Playing with our CAT and Communication-Centric Applications

Davide Basile[1 2], Pierpaolo Degano[2], Gian-Luigi Ferrari[2], and Emilio Tuosto[3]

[1] dbasile@isti.cnr.it I.S.T.I "A.Faedo" CNR Pisa, Italy
[2] {degano,giangi}@di.unipi.it Dept. of Computer Science University of Pisa, Italy
[3] emilio@le.ac.uk Dept. of Computer Science University of Leicester, UK

Abstract. We describe CAT, a toolkit supporting the analysis of communication-centric applications, i.e., applications consisting of ensembles of interacting services. The theoretical underpinning of CAT are *contract automata*, a recent model of service composition, in which communication safety is attained through some notions of *agreement properties*. Once services are modelled as contract automata, CAT enables (*i*) verifying the agreement properties, (*ii*) synthesising the orchestrator that enforces communication safety, (*iii*) detecting misbehaving services, and (*iv*) checking when the services form a choreography (i.e., when a fully distributed composition can safely replace the orchestrator). A distinguished characteristic of CAT is the use of mixed-integer linear programming to verify properties of interest.
We use a simple (albeit non trivial) example to demonstrate CAT.

1 Introduction

Communication is increasingly recognised as a key aspect of modern applications. The relevance of communication in distributed applications is undisputed. Remote procedure/method invocations are loosing their prominence at the application level due to their poor scalability, crucial in modern distributed applications. Communication-based modelling is also appealing for non-distributed software. For instance, application-level protocols can be devised to specify the behavioural constraints ensuring the correct use of a library or of an off-the-shelf component. This trend is also witnessed by the growth that paradigms such as service-oriented or cloud computing had in the software industry. In this context, composition of software becomes paramount and requires proper theoretical foundations as well as tool support. In fact, although scalable, communication-centric applications may pose non trivial obstacle to validation.

We showcase CAT, a prototype toolkit supporting the validation of communication-centric applications. This toolkit (available at `https://github.com/davidebasile/workspace`) is based on *contract automata* [4,5,3], a recently proposed formal model of service composition (surveyed in Appendix). Contract automata abstractly model (the communication pattern of) services as automata whose transitions represent **requests** and **offers**. An interaction between two services occurs when a **match** action is possible, that is when one service's offer matches a partner's request. Intuitively, contract automata capture the behaviour of services by tracking the interactions they are keen to

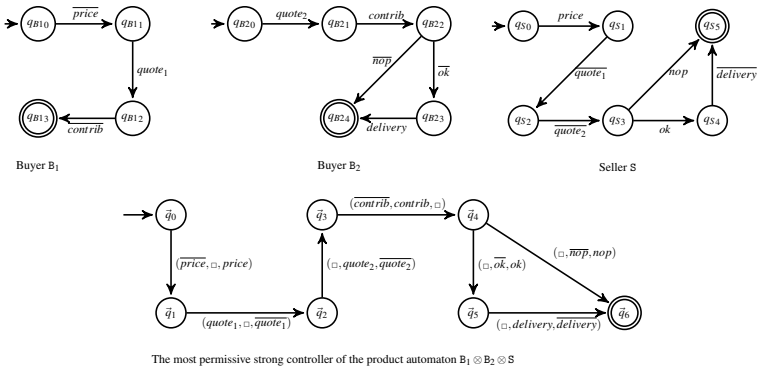

Fig. 1. The contract automata for 2BP

execute with each other. Service composition is naturally described in terms of product automata. The matching between offers and requests has to guarantee **agreement** properties that amount to **safe** communications.

By means of an example we describe how the analysis of communication-centric applications can be supported by CAT. To this purpose we borrow here the two-buyers protocol (2BP) from [9] which we now briefly recall. Two buyers, say $\mathtt{B}_1$ and $\mathtt{B}_2$, collaborate in purchasing an item from a seller S. Buyer $\mathtt{B}_1$ starts the protocol by asking S the price of the desired item (*price*); the seller S makes an offer by sending the message $quote_1$ to $\mathtt{B}_1$ and the message $quote_2$ to $\mathtt{B}_2$. Once received its quote, buyer $\mathtt{B}_1$ sends to $\mathtt{B}_2$ its contribution for purchasing the item (message *contrib*). Buyer $\mathtt{B}_2$ waits for the quote from S and the contribution from $\mathtt{B}_1$. Then, it decides whether to terminate by issuing the *nop* message to S, or to proceed by sending an acknowledgement to S. In the latter case, S sends the item to $\mathtt{B}_2$ (*delivery*), while if it receives *nop* it terminates with no further action. Figure 1 shows the contract automata of $\mathtt{B}_1$, $\mathtt{B}_2$, and S where each interaction is split in offers (overlined labels) and requests (non overlined labels).

We will apply CAT to the above protocol and show how, when the agreement property of interest is violated, we identify and fix defects.

2 CAT at work

We have implemented CAT in Java according to the simple architecture of Figure 2. The main class of CAT extends JAMATA, a framework for manipulating automata yielding methods for loading, storing, printing, and representing finite state automata. In other

Fig. 2. The architecture of CAT

words, CAT originally specializes JAMATA on contract automata, offering to the developers an API for creating and verifying contract automata. Also, CAT interfaces with a separate module for solving linear optimization problems, called AMPL, described in Section 3. This is an original facet of CAT; in fact, it maps the (check of) agreement properties of interest on a linear optimization problem.

The user of CAT has access to its API, which are partially shown here, see the Appendix for the remaining methods. The API can be conceptually classified as follows:

Automata operations consist of the methods `CA proj(int i)`, that returns the automaton specifying the i^{th} service of the composition, `CA product(CA[] aut)` and `CA aproduct(CA[] aut)` that compute respectively the product and the *associative* product of contract automata. Those products essentially interleave the actions of the services and yield the transitions corresponding to matching actions and they correspond to two different types of composition (transitions departing from unreachable states are removed). The `product` operation preserves existing match transitions, while the `aproduct` breaks such transitions and recombines them to account for all possible interleaving and matches due to new services joining an existing composition. Interestingly, `product` has to filter out the offers and request transitions when the source state has a corresponding outgoing match transition. Method `aproduct` is built on top of `product` by invoking `product` on the services obtained as projections of the automaton in input.

Safety check consists of the instance methods `safe`, `agreement`, `strongAgreement`, and `strongSafe` returning `true` if the corresponding agreement property holds on the contract automaton. Intuitively, an automaton admits strong agreement if it has at least one trace made only by match transitions; and it is strongly safe if all the traces are in strong agreement. Basically, strong agreement guarantees that the composition of services has a sound execution, while strong safety guarantees that *all* executions of the composition are sound. Likewise for *agreement* but for the fact that traces also admit (unmatched) offers to model interactions with an external environment. Section 3 discusses the property of *weak agreement*.

Controllers consist of the methods `CA mpc()` and `CA smpc()` that return the *most permissive controller* (MPC), for respectively agreement and strong agreement. A controller basically represents the largest (strongly) safe sub-automaton and is obtained through a standard construction of Control Theory [6].

We describe how to interact with the API of CAT through a simple command line interface (we plan to develop a GUI as well). The API is displayed and the user can choose one of the options (this is not shown here). Each displayed option corresponds to one of the methods described above. For instance, after choosing to compute a product, the user is asked to set the contract automata on which to take the product:

Output 1

```
Do you want to create/load other contract automata? yes
Insert the name of the automaton  to load or leave empty for create a new one: B1

Contract automaton:
Rank: 1, Number of states: [4], Initial state: [0], Final states: [[3]]
Transitions: ([0],[1],[1]), ([1],[-2],[2]), ([2],[3],[3])

Do you want to create/load other contract automata? yes
```

The user inputs the automata in CAT by providing their file names (line 2 of Output 1) and `yes` on line 8 until there are no more automata to load (in which case the user enters `no` to obtain the result of the product). For each entered automaton, CAT prints a textual description on the screen (lines 4-6 in Output 1) reporting the rank, initial and final states, and the list of transitions. The transitions are triples $(\mathtt{s},\mathtt{l},\mathtt{t})$ where `s` is the source state, `l` is the label, and `t` is the target state. These elements are lists of lenght r (the rank of the automaton), for instance, in Output 1 $r = 1$ (cf. on line 5). The i-th element of each list corresponds to the i-th service. In particular, the i-th action in the list of labels identifies the action perfomed by the i-th service; such action is strictly positive (if the action is an offer), strictly negative (if it is a request), and 0 if the service is idle in the transition. For `B1`, actions $\overline{price}$, $quote_1$, and $\overline{contrib}$ are represented with the integers 1, -2, and 3, respectively.

After computing the product automaton, CAT displays the result (`B1xB2xS` in our example and stores it in a file named `B1xB2xS.data`). From the main menu, the user can now choose to compute the MPC of the product automaton (shown in Figure 1); the result is displayed in Output 2 below. Once the product automaton is loaded, CAT will compute the MPC:

Output 2

```
The most permissive controller for strong agreement is:                                  1
Rank: 3                                                                                  2
Number of states: [4, 5, 6], Initial state: [0, 0, 0], Final states: [[3][4][5]]         3
Transitions:                                                                             4
([0, 0, 0],[1, 0, -1],[1, 0, 1])  ([1, 0, 1],[-2, 0, 2],[2, 0, 2])                       5
([2, 1, 3],[3, -3, 0],[3, 2, 3])  ([2, 0, 2],[0, -7, 7],[2, 1, 3])                      6
([3, 2, 3],[0, 4, -4],[3, 3, 4])  ([3, 2, 3],[0, 5, -5],[3, 4, 5])                      7
([3, 3, 4],[0, -6, 6],[3, 4, 5])                                                        8
Do you want to save this automaton? (write yes or no) yes  KS_B1xB2xS                    9
```

The resulting automaton is of rank 3 and corresponds to the MPC of Figure 1. The final states are represented as a list where the i-th element is the list of the final states of the i-th service. This representation allows to check if a state of the MPC is final or not without needing to explicitly enumerate all the final states of the MPC.

The transitions on lines 5-8 in Output 2 represent the transitions of the MPC; note that in each transition there is always an idle services. For instance, the transition ([0, 0, 0],[1, 0, -1],[1, 0, 1]) corresponds to the transition $(\vec{q}_0,(\overline{price},\square,price),\vec{q}_1)$ of the MPC in Fig. 1 (the second component of the label is 0 because `B2` is idle). The MPC can now be saved in a file as per line 9 in Output 2.

The underlying coordination mechanism of contract automata is orchestration. More precisely, services are oblivious of their partners and exchange messages through a "hidden" orchestrator (formalised by the MPC, if any). Whenever possible, one would like to have services interacting without the "supervision" of an orchestrator, using FIFO buffers. Mild conditions [3] ensure that the interactions are sound, in other words the services form a sound choreography. We briefly discuss this issue below.

For synchronous interactions (where buffers have size 1 and a single buffer may be non empty), services have to enjoy the *branching condition* that is necessary and sufficient for services to form a sound choreography. Basically, the branching condition holds if the actions of a service are not affected by the states of the other services in the composition. As said, a branching condition guarantees "unsupervised"

communications soundness when the communication are synchronous [3,5]. However, such branching condition does not suffice for asynchronous interactions (namely when buffers are unbounded and more than one buffer is possibly non empty). In this case it is necessary to check also for the absence of *mixed choice states* (i.e., states where more than one service can perform an offer). The methods of CAT for checking the branching condition and mixed choice states are discussed in the Appendix. Consider now Fig. 1, where in Output 3 the state $\vec{q}_2$ corresponds to `[2,0,2]`, the state $\vec{q}_3$ to `[2,1,3]`, and the transition $(\overline{contrib}, contrib, \square)$ to the label `[3,-3,0]`. The MPC does not enjoy the branching condition, as CAT reports:

```
Output 3
State [2,0,2] violates the branching condition because it has no transition labelled
 [3,-3,0] which is instead enabled in state [2,1,3]
```

It is important to observe that the message in Output 3 also flags states and transitions for which the condition is violated. We discuss the problem by considering the automata in Fig. 1. The local state of buyer $\mathtt{B}_1$ in $\vec{q}_2$ and $\vec{q}_3$ is q_{B12}, while the locale state of $\mathtt{B}_2$ in $\vec{q}_2$ is q_{B20}, and in $\vec{q}_3$ is q_{B21}. Therefore, in the case that $\mathtt{B}_2$ is in local state q_{B20} where it is waiting for $\overline{quote}_2$, without an orchestrator the offer $\overline{contrib}$ from $\mathtt{B}_1$ could fill up the 1-buffer of $\mathtt{B}_2$, leading to a deadlock. A simple fix consists in swapping the order in which the quotes are sent by the seller; CAT reports that the amended protocol (not shown here) enjoys the branching condition. The contract automaton has no mixed choice states, as detected by CAT. A mixed choice state could be introduced in 2BP if, e.g., $\mathtt{B}_2$ could send the acknowledgement to S or receive $\overline{contrib}$ from $\mathtt{B}_1$ in any order. For this variant of 2BP CAT finds the mixed choice state.

3 Linear programming and contract automata

We briefly review a component for solving optimization problems related to contract automata, that complements the functionalities offered by CAT. This component reduces the analysis of the context-sensitive properties of *weak agreement* [4] to a linear programming problem and relies on an ad hoc solver as explained below.

The properties of weak agreement were introduced for solving circularity issues, in which all services are stuck waiting the fulfilment of their requests before providing the corresponding offers [4]. For example, consider the services (rendered as regular expressions) $A = a.\overline{b}$ and $B = b.\overline{a}$; their product does not admit agreement. Circularity is solved by allowing matches between requests and offers even though they are not simultaneous; intuitively, offers may be fired "on credit" provided that the corresponding requests are honoured later on. A trace of an automaton is a weak agreement if for each request there is a corresponding offer, no matter in which order they occur in the trace. The notions of *admitting* weak agreement and of *weakly* safety are then similar to the ones of (strong) agreement reviewed earlier. For example, $A \otimes B$ admits weak agreement. The underling theory and the decision procedures for the properties of weak agreement are developed in [4], and are formalised as mixed linear integer programming. The decision procedures are implemented in *A Mathematical Programming Language* (`AMPL`) [7], a widely used language for describing and solving optimization problems. In this way, the automatic verification of contract automata under properties

of weak agreement exploits efficient techniques and algorithms developed in the area of operational research. We now briefly describe the implementation of the techniques for verifying weak agreement (the AMPL code is provided in the Appendix). The script flow.run, to be launched with the command ampl, is described below:

flow.run

```
#reset;                                                                    1
option solver cplex;              // use the simplex algorithm in C        2
model weakagreement.mod;          // select model for weak agreement       3
data flow.dat;                    // load                                  4
solve;                            // apply the simplex algorithm           5
display gamma;                    // display the result: if gamma >= 0 then property holds  6
```

The script firstly loads the automaton from the file flow.dat (line 4). The description of the automata consists of the number of nodes, the cardinality of the alphabet of actions, and a matrix of transitions for each action a, where there is value 0 at position (s,t) if there is no transition from state s to state t labelled by a, and respectively 1 or -1 if there is an offer or request transition on a. In this case, the contract automata described in flow.dat is representative.

The AMPL linear program to load is given as input parameter to the script (line 3). The two optimization problems available are: weakagreement.mod, the file contains the formalization of the optimization problem for deciding whether a contract automata admits weak agreement, and weaksafety.mod that contains the formalization of the optimization problem for deciding whether a contract automata is weakly safe.

Both formal descriptions are then solved using the solver cplex, that is the simplex method implemented in C. However it is possible to select other available solvers in the script flow.run (line 2). The execution of the script will prompt to the user the value of variables. As proved in [4], if the variable gamma is non negative then the contract automata satisfies the given property. Bi-level optimization problems can not be defined directly in AMPL. Therefore, we cannot plainly apply formalisation of [4] for representing weakly liable transitions (see Appendix) as an optimization problem. However, different techniques of relaxation of the bi-level problem for over approximating the set of weakly liable transitions can be used, as for example lagrangian relaxation. As future work, we are planning to develop a toolchain for fully integrating the above techniques in CAT, in order to reuse them for the functionalities described in Section 2. In particular, CAT will automatically generate a contract automata description flow.dat, execute the script flow.run and collect the results.

4 Concluding Remarks

We described CAT, a tool supporting the analysis of communication-centric applications attained with novel techniques based on combinatorial optimization. A non trivial example was used to show main features of CAT.

An interesting application domain for CAT are service-oriented applications. In this context, model-driven approaches have been advocated for the analysis of service composition. In particular, automata have been used as target models to translate BPEL processes [10] in [12,8]; for instance, constraint automata semantics of REO [1,2] is used

in [11] to analyse web-services. Relations of contract automata with service composition are studied in [4,5,3]. The properties verified by CAT have not been considered by other approaches. For example, the identification - even in presence of circular dependencies of services (see Section 3) - of liable transitions that may spoil a composition complement the verification done in [11] (see Appendix for details about liable transitions). We conjecture that it would be possible to define model transformations from contract automata to BPEL which preserve the analysis discussed here.

A model-driven approach would also ease the integration of CAT with e.g., the tools discussed above. This would provide developers with a wide variety of tools for guaranteeing the quality of the composition of services according to different criteria.

The tool is still a prototype; we plan to improve its efficiency, extend it with new functionalities (e.g., relaxation), and improve its usability (e.g., adding a user-friendly GUI and pretty-printing automata). We note that CAT provides a valid support to the analysis of applications. In fact, CAT is able to detect possible violations of the properties of interest (for example branching condition, mixed choice). A drawback of CAT is that it does not support modelling and design of applications. An interesting evolution of CAT would be to add functionalities for amending applications violating properties of interest. For instance, once liable transitions are identified, CAT could suggest how to modify services to guarantee the property. This may also be coupled with the model-driven approach by featuring functionalities traceing transitions in the actual source-code of services.

References

1. F. Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.
2. C. Baier, M. Sirjani, F. Arbab, and J. J. M. M. Rutten. Modeling component connectors in reo by constraint automata. *Sci. Comput. Program.*, 61(2):75–113, 2006.
3. D. Basile, P. Degano, G. Ferrari, and E. Tuosto. Relating two automata-based models of orchestration and choreography. *JLAMP*, 2015.
4. D. Basile, P. Degano, and G. L. Ferrari. Automata for analysing service contracts. In M. Maffei and E. Tuosto, editors, *TGC 2014*, volume 8902 of *LNCS*, pages 34–50. Springer, 2014.
5. D. Basile, P. Degano, G. L. Ferrari, and E. Tuosto. From orchestration to choreography through contract automata. In *ICE 2014*, pages 67–85, 2014.
6. C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
7. R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL: A mathematical programming language*. AT&T Bell Laboratories Murray Hill, NJ 07974, 1987.
8. X. Fu, T. Bultan, and J. Su. Analysis of Interacting BPEL Web Services. In *WWW '04*, pages 621–630. ACM, 2004.
9. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In G. C. Necula and P. Wadler, editors, *POPL*, pages 273–284. ACM, 2008.
10. M. B. Juric. *Business Process Execution Language for Web Services BPEL and BPEL4WS 2Nd Edition*. Packt Publishing, 2006.
11. S. Tasharofi, M. Vakilian, R. Z. Moghaddam, and M. Sirjani. Modeling Web Service Interactions Using the Coordination Language Reo. In *WS-FM*, volume 4937 of *LNCS*. Springer.
12. A. Wombacher, P. Fankhauser, and E. Neuhold. Transforming BPEL into annotated deterministic finite state automata for service discovery. In *Web Services, 2004*, 2004.

A Contract Automata

A contract automata (cf., Def. 2 below) represents the behaviour of a set of *principals* capable of performing some *actions*; more precisely, as formalised in Def. 1, the actions of contract automata allow them to "advertise" offers, "make" requests, or "handshake" on match actions, that is simultaneous offer/request actions. Consequently, transitions of contract automata will be labelled with tuples of elements in the set $\mathbb{L} \stackrel{\text{def}}{=} \mathbb{R} \cup \overline{\mathbb{R}} \cup \{\square\}$ where: requests of principals will be built out of $\mathbb{R}$ while their *offers* will be built out of $\overline{\mathbb{R}}$, $\mathbb{R} \cap \overline{\mathbb{R}} = \emptyset$, and $\square \notin \mathbb{R} \cup \overline{\mathbb{R}}$ is a distinguished label to represent components that stay idle. We let $a, b, c, \ldots$ range over $\mathbb{L}$ and fix an involution $\overline{\cdot} : \mathbb{L} \to \mathbb{L}$ such that

$$\overline{\mathbb{R}} \subseteq \overline{\mathbb{R}}, \qquad \overline{\overline{\mathbb{R}}} \subseteq \mathbb{R}, \qquad \forall a \in \mathbb{R} \cup \overline{\mathbb{R}} : \ \overline{\overline{a}} = a, \quad \text{and} \quad \overline{\square} = \square$$

As usual, offer actions will be topped by bar (i.e. $\overline{a}$). Let $\vec{v} = (a_1, ..., a_n)$ be a vector of *rank* $n \geq 1$, in symbols r_v, then $\vec{v}_{(i)}$ denotes the i-th element. We write $\vec{v}_1 \vec{v}_2 \ldots \vec{v}_m$ for the concatenation of m vectors $\vec{v}_i$, while $|\vec{v}| = n$ is the rank (length) of $\vec{v}$ and $\vec{v}^n$ for the vector obtained by n concatenations of $\vec{v}$.

Definition 1. *A tuple* $\square^* b \square^*$ *on* $\mathbb{L}$ *is a* request (action) on b *iff* $b \in \overline{\mathbb{R}}$. *A* match (action) on b *is a tuple* $\square^* b \square^* \overline{b} \square^*$ *on* $\mathbb{L}$ *with* $b \in \mathbb{R} \cup \overline{\mathbb{R}}$. *The relation* $\dot{\bowtie} \subseteq \mathbb{L}^* \times \mathbb{L}^*$ *is the symmetric closure of* $\dot{\bowtie} \subseteq \mathbb{L}^* \times \mathbb{L}^*$ *where* $\vec{a}_1 \dot{\bowtie} \vec{a}_2$ *iff* $|\vec{a}_1| = |\vec{a}_2|$ *and both the following conditions hold*

- $\exists b \in \mathbb{R} \cup \overline{\mathbb{R}} : \ \vec{a}$ *is either a request or an offer on* b*;*
- $\exists b \in \mathbb{R} \cup \overline{\mathbb{R}} : \ \vec{a}_1$ *is an offer on* $b \implies \vec{a}_2$ *is a request on* b
- $\exists b \in \mathbb{R} \cup \overline{\mathbb{R}} : \ \vec{a}_1$ *is a request on* $b \implies \vec{a}_2$ *is a offer on* b*.*

Definition 2. *Assume as given a finite set of states* $Q = \{q_1, q_2, \ldots\}$. *Then a* contract automaton $\mathcal{A}$ *of rank* n *is a tuple* $\langle Q, \vec{q_0}, A^r, A^o, T, F \rangle$*, where*

- $Q = Q_1 \times \ldots \times Q_n \subseteq Q^n$
- $\vec{q_0} \in Q$ *is the initial state*
- $A^r \subseteq \mathbb{R}, A^o \subseteq \overline{\mathbb{R}}$ *are finite sets (of requests and offers, respectively)*
- $F \subseteq Q$ *is the set of final states*
- $T \subseteq Q \times A \times Q$ *is the set of transitions, where* $A \subseteq (A^r \cup A^o \cup \{\square\})^n$ *and if* $(\vec{q}, \vec{a}, \vec{q'}) \in T$ *then both the following conditions hold:*
  - $\vec{a}$ *is either a request or an offer or a match*
  - *if* $\vec{a}_{(i)} = \square$ *then it must be* $\vec{q}_{(i)} = \vec{q'}_{(i)}$

A principal *is a contract automaton of rank 1 such that* $A^r \cap co(A^o) = \emptyset$.

Note that a principal is not allowed to make a request on actions that it offers. Below, we introduce two different operators for composing contract automata. Both products interleave all the transitions of their operands. We only force a synchronisation to happen when two contract automata are ready on their respective request/offer action. These operators represent two different policies of orchestration. The first operator is called simply *product* and it considers the case when a service S joins a group of services already clustered as a single orchestrated service S'. In the product of S and S', the first

can only accept the still available offers (requests, respectively) of S' and vice versa. In other words, S cannot interact with the principals of the orchestration S', but only with it as a whole component.

This is not the case with the second operation of composition, called *a-product*: it puts instead all the principals of S at the same level of those of S'. Any matching request-offer of either contracts can be split, and the offers and requests, that become available again, can be re-combined with complementary actions of S, and viceversa. The a-product turns out to satisfactorily model coordination policies in dynamically changing environments, because the a-product is a form of *dynamic orchestration*, that adjusts the workflow of messages when new principals join the contract.

We now introduce our first operation of composition; recall that we implicitly assume the alphabet of a contract automaton of rank m to be $A \subseteq (A^r \cup A^o \cup \{\square\})^m$.

Definition 3 (Product). *Let $\mathcal{A}_i = \langle Q_i, \vec{q}_{0i}, A^r_i, A^o_i, T_i, F_i \rangle, i \in 1 \dots n$ be contract automata of rank r_i. The* product $\bigotimes_{i \in 1 \dots n} \mathcal{A}_i$ *is the contract automaton $\langle Q, \vec{q}_0, A^r, A^o, T, F \rangle$ of rank $m = \sum_{i \in 1 \dots n} r_i$, where:*

- $Q = Q_1 \times \dots \times Q_n$, *where* $\vec{q}_0 = \vec{q}_{01} \dots \vec{q}_{0n}$
- $A^r = \bigcup_{i \in 1 \cdots n} A^r_i$, $A^o = \bigcup_{i \in 1 \cdots n} A^o_i$
- $F = \{\vec{q}_1 \dots \vec{q}_n \mid \vec{q}_1 \dots \vec{q}_n \in Q, \vec{q}_i \in F_i, i \in 1 \dots n\}$

- T *is the least subset of $Q \times A \times Q$ s.t. $(\vec{q}, \vec{c}, \vec{q}') \in T$ iff, when $\vec{q} = \vec{q}_1 \dots \vec{q}_n \in Q$,*
  either *there are $1 \leq i < j \leq n$ s.t. $(\vec{q}_i, \vec{a}_i, \vec{q}'_i) \in T_i$, $(\vec{q}_j, \vec{a}_j, \vec{q}'_j) \in T_j$, $\vec{a}_i \bowtie \vec{a}_j$ and*
  $$\begin{cases} \vec{c} = \square^u \vec{a}_i \square^v \vec{a}_j \square^z \text{ with } u = r_1 + \dots + r_{i-1},\ v = r_{i+1} + \dots + r_{j-1}, |\vec{c}| = m \\ \text{and} \\ \vec{q}' = \vec{q}_1 \dots \vec{q}_{i-1}\ \vec{q}'_i\ \vec{q}_{i+1} \dots\ \vec{q}_{j-1}\ \vec{q}'_j\ \vec{q}_{j+1} \dots \vec{q}_n \end{cases}$$
  or *there is $1 \leq i \leq n$ s.t. $(\vec{q}_i, \vec{a}_i, \vec{q}'_i) \in T_i$ and*
  $\vec{c} = \square^u \vec{a}_i \square^v$ *with $u = r_1 + \dots + r_{i-1}$, $v = r_{i+1} + \dots + r_n$, and*
  $\vec{q}' = \vec{q}_1 \dots \vec{q}_{i-1}\ \vec{q}'_i\ \vec{q}_{i+1} \dots \vec{q}_n$ *and*
  $\forall j \neq i, 1 \leq j \leq n, (\vec{q}_j, \vec{a}_j, \vec{q}'_j) \in T_j$ *it does not hold that $\vec{a}_i \bowtie \vec{a}_j$.*

There is a simple way of retrieving the principals involved in a composition of contract automata obtained through the product introduced above: just introduce projections $\prod^i$ as done below.

Definition 4 (Projection). *Let $\mathcal{A} = \langle Q, \vec{q}_0, A^r, A^o, T, F \rangle$ be a contract automaton of rank n, then the* projection *on the i-th principal is*
$\prod^i(\mathcal{A}) = \langle \prod^i(Q), \vec{q}_{0(i)}, \prod^i(A^r), \prod^i(A^o), \prod^i(T), \prod^i(F) \rangle$ *where $i \in 1 \dots n$ and:*

$$\prod^i(Q) = \{\vec{q}_{(i)} \mid \vec{q} \in Q\} \quad \prod^i(F) = \{\vec{q}_{(i)} \mid \vec{q} \in F\} \quad \prod^i(A^r) = \{a \mid a \in A^r, (q, a, q') \in \prod^i(T)\}$$

$$\prod^i(A^o) = \{\overline{a} \mid \overline{a} \in A^o, (q, \overline{a}, q') \in \prod^i(T)\} \quad \prod^i(T) = \{(\vec{q}_{(i)}, \vec{a}_{(i)}, \vec{q'}_{(i)}) \mid (\vec{q}, \vec{a}, \vec{q'}) \in T \wedge \vec{a}_{(i)} \neq \square\}$$

Definition 5 (a-Product). *Let $\mathcal{A}_1, \mathcal{A}_2$ be two contract automata of rank n and m, respectively, and let $I = \{\prod^i(\mathcal{A}_1) \mid 0 < i \leq n\} \cup \{\prod^j(\mathcal{A}_2) \mid 0 < j \leq m\}$. Then the* a-product *of $\mathcal{A}_1$ and $\mathcal{A}_2$ is $\mathcal{A}_1 \boxtimes \mathcal{A}_2 = \bigotimes_{\mathcal{A}_i \in I} \mathcal{A}_i$.*

Note that if $\mathcal{A}, \mathcal{A}'$ are principal contract automata, then $\mathcal{A} \otimes \mathcal{A}' = \mathcal{A} \boxtimes \mathcal{A}'$.

B Detailing the Implementation of CAT

CAT consists of a class `CAUtil` and of other classes `CA` and `CATransition`, extending two corresponding super-classes of JAMATA. The class `CA` provides the main functionalities of CAT; its instance variables capture the basic structure of our automata:

- `int rank` is the rank of the automaton;
- `int[] initial` is the initial state of the automaton (the array is of size `rank`);
- `int[] states` the vector of the number of local states of each principal in the contract automata (the array is of size `rank`);
- `int[][] finalstates` the final states of each principal in the contract automata;
- `CATransition[] tra` the transitions of the contract automata.

The n local states of a principal are represented as integers in the range $0,\ldots,n-1$; in this case, `states.length = 1` and `states[0] = n`. The state of an automaton of rank $m > 1$ is an m-vector `states` such that `states[i]` yields the number of states of the i^{th} principal. This low-level representation (together with the encoding of actions and labels as integers) enabled us to optimize space.

The class `CATransition`, describes a transition of a contract automata. The instance variables of a `CATransition` object are:

- `int[] source` (the starting state of the transition);
- `int[] label` (the label of the transition);
- `int[] target` (the arriving state of the transition).

The class `CATransition` provides methods to extract its instance variables, to check if the transition is an offer, a request or a match, and to extract the (index of the) principal performing the offer, if any. The methods not discussed in Section 2 are:

Liable detection consists of the methods `CATransition[] liable()` - returning transitions from a state s to a state t such that s is in the MPC but t is not - and `CATransition[] strongLiable()` that similarly returns such transitions for the MPC of the strong agreement property. In particular, liable services are those responsible for leading a contract composition into a failure.

Decentralization includes `int[][] branchingCondition()`, that returns two states and an action for which the branching condition is violated. Another similar method is `int[][] extendedBranchingCondition()` which deals with open-ended interactions. The last method in this category is `int[] mixedChoice()` that returns a mixed-choice state (a state where a principal has enabled both offers and requests inside matches). All such methods return `null` when the conditions they check do not hold.

C AMPL Code

The code of `weakagreement.mod` and `weaksafety.mod` is depicted in Figure 3. For further details about CAT, we refer the interested reader to the full documentation, available online at
`https://github.com/davidebasile/workspace/tree/master/JaMata/doc`.

weakagreement.mod

```
# n number of nodes # m number of actions
param n; param m; param K; param final; #final node
set N := {1..n}; set M := {1..m}; param t{N,N}; param a{N,N,M};
var x_t{N,N} >=0 integer;  var z_t{N,N,N} >=0;
var gamma; var p{N} binary; var wagreement;

#flow constraints
subject to Flow_Constraints {node in N}:
         sum{i in N}( x_t[i,node]*t[i,node] ) - sum{i in N}(x_t[node,i]*t[node,i]) =
         if (node == 1) then  -1
         else if (node == final)  then 1
         else 0;
;

subject to p1{node in N}: p[node] <= sum{i in N}( x_t[node,i]*t[node,i]);
subject to p2{node in N}: sum{i in N}( x_t[node,i]*t[node,i]) <= p[node]*K;

subject to Auxiliary_Flow_Constraints {snode in N diff {1},node in N}:
        sum{i in N}( z_t[snode,i,node]*t[i,node] ) - sum{i in N}(z_t[snode,node,i]*t[node,i]) =
        if (node == 1) then    - p[snode]
        else if (node == snode)  then  p[snode]
        else 0;

subject to Auxiliary_Flow_Constraints2{i in N, j in N,snode in N}:
        z_t[snode,i,j]*t[i,j] <= x_t[i,j]*t[i,j];

subject to threshold_constraint {act in M}:
  sum{i in N,j in N} x_t[i,j]*t[i,j]*a[i,j,act] >= gamma;

#objective function
maximize cost: gamma;
```

weaksafety.mod

```
# n number of nodes # m number of actions
param n; param m;  param K; param final; #final node
set N := {1..n}; set M := {1..m}; param t{N,N}; param a{N,N,M};
var x_t{N,N} >=0 integer; var z_t{N,N,N} >=0;
var gamma; var p{N} binary; var v{M} binary; var wagreement;

#flow constraints
subject to Flow_Constraints {node in N}:
         sum{i in N}( x_t[i,node]*t[i,node] ) - sum{i in N}(x_t[node,i]*t[node,i]) =
         if (node == 1) then  -1
         else if (node == final)  then 1
         else 0;
;

subject to p1{node in N}: p[node] <= sum{i in N}( x_t[node,i]*t[node,i]);
subject to p2{node in N}: sum{i in N}( x_t[node,i]*t[node,i]) <= p[node]*K;

subject to Auxiliary_Flow_Constraints {snode in N diff {1},node in N}:
        sum{i in N}( z_t[snode,i,node]*t[i,node] ) - sum{i in N}(z_t[snode,node,i]*t[node,i]) =
        if (node == 1) then    - p[snode]
        else if (node == snode)  then  p[snode]
        else 0;

subject to Auxiliary_Flow_Constraints2{i in N, j in N,snode in N}:
        z_t[snode,i,j]*t[i,j] <= x_t[i,j]*t[i,j];

subject to vi: sum{i in M} v[i] = 1;

subject to threshold_constraint :
sum{act in M,i in N,j in N} (v[act]*x_t[i,j]*t[i,j]*a[i,j,act]) <= gamma;

#objective function
minimize cost: gamma;
```

Fig. 3. The implementation in AMPL of the optimization problem for deciding weakagreement and weak safety.