A Clean and Efficient Implementation of Choreography Synthesis for Behavioural Contracts

Davide Basile(✉) and Maurice H. ter Beek

Formal Methods and Tools Lab, ISTI–CNR, Pisa, Italy
{davide.basile,maurice.terbeek}@isti.cnr.it

Abstract. The Contract Automata Tool is an open-source tool for the specification, composition and synthesis of coordination of service contracts, including functionalities to deal with modalities and configurations. We discuss an implementation of the abstract parametric synthesis algorithm firstly introduced in our COORDINATION 2019 paper, comprehending most permissive controller, orchestration and choreography synthesis. The tool's source code has been redesigned and refactored in Java 8, and we show the resulting gain in computational efficiency.

Keywords: Service Computing · Contract Automata · Controller Synthesis · Orchestration · Choreography

1 Introduction

Orchestration and choreography are two coordination policies for service composition [14,16,40]. The specifications of services can be provided as behavioural contracts [5] that expose the interface to other services and are used to compute contract-based coordination policies.

Contract automata [10] formalise behavioural service contracts in terms of service offer actions and service request actions that need to match to achieve agreement among a composition of contracts. Modalities are used to indicate when an action must be matched (necessary) and when it can be withdrawn (permitted) in the synthesised coordination [6]. Composing contracts and synthesising a coordination, by refining a spurious composition, are the two main operations supporting contracts. Synthesis builds upon results from supervisory control theory [17,26,41] for synthesising the most permissive controller (mpc for short), duly revisited for synthesising orchestrations and choreographies in [9]. We are aware of only one other approach to coordinating services by supervisory control theory [2]. Contract automata have been equipped with a proof-of-concept tool [7] to show the feasibility of the proposed theoretical approach.

Motivation. According to a recent survey on formal methods among high-profile experts [23], the debate between "leaving the development of professional

Published by Springer Nature Switzerland AG 2021
F. Damiani and O. Dardha (Eds.): COORDINATION 2021, LNCS 12717, pp. 225–238, 2021.
https://doi.org/10.1007/978-3-030-78142-2_14

tools to industry" and "academia should invest effort to develop and consolidate usable tools" has no clear winner. In support of the latter, there is a shared belief that "academia should invest effort to develop and consolidate usable tools" because "this is the only way to provide technological transfer to industry, [as] in most cases efficient implementation requires to know a bit about the underlying theory". Indeed, according to [21,24], tool deficiencies (e.g., ease of use) are rated as one of the top obstacles for the adoption of formal methods by industry. However, to achieve this industrial transfer, in [30] it is recommended that "universities need to find ways to incentives industrial collaboration by adjusting its system of academic and career credits" and "research support and funding agencies need to actively encourage tool development and maintenance beyond prototyping" and "develop flexible funding schemes (in-cash or in-kind) to support the engineering work that is necessary to transform a prototype implementation into a demonstrable implementation". In support of the former, quite some academics believe that "we should not spend time on polishing the things that matter for acceptance in industry, such as user interfaces, have round-the-clock available help desks, liability, etc.", because "there is no business case for long term support of (academic) tools; industry needs stability and performance, academics need to innovate". In fact, it is evident that few research groups manage to work for decades on one tool and have it applied successfully in industry. We note, however, a gap between going beyond prototyping (e.g., by providing well-designed, clean and efficient implementations) and going as far as providing industry-ready tools (e.g., with help desks, industrial certification and floating licenses). Indeed, as also reported in [23], "the tools we develop should be usable (and extensible) by researchers in our own community" and "effort should be devoted to open-source community efforts". To this aim, we present a clean and efficient implementation of theoretical results presented in [9], providing an open-source tool [18] beyond the prototypical level, which can be reused as an API (and extended) by other researchers and developers in service coordination, rather than an off-the-shelf tool, ready to be adopted in industry.

Contribution. In this paper, we discuss improvements in the design and implementation of the contract automata tool [18]. It has been redesigned according to the principles of model-based systems engineering (MBSE for short) [29,42] and those of writing clean and readable code [15,36], which are known to improve reliability and understandability and facilitate maintainability and reuse. The tool has moreover been refactored using lambda expressions and Java Streams as available in Java 8 [25,43], exploiting parallelism. We are not aware of any other synthesis algorithm that uses big data-like parallel operations as Java Streams. The implementation of the abstract parametric synthesis algorithm from [9] and the mpc, orchestration and choreography synthesis are presented. We recompute the contracts of the case study in [9] to demonstrate the gain in computational efficiency of the new implementation, and we briefly address the gain in code readability and maintainability.

Outline. The paper is organised as follows. In Sect. 2, we briefly discuss the tool's new design. In Sect. 3, we recall theoretical results on the abstract synthesis algorithm. In Sect. 4, we present in detail the refactored synthesis implementation and discuss its adherence to the specification. In Sect. 5, we evaluate the improvement, both in absolute terms and in performance gain. In Sect. 6, we present related work whilst Sect. 7 concludes the paper.

2 Design

The tool's architecture has been redesigned with the MBSE tool Sparx Enterprise Architect[1] (EA for short) and Eclipse. EA allows to import Eclipse projects to generate documentation and UML diagrams. The UML class diagram concerning the main package of the tool, displayed in Fig. 1, has been generated by EA. For readability, only fields that are relevant for this paper are visible for each class.

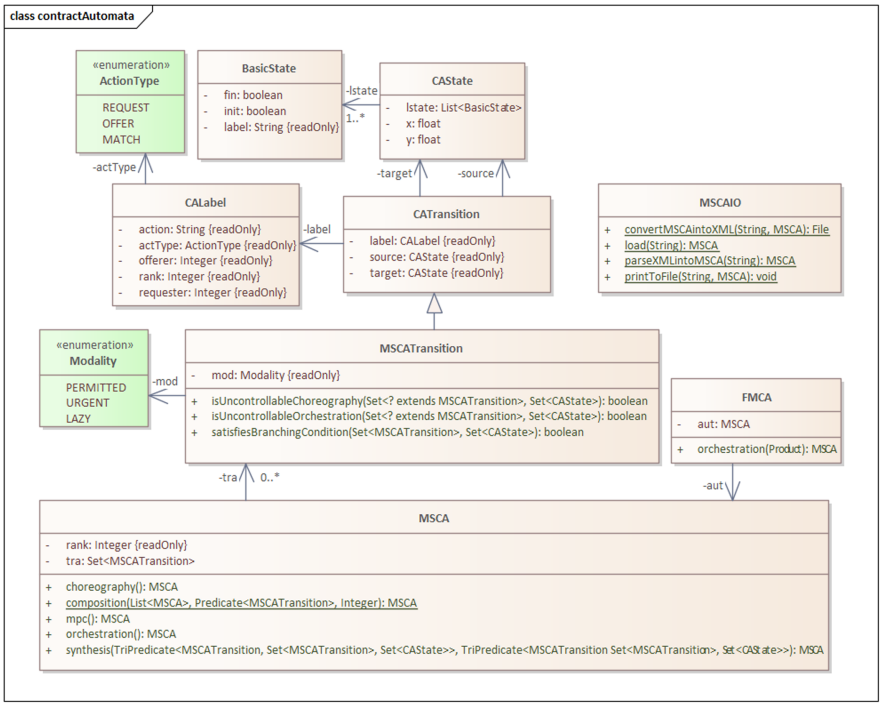

Fig. 1. The class diagram of the contract automata tool [18]

The standard UML class diagram is self-explanatory. The input/output functionalities are grouped in a stand-alone class MSCAIO, used by the application.

[1] https://sparxsystems.com/products/ea/.

The core of the implementation resides in the class MSCA that contains methods for composing and synthesising contracts, discussed below. The decorator pattern is used for the class FMCA, which adds the functionality of synthesising an orchestration for a specific configuration (called a product, cf. [6]).

Another package of the tool, `family`, concerns the aforementioned functionalities, discussed in [6], regarding the possibility of synthesising an orchestration of a product line (also called family) of service contracts, where each configuration is in a (partial) ordering relation with other configurations. The functionalities of this package have not been refactored in Java 8 yet, and do not concern the contribution discussed in this paper. The repository is available at [18].

The GUI Application. One of the advantages of adopting a widely used language such as Java is the availability of many resources. In particular, we implemented a GUI application, publicly available in [19]. This application is importing and using both our CAT library discussed in this paper and the `mxGraph` library [37] that provides features for displaying interactive diagrams and graphs. We specialised the `GraphEditor` example of the library to develop the GUI of the tool. We wish to emphasise the separation of concerns between the tool's usability for end users, addressed by the GUI [19], and the usability of the API offered to other developers, addressed in this paper and available in [18]. Developers can use our library as back-end to other software, and efficiency and clean design of the implementation are our primary concern. Nevertheless, being able to graphically visualise the computed contracts has been helpful for experimenting new developments in the theory of contract automata.

3 Specification

In this section, we recall the specification of the abstract synthesis algorithm from [9] that will be useful to provide some evidence that the implementation in Sect. 4 adheres with the specification. This is a fix-point computation where at each iteration the set of transitions of the automaton is refined (pruning predicate ϕ_p) and a set of forbidden states R is computed (forbidden predicate ϕ_f). The synthesis is parametric on these two predicates, which provide information on when a transition has to be pruned from the synthesised automaton or a state has to be deemed forbidden. The syntheses of mpc, orchestration and choreography are obtained by instantiating these two predicates. We refer to MSCA as the set of (modal service) contract automata, where the set of states is denoted by Q and the set of transitions by T (with $T^{\Box}$ denoting the set of necessary transitions). For an automaton $\mathcal{A}$, the predicate $Dangling(\mathcal{A})$ contains those states that are not reachable from the initial state or that cannot reach any final state.

Definition 1 (Abstract synthesis [9]). *Let $\mathcal{A}$ be an MSCA, and let $\mathcal{K}_0 = \mathcal{A}$ and $R_0 = Dangling(\mathcal{K}_0)$. Given two predicates $\phi_p, \phi_f : T \times MSCA \times Q \to \mathsf{Bool}$, we let the* abstract synthesis function *$f_{(\phi_p,\phi_f)} : MSCA \times 2^Q \to MSCA \times 2^Q$ be defined as follows:*

$$f_{(\phi_p,\phi_f)}(\mathcal{K}_{i-1}, R_{i-1}) = (\mathcal{K}_i, R_i), with$$
$$T_{\mathcal{K}_i} = T_{\mathcal{K}_{i-1}} \setminus \{\, t \in T_{\mathcal{K}_{i-1}} \mid \phi_p(t, \mathcal{K}_{i-1}, R_{i-1}) = true \,\}$$
$$R_i = R_{i-1} \cup \{\, \boldsymbol{q} \mid (\boldsymbol{q} \rightarrow) = t \in T_{\mathcal{A}}^{\square},\ \phi_f(t, \mathcal{K}_{i-1}, R_{i-1}) = true \,\} \cup Dangling(\mathcal{K}_i)$$

The abstract controller is defined in Eq. 1 below as the least fixed point (cf. Theorem 5.2 [9]) where, if the initial state belongs to $R_s^{(\phi_p,\phi_f)}$, then the controller is empty, otherwise it is the automaton with the set of transitions $T_{\mathcal{K}_s^{(\phi_p,\phi_f)}}$ and without states in $R_s^{(\phi_p,\phi_f)}$.

$$(\mathcal{K}_s^{(\phi_p,\phi_f)}, R_s^{(\phi_p,\phi_f)}) = sup(\{\, f^n_{(\phi_p,\phi_f)}(\mathcal{K}_0, R_0) \mid n \in \mathbb{N} \,\}) \tag{1}$$

4 Implementation

The implementation has been refactored in Java 8, the latest major feature release[2] including lambda expressions and streaming API. Streams are used for big data-style processing of data structures, incorporating MapReduce-like operations [20]. Streams can be easily parallelised, abstracting from the underlying realisation with parallel threads. Although a parallel stream can be obtained with a simple method (`parallelStream()`), if not carefully used issues may be encountered, e.g., race conditions. Indeed, the usage of Java 8 Streams is currently under investigation [31–33]. Based on the analysis of 34 Java projects, two important findings listed in [33] are: "Stream parallelization is not widely used", and "Although streams feature performance improving parallelism, developers tend to struggle with using streams efficiently". This seems to confirm the finding of [35], "indicating the difficulty of reasoning [on] concurrent execution by programmers", while in [31,32], focusing on evaluating refactoring, it is noted that "using streams efficiently requires many subtle considerations".

In our implementation, parallel streams are carefully used to speed-up the computation of the set of transitions and forbidden states at each iteration. We both provide an informal argument on the correctness of our implementation below, confirmed by testing our implementation on the case studies in [9] and [6], as well as experimental evidence on the efficiency of the new implementation in Sect. 5. We start by discussing the parametric synthesis method below.

Lines 2–3 show the two parameters of the method. Both predicates take three arguments: the transition under scrutiny, the set of transitions and the set of forbidden states computed so far. Lines 5–6 are used to store references to the transitions, states and initial state, which could be lost during the synthesis. Initially, the set of forbidden states `R` is composed of dangling states (line 7). A Boolean flag `update` is used to flag when the least fixed point is reached. At each iteration (lines 9–20), the set of transitions is refined with a parallel stream filtering away those transitions satisfying the pruning predicate (lines 11–13).

[2] https://www.oracle.com/java/technologies/java8.html.

Similarly, the set `R` is updated by adding dangling states due to pruned transitions (line 14). Source states of transitions satisfying the forbidden predicate are computed using a parallel stream (considering also transitions previously pruned) and added to `R` (lines 15–18). Finally, when the fixed point is reached the dangling transitions are removed (line 21), and if the initial state is not forbidden (line 25) the synthesised MSCA is returned (line 26).

```
1  public MSCA synthesis(
2   TriPredicate<MSCATransition, Set<MSCATransition>, Set<CAState>> pruningPred,
3   TriPredicate<MSCATransition, Set<MSCATransition>, Set<CAState>> forbiddenPred)
4   {
5     Set<MSCATransition> trbackup = new HashSet<MSCATransition>(this.getTransition());
6     Set<CAState> statesbackup= this.getStates();  CAState init = this.getInitial();
7     Set<CAState> R = new HashSet<CAState>(this.getDanglingStates(statesbackup,init)); //R0
8     boolean update=false;
9     do{  final Set<CAState> Rf = new HashSet<CAState>(R);
10         final Set<MSCATransition> trf= new HashSet<MSCATransition>(this.getTransition())
11         if (this.getTransition().removeAll(this.getTransition().parallelStream()
12                         .filter(x->pruningPred.test(x,trf, Rf))
13                         .collect(Collectors.toSet()))) //Ki
14                 R.addAll(this.getDanglingStates(statesbackup,init));
15         R.addAll(trbackup.parallelStream()
16                         .filter(x->forbiddenPred.test(x,trf, Rf))
17                         .map(MSCATransition::getSource)
18                         .collect(Collectors.toSet())); //Ri
19         update=Rf.size()!=R.size()|| trf.size()!=this.getTransition().size();
20    } while(update);
21    this.removeDanglingTransitions();
22    if (R.contains(init)) return null;
23    return this;
24  }
```

Correctness. We provide an informal argument on the adherence of the implementation with respect to the specification provided in Sect. 3. Thanks to the high-level constructs provided by Java, this is quite straightforward since the distance between the implementation and the specification is narrow.

As already stated, the fix-point computation is implemented as a simple `do while` loop, where the Boolean variable `update` (line 19) is used to check that the computed sets have not been modified by the last iteration. This is done by simply checking that their size has not changed. Indeed, only instructions for removing transitions or adding states to R are invoked at each iteration. The functional interface `TriPredicate` is used to type both pruning and forbidden predicates as functions taking three arguments and returning a Boolean. The set R_0 of Definition 1 is computed by the instruction in line 7, using the method `getDanglingStates` that implements the predicate *Dangling*. This method basically performs a forward and backward visit of the automaton. Here the correspondence with Definition 1 is obtained by simply observing that $\mathcal{K}_0 = \mathcal{A}$. Indeed, the predicate *Dangling* of Definition 1 takes as arguments the automaton to which the dangling states are computed. In the implementation, such automaton is the object `this` to which the method `getDanglingStates` is invoked (basically, $\mathcal{K}_i$ in Definition 1 is the object `this`).

The instructions in lines 11–12 perform the set difference on the set of transitions, by removing the transitions satisfying the pruning predicate. This is basically the same as Definition 1. As already stated, the computation of the set

of forbidden states R starts in line 14 by adding the dangling states. In case no transition has been removed, the set of dangling states is unchanged.

Sources of transitions satisfying the forbidden predicate are added in lines 15–18. Here there is a slight divergence from Definition 1: the abstract synthesis algorithm only checks the forbidden predicate on necessary transitions. Since the notion of necessary transition varies depending on whether we are synthesising an mpc, an orchestration or a choreography, in the implementation this check will be implemented by the forbidden predicate passed as argument to the `synthesis` method (see below). Due to the well-known state-explosion problem, it is expected that, for an average composition, the set of transitions is not of a small size. Thus, parallel streams are used to efficiently process each element separately and independently from the other threads, with no concurrency issues.

Instructions in lines 21–22 finalise the automaton to be returned as discussed in Sect. 3.

Mpc Synthesis. Concerning the synthesis of the mpc below, the property of agreement is enforced, i.e., no request shall be left unmatched. Thus, a state is forbidden if it has an outgoing uncontrollable request. Indeed, in the mpc synthesis, necessary requests are uncontrollable, according to the standard notion of uncontrollability in supervisory control theory, called *urgent* in contract automata.

```
1  public MSCA mpc(){
2   return synthesis((x,t,bad) -> bad.contains(x.getTarget()) || x.getLabel().isRequest(),
3                    (x,t,bad) -> !t.contains(x) && x.isUrgent());
4  }
```

The synthesis of the mpc is obtained by instantiating the pruning and forbidden predicates. The pruning predicate checks if a transition has a forbidden target state or is a request (line 2). The forbidden predicate selects source states of necessary transitions (i.e., urgent) that have been previously removed (line 3).

Orchestration Synthesis. The synthesis of the orchestration below is similar to the one of the mpc, apart from the different notion of necessary request.

```
1  public MSCA orchestration(){
2    return synthesis((x,t,bad) -> bad.contains(x.getTarget())|| x.getLabel().isRequest(),
3          (x,t,bad) ->  x.isUncontrollableOrchestration(t, bad));
4  }
```

In the orchestration, necessary requests are semi-controllable (line 3): basically, a necessary request becomes uncontrollable if there exists no execution in which that request is matched, otherwise it is controllable. Intuitively, in an orchestration of service contracts the order (among possible interleavings) in which the necessary requests are matched does not matter, as long as there exists at least one execution in which the match takes place. The orchestrator is in charge of driving the services towards executions in agreement [9].

Choreography Synthesis. The orchestration assumes the presence of an implicit orchestrator driving the executions toward safe behaviour. In a choreography, instead, contract automata are supposed to be able to interact safely on their own, without resorting to a central orchestrator. To do so, the property to be enforced is called *strong agreement*, i.e., all requests and offers have to be matched (this property is also referred to as absence of orphan messages). A property called *branching condition* must hold: automata must be able to send offers independently from the state of other automata. In the choreographic framework, requests are always permitted, whereas offers can also be necessary. Necessary offers use semi-controllability for choreographies, which is weaker than uncontrollability yet stronger than the semi-controllable notion used in the synthesis of orchestration. Indeed, compared to orchestrations, an additional constraint must hold: the transitions matching the necessary offer must share the same source state. This is because the automata must be able to interact correctly by only using local information.

```
1  public MSCA choreography(){
2    MSCA aut; MSCATransition toRemove= null;
3    Set<String> violatingBC = new HashSet<>();
4    do {
5    aut=this.clone().synthesis((x,t,bad)->!x.getLabel().isMatch()||bad.contains(x.getTarget())
6             ||violatingBC.contains(x),(x,t,bad) -> x.isUncontrollableChoreography(t, bad));
7    if (aut==null) break;
8    final Set<MSCATransition> trf = aut.getTransition();
9    toRemove=(aut.getTransition().parallelStream()
10            .filter(x->!x.satisfiesBranchingCondition(trf, new HashSet<CAState>()))
11            .findAny().orElse(null));
12   } while (violatingBC.add(toRemove));
13   return aut;
14 }
```

The choreography synthesis algorithm iteratively calls the synthesis method using semi-controllability for choreographies in the forbidden predicate and strong agreement in the pruning predicate (lines 5–6). After reaching the fixed point, a transition that violates the branching condition is non-deterministically selected (lines 9–11). Depending on which transition is selected and removed, different choreographies can be obtained. The synthesis abstracts away from the way in which transitions violating the branching condition are selected. Notably, removing only one such transition at each synthesis invocation allows to remove a smaller number of transitions than, for example, removing all of them at the first iteration. The iteration continues until there are no transitions violating the branching condition. An alternative implementation could contain only one call to the synthesis method, similarly to the orchestration and mpc methods. It would suffice to only use the instructions in lines 6–7 by moving the branching condition check in line 10 inside the pruning predicate in line 5. In this way, transitions violating the branching condition would be pruned at each step of the called synthesis algorithm, and this method would compute a smaller, possibly empty choreography. The specification did not fix any strategy for selecting which transition violating the branching condition should be pruned and when. This indeed could be decided according to different criteria.

Table 1. Improvement in computational runtime (ms) of the current tool version [18]. All experiments run on a machine with Processor Intel(R) Core(TM) i7-8500Y CPU at 1.50 GHz, 1601 Mhz, 2 Core(s), 4 Logical Processor(s), 16 GB RAM, 64-bit Windows 10.

	composition			orchestration			choreography		
	runtime in [9]	current runtime	runtime speedup	runtime in [9]	current runtime	runtime speedup	runtime in [9]	current runtime	runtime speedup
$\mathcal{A}_1$	65594	1312	49.99x	715216	2872	249.03x	–	–	–
$\mathcal{A}_2$	66243	1006	65.85x	–	–	–	459311	1604	286.35x

5 Evaluation

A rough measure of the improvement for what concerns code readability and maintainability can be obtained by comparing the lines of source code (LOC for short) with those used in the previous prototypical implementation [22]. The previous choreography synthesis used 211 LOC, while the current implementation uses only 14 (choreography) + 24 (synthesis) LOC. Similarly, the previous orchestration synthesis used 178 LOC, which have been refactored in 2 (orchestration) + 24 (synthesis) LOC. The synthesis method is factorised for orchestration, choreography and mpc, which was not possible before, so reducing code duplication. Finally, UML diagrams (cf. Fig. 1) provide the benefits of graphical documentation of the architecture of the tool that was not previously available.

To evaluate the gain in efficiency of the current implementation [18], we compare its performance with that of the previous implementation [22], which was programmed using quick incremental patches over the years and without parallelism. In [9], the previous implementation was applied to a case study, with the performances reported in Examples 2.5, 3.4 and 4.6 and recalled in Table 1. Table 1 also reports the data of applying the current implementation and the speedup, showcasing the improvement obtained thanks to redesigning the tool, cleaning the code and refactoring the algorithms by using parallel streams.

6 Related Work

The literature offers several approaches to the problem of synthesising a choreography of interacting components, with supporting tools. Recently, a tool chain for choreographic design has been proposed in [28]. Choreographies are designed using a kind of BPMN2 Choreography Diagrams, but equipped with a formal semantics. The operation of closure at the semantic level is used to insert missing behaviour, which can be suggested as amendments to be validated by a human. Our approach is completely automatised. Our choreography synthesis is based on the synthesis of the mpc that *refines* the composition by removing rather than adding behaviour. The composition can be computed automatically, starting from local components, to represent all their possible interactions independently from the selected communication pattern (e.g., non-blocking output, blocking output). The composition could also be the starting point, in this case

representing a global choreography to be realised, if possible. Our algorithm is non-deterministic and accounts for necessary and permitted actions.

In [3], the synthesis of so-called coordination delegates is discussed. Coordination delegates are used to enforce the behaviour prescribed by a choreography designed as a BPMN2 Choreography Diagram, and are additional components that interact with each other and with the services identified by the choreography to enforce the prescribed behaviour. A mature tool for software development according to this proposal is presented in [4]. Similarly, in [38], BPMN2 Choreography Diagrams are used to automatically derive a conformant choreography-based software architecture. Informal diagrams are mapped to coloured Petri nets, specifying the coordination logic. Coordination delegates are synthesised to fulfill the inferred coordination logic that is needed to enforce the realisability of the choreography. They communicate with the participants they are supervising or among themselves. These delegates are synthesised using the approach described in [12] to either relax or restrict the original behaviour. This is obtained by introducing extra communications performed by the delegates.

These approaches to the synthesis of coordination delegates are similar to the synthesis of distributed controllers in [34], which studies the fundamental problem of supervisory control synthesis for local controllers interacting among them through a coordinator.

Compared to the above work, our approach to choreography synthesis does not introduce any intermediate component, nor additional behaviour. This is not the case for our orchestration synthesis that assumes the presence of an orchestrator dictating the overall execution by interacting with local components. Since contract automata are *composable* and our synthesised choreography is again a contract automaton, we conjecture that similar results could be obtained by (i) partitioning the composition, (ii) separately synthesising a choreography for each partition, and (iii) computing the orchestration of the composition of the choreographies. The conditions under which such partitions can be computed to obtain non-empty choreographies need to be investigated, perhaps exploiting existing research on requirements splitting for supervisory control synthesis [27].

A static analyser for Go programs that uses a global session graph synthesis tool to detect communication deadlocks is discussed in [39]. From Go programs, communicating finite state machines (CFSM for short) are extracted, and used to synthesise a global choreography that represents deadlock-freeness in the original program. Our synthesised choreography has also been interpreted in the framework of CFSM [11], and the composition of contract automata enables to proceed bottom-up by composing local components into a choreography. Notably, if such choreography cannot be realised by composing local components, our algorithm automatically detects which portion of behaviour is to be pruned, if possible, to synthesise a deadlock-free choreography. We conjecture that this result could be used to suggest amendments to the original Go program.

The authors of this paper have gained experience in applying other tools for controller synthesis, viz., CIF 3 [13] and Uppaal Stratego [8]. It would be

interesting to investigate an encoding of the coordination syntheses discussed in this paper in the tools discussed in this section, to draw a comparison.

Since our implementation is cleanly designed, efficient and implemented in a widely used language in few lines of code, we hope that it will be exploited in other tools as a further option to the synthesis problem for orchestration, choreographies and most permissive controllers.

7 Conclusion

We have presented recent improvements in the contract automata tool. The source code has been redesigned using MBSE techniques, and refactored in Java 8 exploiting parallel streams, including the novel choreography synthesis. The correspondence between the formal specification and the implementation is discussed. The obtained improvements are emphasised by comparisons with the previous tool version.

Future Work. In the future, we would like to formally prove that the implementation respects its specification using, e.g., the theorem prover KeY [1] that adopts specifications written in Java Modeling Language. While this is the state-of-the-art for Java, there is no support for Java Streams and lambda expressions to date, leaving this as a long-term goal. Also, the current version of our tool supports product lines of orchestrations but not of choreographies, which we plan to investigate, together with real-time support. Our approach refines a spurious composition to a choreography, in the style of controller synthesis. As discussed in Sect. 6, other approaches propose to add additional behaviour during synthesis. A full-fledged comparison with other approaches is a matter of future investigation. It is also interesting to exploit the compositionality offered by contract automata to combine choreography and orchestration synthesis with the goal of maximising the set of contracts that can correctly interact whilst minimising the overhead of the orchestration.

Acknowledgments. We acknowledge funding from the MIUR PRIN 2017FTXR7S project IT MaTTerS (Methods and Tools for Trustworthy Smart Systems).

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book: From Theory to Practice. LNCS, vol. 10001. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49812-6
2. Atampore, F., Dingel, J., Rudie, K.: Automated service composition via supervisory control theory. In: Proceedings of the 13th International Workshop on Discrete Event Systems (WODES 2016), pp. 28–35. IEEE (2016). https://doi.org/10.1109/WODES.2016.7497822

3. Autili, M., Inverardi, P., Perucci, A., Tivoli, M.: Synthesis of distributed and adaptable coordinators to enable choreography evolution. In: de Lemos, R., Garlan, D., Ghezzi, C., Giese, H. (eds.) Software Engineering for Self-Adaptive Systems III. Assurances. LNCS, vol. 9640, pp. 282–306. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-74183-3_10
4. Autili, M., Salle, A.D., Gallo, F., Pompilio, C., Tivoli, M.: CHOReVOLUTION: service choreography in practice. Sci. Comput. Program. **197** (2020). https://doi.org/10.1016/j.scico.2020.102498
5. Bartoletti, M., Cimoli, T., Zunino, R.: Compliance in behavioural contracts: a brief survey. In: Bodei, C., Ferrari, G.-L., Priami, C. (eds.) Programming Languages with Applications to Biology and Security. LNCS, vol. 9465, pp. 103–121. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25527-9_9
6. Basile, D., et al.: Controller synthesis of service contracts with variability. Sci. Comput. Program. **187** (2020). https://doi.org/10.1016/j.scico.2019.102344
7. Basile, D., ter Beek, M.H., Gnesi, S.: Modelling and analysis with featured modal contract automata. In: Proceedings of the 22nd International Systems and Software Product Line Conference (SPLC 2018), vol. 2, pp. 11–16. ACM (2018). https://doi.org/10.1145/3236405.3236408
8. Basile, D., ter Beek, M.H., Legay, A.: Strategy synthesis for autonomous driving in a moving block railway system with Uppaal Stratego. In: Gotsman, A., Sokolova, A. (eds.) FORTE 2020. LNCS, vol. 12136, pp. 3–21. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-50086-3_1
9. Basile, D., ter Beek, M.H., Pugliese, R.: Synthesis of orchestrations and choreographies: bridging the gap between supervisory control and coordination of services. Log. Methods Comput. Sci. **16**(2) (2020). https://doi.org/10.23638/LMCS-16(2:9)2020
10. Basile, D., Degano, P., Ferrari, G.L.: Automata for specifying and orchestrating service contracts. Log. Methods Comput. Sci. **12**(4:6), 1–51 (2016). https://doi.org/10.2168/LMCS-12(4:6)2016
11. Basile, D., Degano, P., Ferrari, G.L., Tuosto, E.: Relating two automata-based models of orchestration and choreography. J. Log. Algebr. Meth. Program. **85**(3), 425–446 (2016). https://doi.org/10.1016/j.jlamp.2015.09.011
12. Basu, S., Bultan, T.: Automated choreography repair. In: Stevens, P., Wąsowski, A. (eds.) FASE 2016. LNCS, vol. 9633, pp. 13–30. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49665-7_2
13. ter Beek, M.H., Reniers, M.A., de Vink, E.P.: Supervisory controller synthesis for product lines using CIF 3. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016, Part I. LNCS, vol. 9952, pp. 856–873. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47166-2_59
14. ter Beek, M.H., Bucchiarone, A., Gnesi, S.: Web service composition approaches: from industrial standards to formal methods. In: Proceedings of the 2nd International Conference on Internet and Web Applications and Services (ICIW 2007). IEEE (2007). https://doi.org/10.1109/ICIW.2007.71
15. Boswell, D., Foucher, T.: The Art of Readable Code. O'Reilly, Sebastopol (2011)
16. Bouguettaya, A., et al.: A service computing manifesto: the next 10 years. Commun. ACM **60**(4), 64–72 (2017). https://doi.org/10.1145/2983528
17. Caillaud, B., Darondeau, P., Lavagno, L., Xie, X. (eds.): Synthesis and Control of Discrete Event Systems. Springer, New York (2002). https://doi.org/10.1007/978-1-4757-6656-1
18. https://github.com/davidebasile/ContractAutomataLib

19. https://github.com/davidebasile/ContractAutomataApp
20. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008). https://doi.org/10.1145/1327452.1327492
21. Ferrari, A., Mazzanti, F., Basile, D., ter Beek, M.H., Fantechi, A.: Comparing formal tools for system design: a judgment study. In: Proceedings of the 42nd International Conference on Software Engineering (ICSE 2020), pp. 62–74. ACM (2020). https://doi.org/10.1145/3377811.3380373
22. https://github.com/davidebasile/ContractAutomataLib/blob/old-backup/src/FMCA/FMCA.java#L1200 . Lines 1200–1378 contain the orchestration synthesis, lines 1385–1596 the choreography synthesis (the utility methods are not counted)
23. Garavel, H., Beek, M.H., van de Pol, J.: The 2020 expert survey on formal methods. In: ter Beek, M.H., Ničković, D. (eds.) FMICS 2020. LNCS, vol. 12327, pp. 3–69. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58298-2_1
24. Gleirscher, M., Marmsoler, D.: Formal methods in dependable systems engineering: a survey of professionals from Europe and North America. Empir. Softw. Eng. **25**(6), 4473–4546 (2020). https://doi.org/10.1007/s10664-020-09836-5
25. Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., Lea, D.: Java Concurrency in Practice. Addison-Wesley, Amsterdam (2006)
26. Goorden, M.A., et al.: The road ahead for supervisor synthesis. In: Pang, J., Zhang, L. (eds.) SETTA 2020. LNCS, vol. 12153, pp. 1–16. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-62822-2_1
27. Goorden, M., van de Mortel-Fronczak, J., Reniers, M., Fokkink, W., Rooda, J.: The impact of requirement splitting on the efficiency of supervisory control synthesis. In: Larsen, K.G., Willemse, T. (eds.) FMICS 2019. LNCS, vol. 11687, pp. 76–92. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-27008-7_5
28. Guanciale, R., Tuosto, E.: PomCho: a tool chain for choreographic design. Sci. Comput. Program. **202** (2021). https://doi.org/10.1016/j.scico.2020.102535
29. Henderson, K., Salado, A.: Value and benefits of model-based systems engineering (MBSE): evidence from the literature. Syst. Eng. **24**(1), 51–66 (2021). https://doi.org/10.1002/sys.21566
30. Huisman, M., Gurov, D., Malkis, A.: Formal methods: from academia to industrial practice. A travel guide. arXiv:2002.07279 [cs.SE], February 2020. https://arxiv.org/abs/2002.07279
31. Khatchadourian, R., Tang, Y., Bagherzadeh, M.: Safe automated refactoring for intelligent parallelization of Java 8 streams. Sci. Comput. Program. **195** (2020). https://doi.org/10.1016/j.scico.2020.102476
32. Khatchadourian, R., Tang, Y., Bagherzadeh, M., Ahmed, S.: Safe automated refactoring for intelligent parallelization of Java 8 streams. In: Proceedings of the 41st International Conference on Software Engineering (ICSE 2019), pp. 619–630. IEEE (2019). https://doi.org/10.1109/ICSE.2019.00072
33. Khatchadourian, R., Tang, Y., Bagherzadeh, M., Ray, B.: An empirical study on the use and misuse of Java 8 streams. In: Wehrheim, H., Cabot, J. (eds.) FASE 2020. LNCS, vol. 12076, pp. 97–118. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45234-6_5
34. Komenda, J., Masopust, T., van Schuppen, J.H.: Supervisory control synthesis of discrete-event systems using a coordination scheme. Automatica **48**(2), 247–254 (2012). https://doi.org/10.1016/j.automatica.2011.07.008

35. Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes – a comprehensive study on real world concurrency bug characteristics. In: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2008), pp. 329–339. ACM (2008). https://doi.org/10.1145/1346281.1346323
36. Martin, R.C.: Clean Code. Prentice Hall, Upper Saddle River (2008)
37. https://jgraph.github.io/mxgraph/java/index.html
38. Najem, T.: A formal semantics for supporting the automated synthesis of choreography-based architectures. In: Proceedings of the 13th European Conference on Software Architecture (ECSA 2019), vol. 2, pp. 51–54. ACM (2019). https://doi.org/10.1145/3344948.3344949
39. Ng, N., Yoshida, N.: Static deadlock detection for concurrent go by global session graph synthesis. In: Proceedings of the 25th International Conference on Compiler Construction (CC 2016), pp. 174–184. ACM (2016). https://doi.org/10.1145/2892208.2892232
40. Peltz, C.: Web services orchestration and choreography. IEEE Comput. **36**(10), 46–52 (2003). https://doi.org/10.1109/MC.2003.1236471
41. Ramadge, P.J., Wonham, W.M.: Supervisory control of a class of discrete event processes. SIAM J. Control. Optim. **25**(1), 206–230 (1987). https://doi.org/10.1137/0325013
42. Tockey, S.: How to Engineer Software: A Model-Based Approach. Wiley, Hoboken (2019)
43. Warburton, R.: Java 8 Lambdas: Pragmatic Functional Programming. O'Reilly, New York (2014)