$ denoting whether a program configuration is final in situation s. The constructs available within a ConGolog program δ are given by:

$$\delta ::= \mathbf{a}(\boldsymbol{x}) \mid \delta_1; \delta_2 \mid \phi? \mid (\delta_1 \mid \delta_2) \mid \pi x.\delta \mid \delta^* \mid (\delta_1 \parallel \delta_2) \mid (\delta_1 \mid\mid\mid \delta_2)$$

The full semantics is presented in [8], with ||| being introduced solely to represent simultaneous/joint execution (true concurrency) between two programs [6]. We use the standard abbreviations **if** ϕ **then** δ_1 **else** δ_2 **endIf** $\stackrel{def}{=} \phi?;\delta_1|\neg\phi?;\delta_2$, **while** ϕ **do** δ **endWhile** $\stackrel{\text{def}}{=} (\phi?;\delta)^*;\neg\phi?$ and **loop :** $\delta \stackrel{def}{=}$ **while** $\top$ **do** δ.

Putting this together with the situation calculus, each manufacturing production resource is thereby modelled by a pair of $\langle\delta, \mathcal{D}\rangle$. An example resource production program δ is given by:

$$\textbf{loop:}\ \mathsf{Nop}|(\mathsf{In}(\mathit{part}, 2);(\mathsf{Nop}|\mathsf{Hold}(\mathit{part}, \mathit{force}, 2)))^*;\mathsf{Out}(\mathit{part}, 2)$$

This program states that it can either hold in no-operation, Nop, or it can take a part in, afterward performing either no-operation or holding it in place with a given force, possibly indefinitely, and then afterward it must transfer the part back out.

2.3 Characteristic Graphs

Characteristic graphs [5] allow a succinct representation of all subprogram configurations of a program δ. A characteristic graph $\mathcal{G}$ consists of the triple $\langle V, E, v_0\rangle$, defined as follows:

- The set of vertices V correspond to reachable subprograms, formed of a pair $\langle\delta', \varphi\rangle$, where δ' is the remaining executable program and φ provides the conditions under which the current execution is final.
- v_0 is the symbol for the distinguished initial program vertex, $v_0 = \langle\delta^0, \varphi^0\rangle$.
- Edges in E are labelled with tuples $\pi\boldsymbol{x}\colon t/\psi$, stating a transition can occur for action t with a choice (π) of instantiations for the variable list $\boldsymbol{x}$ (omitted if not required), as long as the condition specified in formula ψ holds.
- Edges in E, restricted to recipes, can have actions that are compound.

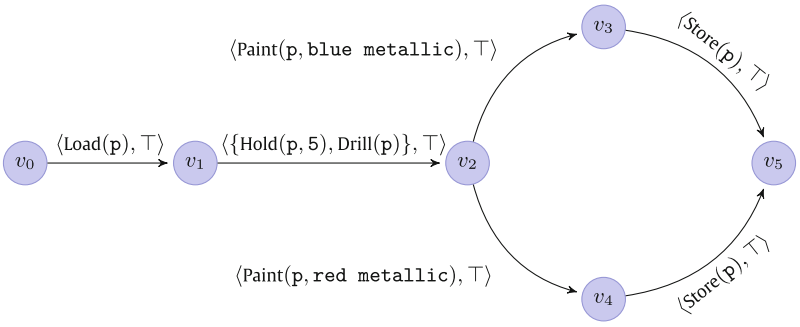

Fig. 1. Characteristic graph for a process recipe G_σ. For brevity we omit vertex labels, however, note that v_5 is a final configuration $\langle nil, \top\rangle$.

An example characteristic graph of a recipe is represented in Fig. 1. This corresponds to: $\mathsf{Load}(\mathtt{p});(\mathsf{Hold}(\mathtt{p}, 5)|||\mathsf{Drill}(\mathtt{p}));(\mathsf{Paint}(\mathtt{p}, \mathtt{metallic\ blue})|\mathsf{Paint}(\mathtt{p}, \mathtt{metallic\ red}));\mathsf{Store}(\mathtt{p})$

3 Controller Synthesis

A *controller* is composed of executions of compound actions, partially ordered, that realise all recipe transitions of a target ConGolog recipe program (note that, unlike resources, the recipe does not have a basic action theory $\mathcal{D}$). We assume recipes are terminating processes, allowing us to consider final configurations and not infinite runs. Also, observe that, since the action Nop (no-operation) for any resource i has no effect, it is given that $\mathsf{Poss}(\mathbf{a} \cup \mathsf{Nop}, s) \equiv \mathsf{Poss}(\mathbf{a}, s)$.

Algorithm 1: Synthesise

Data: Recipe characteristic graph $\mathcal{G}_\sigma$, resources' characteristic graphs $\mathcal{G}^n_{i=1}$, $\mathcal{D}_{global}$, S^F_0, plan length limit N, fairness limit F, object domain Δ
Result: Controller χ if realisable
$\chi_{initial}.\mathcal{Q}$.push(AddStages($\mathsf{S}^F_0$, v_0 of $\mathcal{G}_\sigma$, F));
$\chi_{best}.length \leftarrow \infty$; $frontier$.push(χ_{initial});
while $\neg empty(frontier) \wedge \chi_{best}.length \geq frontier.top().length$ **do**
  $\chi_c, \mathcal{P}_c \leftarrow frontier$.pop(), $\chi_c.\mathcal{Q}$.pop();
  if *compound actions in* $\chi_c \geq N$ **then**
    continue;
  forall *compound actions* **ca** *of trace* $\in$ *GatherTransitions*($\mathcal{G}^n_{i=1}$, Δ) **do**
    if *Poss*(**ca** , $\mathcal{P}_c.S$) $= \bot \vee \bigwedge^n_{i=1}$ *transition* $\psi_i[\mathcal{P}_c.S] = \bot$ **then**
      continue;
    Form χ', $\mathcal{P}'$ with Do(**ca**, $\mathcal{P}_c.S$) and *trace*;
    $\chi'.length \leftarrow 1 +$ (1 * simple actions (non-Nop) of **ca**)
    if **ca** *from recipe transition* $\mathcal{P}_c$ *is executed* **then**
      AddStages($\mathcal{P}'.\mathsf{S}$, $\mathcal{P}'$.to, F);
      if $\mathcal{P}_c$*.to's transitions in* $\mathcal{G}_\sigma = \emptyset \wedge \neg \bigwedge^n_{i=1} \mathcal{G}'_i s$ *Final* $\varphi[\mathcal{P}'.S] = \bot$ **then**
        continue;
      if *empty*($\mathcal{X}'.\mathcal{Q}$) **then**
        Potentially **update** χ_{best};
        continue;
      else
        Enqueue χ' into $frontier$;
    else
      Enqueue $\mathcal{P}'$ into $\chi'.\mathcal{Q}$;
      Enqueue χ' into $frontier$;
  return either χ_{best} or **unrealisable** if $\chi_{best}.length = \infty$

Controller synthesis resorts to a two-player game between the proponent *controller*, and the antagonist *environment*, due to the non-determinism possibly involved in recipes that allow runtime choices such as $\pi x.\delta$. That is, no matter what 'devillish' non-determinism move the recipe program δ_σ makes, the controller can respond, where the resources' ConGolog programs operate under 'angelic' non-determinism (that is, non-determinism we can control) [7,12].

We start by defining $\mathcal{D}_{global} \stackrel{def}{=} \bigcup_{i=1}^{n} \mathcal{D}_i$ which is the result of combining the individual basic action theories. The initial facility situation constant is then represented by S_0^F. While the state space involved may appear to be finite, given a finite number of subprogram configurations, finite number of actions types, and finite object domain Δ, this is not the case. In particular, the construct δ^* is considered 'unsafe' [3,10] since it allows infinite computations, and by proxy, the same extends to higher-level constructs **while** and **loop**. The way we deal with this is through a *fairness assumption*, i.e. placing a limit on the number of folds a δ^* execution can reoccur.

Proposition 1. *There is a finite number of transitions that can be considered under a fairness assumption as each explored cycle is only allowed k unfoldings, or a finite number of considerations by imposing a plan upper limit of N at any stage (per recipe transition or globally)*

Reasoning about preconditions (Poss) formulas for compound actions typically decomposes into $\bigwedge_{n=1}^{i} \mathsf{Poss}(\mathbf{a}_n, s)$. However, we also allow special mappings for certain actions, such as for the In and Out actions:

$$\mathsf{Poss}(\mathsf{In}(part, i), s) \equiv part(part, s) \wedge \neg\exists\, p.\ at(p, i, s)$$
$$\mathsf{Poss}(\mathsf{Out}(part, i, s)) \equiv part(part, s) \wedge at(part, i, s)$$
$$\mathsf{Poss}(\{\mathsf{In}(part, i), \mathsf{Out}(part, j)\}, s) \equiv \mathsf{Poss}(\mathsf{In}(part, i), s) \wedge \mathsf{Poss}(\mathsf{Out}(part, j), s) \wedge routable(i, j)$$
$$withinReach(part, i, s) \equiv at(part, i, s) \vee (\exists\, j.\ j \neq i \wedge at(part, j, s) \wedge coopMatrix(i, j)$$

Successor state axioms are aware of compound actions and can reason about containment for simple actions within compound actions, for example:

$$equipped(e, i, \mathsf{do}(\mathbf{a}, s)) \equiv \mathsf{Equip}(e, i) \in \mathbf{a} \vee (equipped(e, i, s) \wedge \mathsf{Unequip}(i) \notin \mathbf{a})$$

Algorithm 1 outlines the synthesis strategy we consider. The priority queue *frontier* contains the controller candidates that have been generated sorted by number of actions, where the candidate with the lowest number of actions is at the top of the queue. The inner queue of each controller, $\mathcal{Q}$, considers the recipe transitions/'phases' that are yet to be completed from (AddStages, which holds responsibility for the fairness limit). Function GatherTransitions operates on the object domain and resource characteristic graphs. It picks all potential transitions, keeping in mind the current characteristic graph states $\mathcal{G}_{i=1}^{n}$, and action permutations with variable substitution from Δ. The *cost* (*length* in the algorithm) ascribed to each controller can be viewed as the sum of the number of all simple actions (non-Nop) and the number of compound actions.

One possible controller execution (as controllers involve runtime execution choices), once flattened out, would appear like this, assuming three arbitrary resources including our previously defined example resource and example recipe:

$$\{\mathsf{Load}(\mathtt{p}), \mathsf{Nop}, \mathsf{Nop}\} \rightarrow \{\mathsf{Out}(\mathtt{p}, 1), \mathsf{In}(\mathtt{p}, 2), \mathsf{Nop}\} \rightarrow$$
$$\rightarrow \{\mathsf{Hold}(\mathtt{p}, 5), \mathsf{Paint}(\mathtt{p}, \mathtt{blue\ metallic}), \mathsf{Nop}\} \rightarrow \{\mathsf{Nop}, \mathsf{Out}(\mathtt{p}, 2), \mathsf{In}(\mathtt{p}, 3)\} \rightarrow$$
$$\rightarrow \{\mathsf{Nop}, \mathsf{Nop}, \mathsf{Store}(\mathtt{p})\}$$

In this execution, part p is first loaded by the first resource, then transferred to the second resource, which holds it and paints it in blue metallic. Finally, the part is transferred to the third resource and stored.

4 Conclusion

In summary, we present an *optimal progression search* approach for synthesising manufacturing controllers using the situation calculus. The next step is to carry out experimental evaluation using our prototype (https://github.com/nightly/scs). In the future, our tool could be easily extended to provide an explicit treatment of time, further exploration of heuristics and abstraction techniques, non-Markovian actions and effects, exogenous events, as well as noisy [2] and fallible [4] sensing.

Acknowledgements. This work was supported by the SURE Research Projects Fund and Research Development Fund of the University of Bradford and Innovate UK grant no: 10028947 (Made Smarter Innovation: Sustainable Smart Factory).

References

1. Adalat, O., Talal, M., Ali Cherif, M.A., Scrimieri, D.: Model-based generation of manufacturing process plans through incremental topology formation. In: Panoutsos, G., Mahfouf, M., Mihaylova, L. (eds.) Advances in Computational Intelligence Systems - Contributions Presented at the 21st UK Workshop on Computational Intelligence, Sheffield, UK, 7–9 September 2022. Advances in Intelligent Systems and Computing. Springer, Cham (2023, in press)
2. Bacchus, F., Halpern, J.Y., Levesque, H.J.: Reasoning about noisy sensors and effectors in the situation calculus. Artif. Intell. **111**(1–2), 171–208 (1999)
3. Beck, D., Lakemeyer, G.: Reinforcement learning for Golog programs with first-order state-abstraction. Log. J. IGPL **20**(5), 909–942 (2012)
4. Claßen, J., Delgrande, J.P.: An account of intensional and extensional actions, and its application to belief, nondeterministic actions and fallible sensors. In: Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning, vol. 18, pp. 194–204 (2021)
5. Claßen, J., Lakemeyer, G.: A logic for non-terminating Golog programs. In: KR, pp. 589–599 (2008)
6. De Giacomo, G., Felli, P., Logan, B., Patrizi, F., Sardina, S.: Situation calculus for controller synthesis in manufacturing systems with first-order state representation. Artif. Intell. **302**, 103598 (2022)
7. De Giacomo, G., Lespérance, Y.: The nondeterministic situation calculus. In: Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning, vol. 18, pp. 216–226 (2021)
8. De Giacomo, G., Lespérance, Y., Levesque, H.J.: ConGolog, a concurrent programming language based on the situation calculus. Artif. Intell. **121**(1–2), 109–169 (2000)
9. De Giacomo, G., Vardi, M., Felli, P., Alechina, N., Logan, B.: Synthesis of orchestrations of transducers for manufacturing. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 32 (2018)
10. Kalantari, L., Ternovska, E.: A model checker for verifying ConGolog programs. In: AAAI/IAAI, pp. 953–954 (2002)
11. Reiter, R.: Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems. MIT Press, Cambridge (2001)
12. Sardina, S., De Giacomo, G.: Composition of ConGolog programs. In: Twenty-First International Joint Conference on Artificial Intelligence (2009)