Abstractions for Collective Adaptive Systems*

Omar Inverso[1], Catia Trubiani[1], and Emilio Tuosto[1,2]

[1] Gran Sasso Science Institute, Italy
{omar.inverso, catia.trubiani, emilio.tuosto}@gssi.it
[2] University of Leicester

Abstract This paper advocates behavioural abstractions for the coordination of *collective adaptive systems* (CAS). In order to ground the discussion in a concrete framework, we sketch mechanisms based on behavioural types for some recently proposed calculi that formalise CAS. We analyse new typing mechanisms and show that such mechanisms enable formal specifications of CAS that cannot be easily attained through existing behavioural types. We illustrate how a quantitative analysis can be instrumented through our behavioural specifications by means of a case study in a scenario involving autonomous robots. Our analysis is auxiliary to our long term aim which is three-fold: (i) study suitable typing mechanisms for CAS, (ii) identify basic properties of CAS that may be enforced by typing, and (iii) consider quantitative properties of CAS.

1 Introduction

Collective adaptive systems (CAS) consist of computational agents that collaborate and take decisions to achieve some goals [17]. Typically, agents of CAS are autonomous units that coordinate distributively. Namely, there is no central unit and the overall state of the computation is fragmented in the local state of agents. Also, these agents execute in a cyber-physical context and are supposed to adapt to changes of these open-ended environments. Hence, at a given moment, agents have a partial knowledge about the overall state of the computation. For each agent, such knowledge is made of information possibly acquired autonomously (e.g., by sensing and elaborating information about the surrounding environment) or communicated by other agents. Decisions are *local*: agents use their knowledge to establish the next course of action.

We will use a simple scenario to illustrate some key features of CAS. The scenario is centred around the *stable marriage problem* [20] (SM), that finds applications in many domains to determine matching of resources. Given two equally-sized sets of elements, each having individual preferences (in form of an ordered list) on the elements of the other group, Stable Marriage (SM) amounts to finding a stable matching between the elements of the two groups. The original

* Research partly supported by the EU H2020 RISE programme under the Marie Skłodowska-Curie grant agreement No 778233. Work partly funded by MIUR projects PRIN 2017FTXR7S *IT MATTERS* and PRIN 2017TWRCNB *SEDUCE*.

formulation was described in terms of groups of women and men, whence the word marriage originates; in the following, we retain the original terminology.

The SM protocol can be solved by pairing men to women so that no man and woman would rather prefer each other to their current partner. In the classical solution of [20] each man proposes himself to his most favourite woman, according to his preferences. When a man's offer is rejected or a woman drops her current partner due to a better pretender, the man tries with his next preferred woman. A woman accepts any proposal when single or, depending on her preferences, she chooses the best man between her current partner and the one making advances, abandoning her current partner in the second case. The SM protocol guarantees the existence of a unique stable matching.

The following python-like pseudocode describes the use of the algorithm of [20] for the classical SM protocol where agents use preference lists to take their decisions.

```
def B(prefs, myID):
  ... # code to handle prefs
  for charger in prefs:
    send("p", myID) at charger
    recv(res)
```

```
def C(aID, aPID):
  while true:
    recv("p", idNew)
    if choose(aPID, idNew):
       send("no") to aPID
    else: send("no") to idNew
```

Assume that the parameter *`ID` is used to identify agents; for instance, `B` communicates its identifier `myID` to `C` and the latter may use it in further communications.

In a CAS scenario, we can imagine a number of agents executing either (or both) the snippets above in order to "pair up": for instance, an autonomous agent roaming in a smart city in need to recharge its battery may execute `B` to search for a charger agent executing `C`. Now suppose that chargers are not available in the immediate proximity of an agent needing a recharge.

In such a scenario, a key element for *correctness*[3] is therefore *how* information spreads among agents. For instance, the message sent by `B` should reach a charger. Simple communication mechanisms such as point-to-point (p2p) communications present some limitations in this application domain. The main drawback of these mechanisms is that they impose the design of ad-hoc communication protocols to (1) identify partners, (2) disseminate information, (3) update local knowledge of agents. In the snippet above, partners include identifiers and it is assumed that the communication infrastructure delivers messages properly. This requires to configure agents appropriately: the identifiers used must exist, be unique, and immutable. For instance, malfunctions can arise when renaming chargers or assigning the same name to different chargers. Moreover, the deployment of new chargers would go "unnoticed" to existing agents unless updating their preference list. Besides p2p communications have an overhead since multi-party interactions are commonplace in CAS.

As reviewed in Sec. 7, a number of proposals have indeed been made to provide suitable communication mechanism. Those approaches aim to provide suitable linguistic abstractions to specify CAS. For instance, the calculus of attribute-based communication (AbC) [1] allows one to specify the scenario above by addressing agents according to their attributes so that, e.g., `B` can send a request to *any* charger within a given distance and with a given amperage and cost.

[3] We will come back to correctness soon.

However, those mechanisms are complex and often hard to enforce in communication infrastructures. Also, it is difficult to understand how those mechanisms can be used to guarantee interesting properties of CAS (cf. Sec. 7). This point can be explained with an analogy. Synchronous communications allow the designer to assume that after an interaction the receiving agent has acquired the information. Said differently, this type of interaction allows the sender to make safe assumptions on the state of the receiver. This has important consequences on the way one can reason about the system. So, fixed a sophisticated communication mechanism, natural questions to ask are:

1. "What can be safely assumed about the distributed state of the system after an interaction among agents?"
2. "What (behavioural) properties a given communication mechanism enforces?"
3. "How can one *statically* validate such properties?"
4. "Can behavioural abstractions support or improve run-time execution?"
5. "Can behavioural specifications foster quantitative analysis of CAS?"

This paper discusses the characteristics of behavioural abstractions for CAS. We argue that existing frameworks are not suitable for the specification or the static analysis of CAS. For the sake of the argument we will use AbC as an illustrative language and sketch behavioural abstractions tailored on the linguistic mechanisms of AbC. Giving a full fledged typing discipline is out of the scope of this paper. Our main goal is to discuss what are the main features that behavioural types for CAS should provide and how such features can be used to support quantitative analysis of CAS.

Outline. Sec. 2 surveys AbC using the SM protocol. Sec. 3 sketches a behavioural type for CAS. For the sake of the presentation, we will refrain from technical definitions and give only informal descriptions to motivate our proposal in Sec. 4. Sec. 5 uses the simple examples in Sec. 3 and a case study from the robotic domain for examing the questions above. Sec. 6 shows how the behavioural types discussed here could be helpful for quantitative analysis of CAS. Related and future work are discussed in Sec. 7 together with concluding remarks.

2 A Bird-eye View of AbC

The calculus of attribute-based communication [1] (AbC), was initially conceived as a simplified version of SCEL [16]. Intuitively, AbC can be seen as a generalisation of traditional message-passing. In message-passing, communication is typically restricted to either one-to-one (point-to-point) or one-to-all (broadcast). AbC relaxes this restriction by allowing one-to-many communication via attribute-based send and receive constructs. Interestingly, communication across dynamically-formed many-to-many groups can take place. This allows to model effortlessly several different classes of distributed systems, which would be harder to do under more traditional formalisms, such as Actors [3], channels [27], or broadcast [29], that rely on the identity of the communicating peers.

Informally, an AbC system consists of multiple communicating *components*. Depending on the domain and the interaction patterns of interest, each component exposes some *attributes*. A send operation can thus target all the components satisfying a given *predicate* on such attributes. Every component satisfying the sending predicate is thus selected as a potential receiver. At the other end, similarly, each potential receiver performing a receive operation eventually gets messages depending on its receiving predicate. Any other message is discarded. For instance, cooperating robots in a swarm may be modelled as separate components that expose their battery level and location. A robot can ask cooperation to robots within a certain distance and with a minimum battery level. However, a robot which is already busy may want to discard this invitation.

More concretely, the core syntax of (our variant of) AbC is as follows

$$
\begin{array}{llr}
C & ::= \mathit{\Gamma} : P \;\big|\; C_1 \parallel C_2 & \text{Components} \\
P & ::= 0 \;\big|\; \alpha.P \;\big|\; [\mathsf{a} := E]P \;\big|\; \langle \mathit{\Pi} \rangle P \;\big|\; P_1 + P_2 \;\big|\; P_1 \mid P_2 \;\big|\; K & \text{Processes} \\
E & ::= \mathtt{v} \;\big|\; x \;\big|\; \mathsf{a} \;\big|\; \mathtt{this.a} \;\big|\; \_x \;\big|\; (E) & \text{Expressions} \\
\alpha & ::= E@\mathit{\Pi} \;\big|\; \mathit{\Pi}(E) & \text{Actions} \\
\mathit{\Pi} & ::= \mathtt{true} \;\big|\; \mathtt{false} \;\big|\; E_1 \bowtie E_2 \;\big|\; \mathit{\Pi}_1 \wedge \mathit{\Pi}_2 \;\big|\; \neg \mathit{\Pi} \;\big|\; \{\mathit{\Pi}\} & \text{Predicates}
\end{array}
$$

An AbC system consists of multiple *components*. A component is either the parallel composition of two components, or a pair $\mathit{\Gamma} : P$, where $\mathit{\Gamma}$ denotes the attribute environment of the component, and P a process describing its behaviour. Processes exchange expressions E which may be a constant value $\mathtt{v}$, a variable x, an attribute name a, a reference $\mathtt{this.a}$ to an attribute a of the *local* environment, or a placeholder $\_x$ used by the pattern matching mechanism. Values include basic data types (e.g., atoms `'v'`, numeric or boolean values), arithmetic expressions, and tuples $\llparenthesis E_1, \dots, E_n \rrparenthesis$ including the empty tuple $\llparenthesis\rrparenthesis$.

The *attribute environment* $\mathit{\Gamma}$ is a mapping from attribute names to values. As mentioned, attributes expresses specific features of the components that are relevant for communication, for example the position of an agent on a grid, the battery level of a robot, the role of the component or its identity, etc. The set of attributes exposed by each component is statically fixed upfront.

A *process* P can be the inactive process 0 (hereafter omitted where possible), an action prefixing process $\alpha.P$, an *update* process $[\mathsf{a} := E]P$, an *awareness* process $\langle \mathit{\Pi} \rangle P$. Processes can be composed using the usual non-deterministic choice $P_1 + P_2$ and parallel composition constructs $P_1 | P_2$, and invocation K.

The prefixing process executes *action* α and continues as P. The update process sets attribute a to the value of expression E. The awareness process blocks the execution of process P until predicate $\mathit{\Pi}$ holds. Non-deterministic choice and parallel composition work as usual. Note that the awareness construct may be combined with non-deterministic choice to express branching.

An *attribute-based send* action $E@\mathit{\Pi}$ sends the evaluation of expression E to those components whose attributes satisfy predicate $\mathit{\Pi}$. An *attribute-based receive* action $\mathit{\Pi}(E)$ uses expressions E to pattern match expressions in send actions. More precisely, the occurrences of a placeholder $\_x$ are bound in E

and the continuation of the receive prefix; upon communication, $\_x$ is assigned the corresponding values in the message received from any component whose attributes (and possibly values communicated in the body of the message) satisfy predicate Π. Send operations are asynchronous while receive are blocking.

An update operation is performed atomically with the action following it, if the component under the updated environment can perform that action. It is possible to model an update operation in isolation $[\mathsf{a} := E]$ via an empty send action $(\!|\,|\!)@\mathtt{false}$ to obtain $[\mathsf{a} := E](\!|\,|\!)@\mathtt{false}$.

A predicate Π can be either $\mathtt{true}$, a comparison between two expressions $E_1 \bowtie E_2$ (e.g., $x + y \leq 2z$), a logical conjunction of two predicates $\Pi_1 \wedge \Pi_2$ or a negation of a predicate $\neg\Pi$.

Back to our example, the behaviour of the robot looking for help can be partially sketched by the following fragment:

$$(\!|\mathtt{'help'}, \mathtt{this}.\mathsf{pos}|\!)@\{\mathtt{distance}(\mathtt{this}.\mathsf{pos}, pos) < k \wedge \mathsf{battery} > 0.5\}$$

where the robot sends an `'help'` message along with its position to all robots within distance k and with at least 50% battery charge left. For simplicity, we use `distance` as a shorthand for a more complex expression that computes the distance between any two given positions. Note that the robot uses `this.pos` to indicate the local attribute which refers to its own position. A possible reaction of a robot at the other end could be:

$$\langle\neg\mathtt{this}.\mathsf{busy}\rangle(\mathsf{message} = \mathtt{'help'})(\mathsf{message}).\cdots +$$
$$\langle\mathtt{this}.\mathsf{busy}\rangle(\mathsf{message} = \mathtt{'help'})(\mathsf{message})$$

To further illustrate the effectiveness of AbC in describing non-trivial interaction patterns, we consider the stable marriage protocol (cf. Sec. 1) and its attribute-based implementation [13,14].

In AbC, we can model men and women as components whose attributes are the identifier id, the preference list prefs, and the current partner. A man updates his attribute partner to the first element of prefs, and then sends a `propose` message to components whose id attribute coincide with the partner attribute; he then waits for a `no` message to reset the partner, and so on:

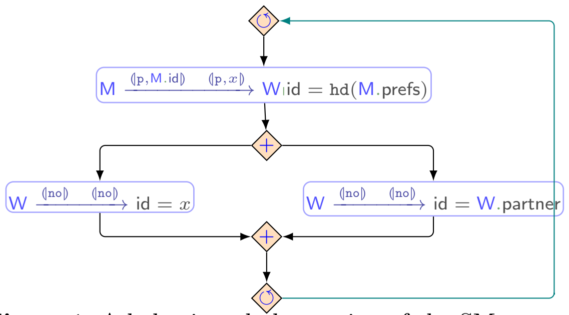
$$\begin{aligned}\mathrm{M} =&[\mathtt{this}.\mathsf{partner} := \mathtt{head}(\mathtt{this}.\mathsf{prefs}), \mathtt{this}.\mathsf{prefs} := \mathtt{tail}(\mathtt{this}.\mathsf{prefs})]\\ &(\mathtt{propose}, \mathtt{this}.\mathsf{id})@(\mathsf{id} = \mathtt{this}.\mathsf{partner}).\mathrm{Wait}\\ \mathrm{Wait} =&[\mathtt{this}.\mathsf{partner} := \mathtt{null}](no)(\_m).\mathrm{M}\end{aligned}$$

Women wait for incoming proposals (Handle process), and either prefer the proposer to her current partner (process A), or otherwise (process R). Note that immediately after receiving a proposal, women can handle other proposals. Notice that both R and A use a reversed form of preference lists to compare identifier of the current partner with one of a new proposer:

$$\begin{aligned}\mathrm{Handle} =&(\mathtt{propose})(\mathrm{Handle} \mid (\mathtt{propose}, \_id).(\mathrm{A}(id) + \mathrm{R}(id)))\\ \mathrm{A}(id) =&\langle\mathtt{this}.\mathsf{prefs}[\mathtt{this}.\mathsf{partner}] < \mathtt{this}.\mathsf{prefs}[id]\rangle\\ &[ex := \mathtt{this}.\mathsf{partner}, \mathtt{this}.\mathsf{partner} := id](no)@(\mathsf{id} = ex)\\ \mathrm{R}(id) =&\langle\mathtt{this}.\mathsf{prefs}[\mathtt{this}.\mathsf{partner}] > \mathtt{this}.\mathsf{prefs}[id]\rangle(\mathtt{no})@(\mathsf{id} = id)\end{aligned}$$

3 AbC-inspired Behavioural Types

We scribble a few abstractions for CAS that we dub AbC-*inspired behavioural types* (ABeT for short). Basically, ABeT mix together ideas from session types [23] in order to support the specification, verification, and analysis of CAS. In particular, ABeT blend *global choreographies* [31,22], and *klaimographies* [8]. Instead of appealing to formal syntax and semantics, we opt for an intuitive visual description. To this purpose, we use diagrams (see Fig. 1) depicting the communication pattern of the SM protocol. We now survey the core features of ABeT.



$\mathsf{M} \xrightarrow{(\!|\mathsf{p}, \mathsf{M.id}|\!) \quad (\!|\mathsf{p}, x|\!)} \mathsf{W} \mid \mathsf{id} = \mathtt{hd}(\mathsf{M.prefs})$
$\mathsf{W} \xrightarrow{(\!|\mathsf{no}|\!) \quad (\!|\mathsf{no}|\!)} \mathsf{id} = x$
$\mathsf{W} \xrightarrow{(\!|\mathsf{no}|\!) \quad (\!|\mathsf{no}|\!)} \mathsf{id} = \mathsf{W.partner}$

Figure 1. A behavioural abstraction of the SM protocol

The diagram in Fig. 1 represents a distributed workflow of *interactions*, which are the atomic elements of ABeT; the general form of an interaction for CAS is inspired by AbC and takes the form

$$\mathsf{A} \mid \rho \xrightarrow{e \quad e'} \mathsf{B} \mid \rho' \tag{1}$$

The intuitive meaning of the interaction in (1) is as follows:

> *any* agent, say A, *satisfying* ρ generates an expression e for *any* agents satisfying ρ', dubbed B, provided that expression e' *matches* e.

We anticipate that A and B here are used just for convenience and may be omitted for instance writing $\rho \xrightarrow{e \quad e'} \mathsf{B} \mid \rho'$ or $\rho \xrightarrow{e \quad e'} \rho'$. Also, we abbreviate $\mathsf{A} \mid \rho$ with A when ρ is a tautology. This allows us to be more succinct. For instance, the top-most interaction in Fig. 1 specifies that

- all "man" agents propose to the "woman" who is top in their preference list by generating a tuple made of the constant value p and their identifier M.id
- any "woman" agent (denoted as W in the rest of the diagram) whose identifier equals the head hd(M.prefs) of her preference list pattern matches the tuple sent by M and records in the variable x the identity of the proposer.

Interactions such as (1) introduce already some abstractions:

– Similarly to global choreographies, (1) abstracts away from asynchrony; interactions are supposed to be atomic
– As in klaimographies, (1) abstracts away from the number of agents in execution
– As behavioural types, (1) abstracts away from the local behaviour of the agents; for instance, it does not tell if/how the local state of agents are changed (if at all) by the interaction

Interactions are the basic elements of an algebra of protocols [31] that allows us to iterate them or compose them in non-deterministic choices or in parallel. In Fig. 1, the ⟐-gates mark the entry and exit points of the loop of the SM protocol. Likewise, the ⊕-gates mark the branching and merging points of a non-deterministic choice. The "body" of the loop in Fig. 1 yields the choice specified by the SM protocol. We will also use ◫-gates to mark forking and joining points of parallel protocols.

These abstractions are "compatible" with the pseudocode in Sec. 1. In fact, one could assign[4] B the communication pattern of M in Fig. 1 and to C the communication pattern of W. Besides, other "implementations" of the SM protocol could be given those types. For instance, M can also be the type of an agent that puts back in its list of preferences the identifier of a charger which denies the service (so to try again later).

4 Speculating on ABeT

We advocate new behavioural types for CAS. In the following we argue why "standard" behavioural types fall short of the requirements imposed by CAS.

Communication, roles & instances. As discussed in Sec. 1, point-to-point communications are not ideal for CAS. In particular, features such as attribute-based communications would be unfeasible with point-to-point interactions. Since the communication mechanism adopted in ABeT is inspired by AbC, it seems natural to get inspiration from klaimographies which, as far as we know, are the only behavioural abstractions that are not based on point-to-point communication. Klaimographies indeed are based on generative communication [9]. We envisage attribute-based communication as a further abstraction of generative communication. In fact, it would be interesting to look at ABeT as a mechanism to support the realisation of CAS by realising AbC specifications with distributed tuple spaces. We will return to this point in Sec. 7.

A basic abstraction borrowed from klaimographies is the notion of *role*. For instance, Fig. 1 describes the communication pattern of the "male" and "female" roles of SM protocol. Such roles are then enacted by *instances* of agents. The distinction between roles and instances is instrumental to abstract away from the number of agents, which is a paramount aspect of CAS. This is also a reason

[4] A suitable typing discipline, out of the scope of this paper, could do the assignment.

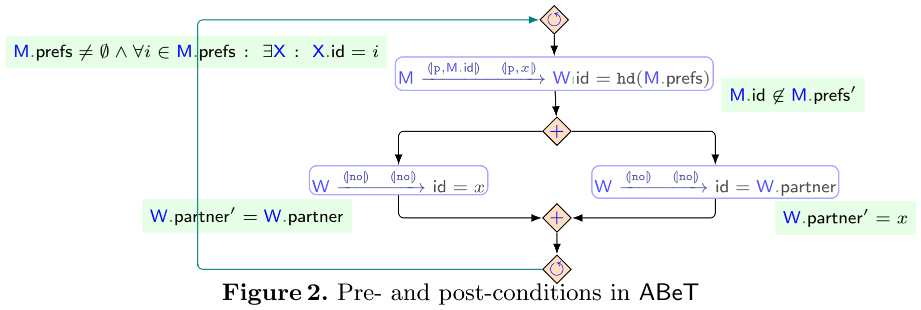

M.prefs ≠ ∅ ∧ ∀i ∈ M.prefs : ∃X : X.id = i
M (p, M.id) (p, x) W id = hd(M.prefs)
M.id ∉ M.prefs′
W (no) (no) id = x
W (no) (no) id = W.partner
W.partner′ = W.partner
W.partner′ = x

Figure 2. Pre- and post-conditions in ABeT

why we believe that standard behavioural type systems are not suitable to capture the complexity of CAS. For instance, *parameterised session types* [12] use indexed roles that can be rendered with a given number of instances. However, parameterised session types can only handle point-to-point communications.

Another peculiarity of CAS is that agents may play more than one role in the protocol. This will be indeed the case for the scenario considered in Sec. 5. We are not aware of any existing behavioural type framework allowing this.

Correctness. The agent R described at the end of Sec. 3 has a drastically different behaviour than agent B in Sec. 1. In fact, provided that preferences satisfy some properties, the latter has been shown in [20] to "stabilises" in a state where agents B are deadlocked waiting for a message from a charger C that is never sent and that each male agent pairs up with a female one and vice versa. The former CAS can instead diverge if e.g., the identifier of an agent B is not in the preference list of any charger agent.

The previous example suggests that ABeT do not guarantee progress. This is a distinctive difference from standard behavioural types, where the usual theorem is that well-typed programs are (dead)lock-free. Notice though, that progress (at least the strong variant usually taken in behavioural type theories) is not necessarily desirable in CAS. For instance, agents B *must* deadlock in order to successfully run the protocol. These observations immediately impose the following questions

– what notion of correctness suits CAS?
– what behavioural property do typed CAS enjoy?
– what features should ABeT have to capture such properties?

We argue that, to address these questions, it is not enough to capture the control-flow of CAS in behavioural abstractions. Basically, to some extent ABeT should also allow one to model the local behaviour of agents, at least to some extent. To illustrate this we use the variant of SM protocol in Fig. 2 where the top-most interaction is now decorated with some pre- and post-conditions. The idea is that these conditions constraint the local behaviour of agents. Intuitively,

– the pre-condition in Fig. 2 requires that, before sending a proposal, all the identifiers in the preference list of an agent M are assigned to some agent
– the post-condition in Fig. 2 states that, after being used for a proposal, the identifier is removed from the preference list of M.

Likewise, the post-conditions on the interactions of each branch state that the partner of W is set to the agent to whom the message `no` is not sent. Using pre- and post-conditions we can assign the decorated ABeT of Fig. 2 to agent `B` but not to agent `R`.

Asserted interactions have been proposed in [7] in order to specify relations among data exchanged by participants. We advocate the use of assertions also to specify local behaviour of CAS agents. For instance, the post-conditions on the branches in Fig. 2 is a mere assertion on the state of W and it does not involve any communication.

Progress & well-formedness. Related to correctness is the problem of identifying suitable notions of progress for CAS. The classical SM protocol is designed so that agents are actually suppose to deadlock. This is inconceivable in all the behavioural type systems but klaimographies, which brings us to a fundamental peculiarity of ABeT.

The typical result of behavioural type theories is the definition of *well-formedness* conditions that guarantee some form of progress. The idea is that well-formedness constraints are conditions on the choreography sufficient to ensure that the protocol deadlocks only if all participants terminates. For instance, all the known well-formed conditions of behavioural types would reject the SM protocol as ill-typed. In fact, the choice in the SM protocol is not propagated well according to the canons of behavioural types. This is due to the following reason. Behavioural types require that branches of a distributed choice to be "alternative", namely only one of them should be enacted by *all* the involved agents. Well-formedness conditions are usually imposed to guarantee that all the agents involved in a distributed choice unequivocally agree on which branch to follow. The rationale is to avoid mismatches in the communications when some participants follow a branch while other execute another branch of the choice. This is exactly what happens in the SM protocol. The choice involves the agent W, M, and W.partner. However, W communicates her decision only to one between M and W.partner.

Run-time support. An advantage of behavioural abstraction is that they allow us to *project* specifications to lower down levels of abstraction. For instance, one can obtain specification of each role by projecting the ABeT of Fig. 1 similarly to what done for klaimographies in [8]. Most behavioural types are amenable of being projected on executable code [21] as well as monitors to enforce some run-time properties [6,28,18]. These projections are "simple" since the communication model in the behavioural types is very close to existing ones (such as those of message-oriented middlewares or programming languages such as Erlang or GoLang). Attaining such projection operations for ABeT would not be that straightforward. Attribute-based communication requires some ingenuity to be effectively realised on the common communication infrastructures available for distributed systems.

We think that ABeT can support the run-time execution of CAS systems. The idea is to generate *ghost* components that decide how to spread the information among agents. The ghost of an agent can be statically derived by projection a global type in order to regulate the spread of the information across CAS. For instance, a static analysis could determine that some roles are not interested receiving some data, hence the ghosts of agents willing to synchronise on those patterns would spread the relevant information among themselves before sending it to any other agents. Similarly, the ghost of an agent that has to spread some information to other agents can prioritise the ones that are more likely to need that information in the near future than those that will not access it soon.

5 Autonomous Robots

To highlight other features of ABeT, in this section we consider a more complex scenario. Initially, we assume that agents can play two roles for simplicity but one can easily imagine that more roles could be involved.

5.1 A coordination protocol

Consider a decentralised system where robots roam a physical environment to delivery some goods. When a robot is in operational state, its battery depletes according to some factors (such as its load, speed, etc.). Therefore, robots collaborate to optimise the use of energy by *offering* and *requiring* energy. More precisely, a robot can offer other robots a recharging service when e.g., its energy is above a given threshold or seek for a recharge of its battery from other robots when its battery consumption becomes high. This way, energy is exchanged among robots, and it is less likely that robots cannot terminate their delivery tasks due to battery drain. This means that all the robots simultaneously assume the role of being consumers and suppliers of energy. The collective behaviour of robots is a strategy to accomplish their goal.

We now design a protocol that robots may use to administer their batteries. Let us assume that Sue and Rick are two robots of our delivery system enacting a specific role for our scenario. Namely, Sue behaves as energy supplier while Rick looks for a recharge, respectively. The robots repeatedly behave as follows.

Initially, Sue advertises her offer by sending to every potential consumer the available charge of the battery, and her contact details. She then waits for incoming answers from any interested robot. When Rick receives an offer of energy, he decides whether to make a request. For instance, this decision depends on the quantity of energy offered and the quantity of energy needed. If Rick is interested to acquire a certain amount of energy from Sue, it makes a request to Sue and sends his contact details. Rick then waits for Sue to confirm or to cancel the offer of energy; this is necessary because in the meantime Sue may have committed to supply some energy to other robots or she may have switched to a consumer modality depending on the level of her battery. Therefore, upon Rick's request, Sue decides whether the offer is still valid. If that is the case, Sue notifies Rick with the amount of energy she is willing to supply. Otherwise, Sue tells Rick that the offer is no longer available. For simplicity, we will assume that robots have identifiers that encompass all their contact details.

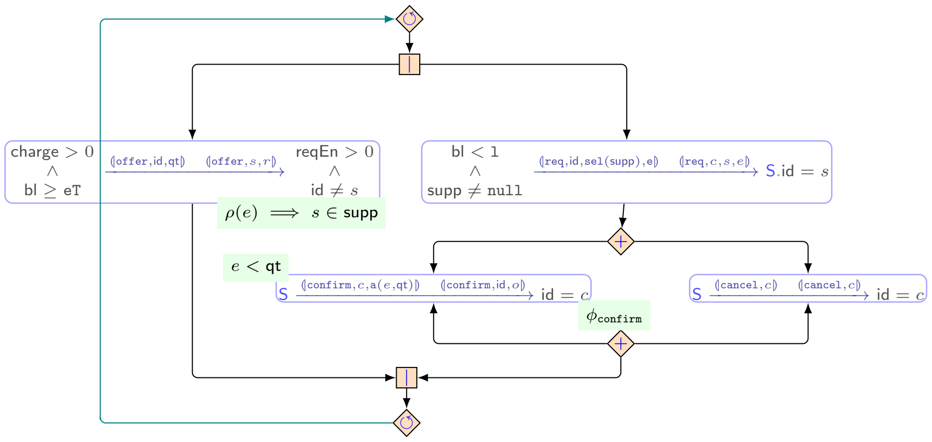


Figure 3. ABeT specification of robots

5.2 A specification in ABeT

The protocol in Fig. 3 captures the scenario in Sec. 5.1. The protocol is a loop in which robots manage energy through two parallel threads. In the left-most thread, energy offers are advertised by robots whose local state is such that they can offer a recharge ($\mathsf{charge} > 0$) and have a battery level above a fixed threshold ($\mathsf{bl} \geq \mathtt{eT}$). This advert is addressed to any robot requiring energy ($\mathsf{reqEn} > 0$). Those robots may not consider the offer if they deem it inconvenient; this is modelled by the conditional post-condition $\mathsf{c}(e) \implies s \in \mathsf{supp}$ where the proposition $\mathsf{c}(\_)$ establishes whether the energy e offered is “convenient”; only if this is the case the robot updates its local state adding the supplier’s identifier in its list of potential suppliers. Notice that the condition $\mathsf{id} \neq s$ avoids adding the consumer’s identity in supp; this avoids dealing with self-offers.

Once advertisements are out, a robot in need of energy can contact one of the suppliers chosen from those stored in its supp attribute ($\mathtt{sel}(\mathsf{supp})$). At this point the contacted supplier decides whether to confirm the offer (left-most branch of the choice) or cancel it. The former case requires that the amount of requested energy is lower than the supplier’s energy level (pre-condition $e < \mathsf{qt}$). In our specification the actual amount of offered energy is a function of the required energy and the available quantity ($\mathsf{a}(e, \mathsf{qt})$); in the simplest case this may just equal e, but one could think of a lower amount if the local state of the supplier had changed between the advertisement and the current request of energy. Upon confirmation of the offer, the consumer updates its local state accordingly; this is modelled with the post-condition $\phi_{\mathtt{confirm}}$ that reads as

$$\mathsf{S}.\mathsf{charge} = \mathsf{S}.\mathsf{qt} - o \qquad \wedge \qquad \mathsf{C}.\mathsf{bl} = \mathsf{C}.\mathsf{bl} + o \wedge \mathsf{C}.\mathsf{reqEn} = \mathsf{C}.\mathsf{reqEn} - o$$

namely, the supplier is supposed to subtract the offered energy from its quantity while the consumer has to add the same amount to its battery level and updates its reqEn attribute accordingly. This scenario offers the possibility of a few remarks, besides the general ones in Sec. 4.

The protocol describes the behaviour of robots in need of energy and, *at the same time*, their behaviour when offering energy. In other words, the same agent can play both roles and behave according to a "supplier-" or "consumer-modality". Crucially, the specification in Fig. 3 does not constraint *a-priori* when or how these modalities can be played. For instance, the following would be all valid implementations:

– some robots act purely as suppliers (e.g., to model charging stations) while others act purely as consumers (e.g., mobile robots);
– robots can behave according to either of the modalities at a time;
– robots can offer or require energy at the same time.

This variability is not possible in most existing settings based on behavioural types where single-threadness is necessary. Notice that the behaviours mentioned above may determine different roles, e.g., a recharging station pursues a different goal than robots and it follows different communication patterns.

Another observation is that the protocol can be configured to obtain different emerging behaviours by tuning up some parameters such as the thresholds, the function a used when confirming the offers, or the proposition c used when accepting offers. For instance, one could set the battery level so that a robot acting just as charging station continuously makes offers (e.g., $\mathsf{bl} = 0$). Likewise, the protocol is permissive on local computations. For example, the criteria used by a robot to select the supplier to contact are abstracted away by the function `sel(_)` which can be implemented in a myriad of different ways. To mention but a few options, consider solutions where

– the attribute supp just contains the latest received offer and `sel(_)` is just the identity function;
– `sel(_)` treats the attribute supp as a priority list;
– the robot may store information about the amount of energy offered by suppliers or its physical position and select the one closest to its needs.

In principle, one could be more specific on those aspects by tuning up pre- and post-conditions of interactions.

6 Quantitative Analysis

We now discuss how to derive a quantitative analysis for CAS using the ABeT specification of the robots case study of Sec. 5. Starting from the specification of the interaction protocol, we build a quantitative abstraction that provides some analysis of the system under study. To this end, in this paper we propose to adopt Queuing Network (QN) models [25] which are widely applied in the software performance engineering community [24,30,5].

A QN model is a collection of *service centres* each regulated by a *rate*. Intuitively, service centres represent resources shared by a set of *jobs*, also called *customers* [25]. Incoming workload can be modelled as: (i) a *closed* class (i.e., a constant number of customers in each class and a think time regulating the delay

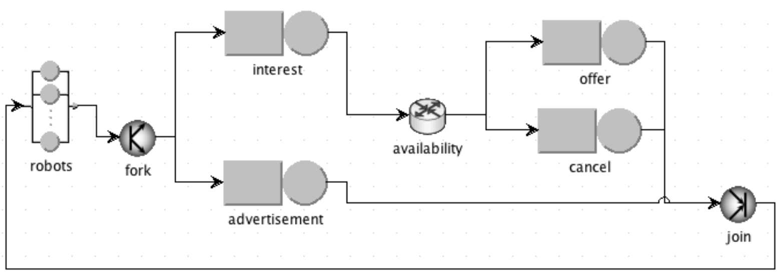


Figure 4. QN model for the Exchange of resources among robots.

of performing a system request), or (ii) an *open* class (i.e., the customer arrival rate). The former workload class is modelled through *delay* centres, whereas the latter required the specification of *source* and *sink* nodes for indicating the start and the completion, respectively, of requests. Delay centres, unlike service centres, do not have a queue. They are intended to model delays occurring when specifying closed workloads (i.e., requests re-iterate in the network in a loop-based fashion) or communication networks.

Service and delay centres are connected through *links* that form the network topology; a service centre consists of a *server* and a (possibly unbound) *queue*. Servers process *jobs* scheduled from their queues according to a specific policy (e.g., FIFO). Each processed request is then routed to another service centre through links. More precisely, each server picks the next job (if any) from its queue, processes it, and sends the result on its outgoing link; a link may go through a *router node* which forwards it to the queue of another service centre according to a given probability.

In order to capture parallelism, QN models feature *fork* and *join* nodes. The former nodes are used to split the jobs into several tasks to be executed in parallel while a join node represents the point where these tasks are synchronized. Summarizing, a QN model can be interpreted as a directed graph whose nodes are service centres and their connections are the graph edges. Jobs go through edges set on the basis of the behaviour of customers' requests.

Fig. 4 depicts the QN model obtained from the ABeT specification given in Sec. 5.2. More precisely, the topology of the model tightly correspond to the structure of specification given in Fig. 3 as follows:

- each interaction becomes a service centre;
- each parallel and join gate of the type in Fig. 3 respectively becomes a fork and a join node of the QN model;
- the non-deterministic choice becomes a router node;
- the delay centre can be used to model network delays in the communication among agents, however we decided to skip such modelling and to insert one delay centre only that is representative of the system closed workload (i.e., number of robots and their thinking time, if any).

The delay node namely *robots* represents the incoming requests, and for our case study we have three classes of jobs: suppliers, consumers, supp&cons, i.e., robots simultaneously acting as suppliers and consumers. In parallel, as showed by the

Parameter	Value
robots	wp=30
advertisement	$\lambda = 10$
interest	$\lambda = 10$
offer	$\lambda = 10$
cancel	$\lambda = 10$
availability	$\pi = 0.5$

Table 1. Setting of parameters

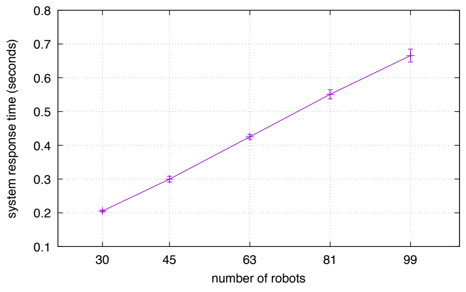


Figure 5. QN model results

fork node, robots can (i) put their offers, as modelled by the *advertisement* queuing centre, or (ii) verify if an offer matches their request, as modelled by the *interest* queuing centre. A router node, namely *availability* regulates the probability with which robots send an *offer* or a *cancel* message. In case of a confirmed offer, it is necessary to update the attributes, as modelled by the offer queueing centre. In case of cancel, it is also possible to specify a service time denoting the check of the corresponding message. All the requests outcoming from *offer*, *cancel*, and *advertisement* are routed to the *join* node modelling the completion of the iteration so that robots can re-iterate their behaviour.

To run a model-based performance analysis, we set the values for the parameters of the QN model. Table 1 shows an example of parameters' setting; the analysis is later performed by varying the parameter *robots*. The delay centre is associated with the specification of the closed workload. In particular, we set a workload population (wp) equal to 30 robots for the three classes of jobs (i.e., suppliers, consumers, supp&cons), thus considering 10 robots for each of the classes. The service centres are associated with parameters drawn from exponential distributions with average λ. For example, if the *advertisement* node is assigned $\lambda = 10$, the inter-arrival time drawn from the distribution averages one request every 0.1 time units (i.e., milliseconds in this case). Here we give a flavour of the type of analysis that can be performed in such scenarios. Note that the router node is associated with a π value denoting the probability associated to the branching points set with equal probabilities. Of course other values can be set to evaluate the behaviour under different assumptions. The obtained QN model has been analysed with the Java Modeling Tool (JMT) [11]. Fig. 5 presents simulation results for the timing analysis, varying the population of robots, on the x-axis, up to 99, i.e., denoting up to 33 robots of each type.

As expected, the system response time (reported on the y-axis) increases as the number of robots increases going from 0.21 to 0.67 seconds. For example, a population of 45 robots (i.e., 15 robots for each type), results in a mean response time of 0.299 seconds and maximal and minimal values estimated to be 0.308 and 0.291 seconds, respectively.

7 Conclusions, Related & Future Work

We outlined some ideas for behavioural abstractions of CAS in the context of the AbC calculus. By means of some examples we argued for a new class of behavioural specifications tailored on CAS. In the following, we review the state-of-the-art by focusing on behavioural and quantitative abstractions for CAS.

Behavioural Abstractions. In the past thirty years behavioural types [23] have been widely adopted to reason about and support the development of communicating systems. Extensive research yielded important results for distributed systems where channel-based communications are adopted. To the best of our knowledge, klaimographies [8] are the only attempt to develop behavioural types for generative communication [9], recently applied to event-notification systems [19]. This recent results inspired the ideas described in this paper. In fact, CAS coordination mechanisms are often reduced to some sorts of event-notification mechanisms. In fact, we believe that event-notification is probably the most suitable approach to implement languages used to specify CAS.

Linguistic mechanisms. A recent work on modelling the interactions of CAS is [2] where a language-based approach is introduced for building groups of communicating partners by considering their attributes. The same attribute-based communication (AbC) paradigm has been also used in our previous work [13,15] in which different forwarding strategies for message passing together with Erlang are exploited for dealing with scalability issues. The precursor of AbC is represented by SCEL [16] aimed to model the dynamic formation of ensembles of interacting autonomic components, and to verify some system properties (e.g., reachability). Programming actor-based CAS is discussed in [10] where global-level system specifications are translated into Scala/Akka actors with the goal of carrying coordination tasks that involve large sets of devices.

Quantitative Abstractions. The idea of proposing a quantitative evaluation of CAS finds its root in [32] where ordinary differential equations (ODEs) are adopted as trade-off between expressiveness and efficiency of the analysis. The modelling and quantitative verification of CAS is also proposed in [26] where a language (i.e., CARMA) aims to capture the timing and the probabilistic behaviour of agents. In [4] the focus is on verifying the robustness of CAS, against various classes of attacks, by embedding trust models of communication, i.e., reputation and trust relationships among the agents' exchanged information. Engineering resilient CAS is tackled in [33] with the goal of providing reusable blocks of distributed behaviour that can be substituted by functionally equivalent ones, possibly more efficient in terms of performance or showing desirable dynamics, i.e., without affecting the overall system behaviour and resilience. These works mostly focus on specific aspects, such as efficiency [32,33] dependability [26], and reputation [4]. As opposite to these approaches, we aim to narrowing the behavioural and quantitative abstractions within CAS, thus to quantify the impact of design choices acting on the communication among agents with their non-functional properties.

Future work. The obvious next step is the precise formalisation of behavioural type systems that capture the ideas sketched in the previous sections. A natural starting point is the elaboration of klaimographies. We plan to refine the types described here and identify relevant properties for CAS that could be statically checked. For instance, a system that allows us to check that AbC implementations type check against type specifications such as those seen for the scenario of Sec. 5. This would require the identification of suitable local types as well as the definition of suitable projection operations. We are also interested in translating the results of the behavioural type systems into insights for their quantitative analysis. As long term goal, we aim to bridge behavioural abstractions with their quantitative counterpart, thus to provide a deeper analysis of CAS.

References

1. Y. Abd Alrahman, R. De Nicola, and M. Loreti. A calculus for collective-adaptive systems and its behavioural theory. *Inf. Comput.*, 268, 2019.
2. Y. Abd Alrahman, R. De Nicola, and M. Loreti. Programming interactions in collective adaptive systems by relying on attribute-based communication. *Sci. Comput. Program.*, 192:102428, 2020.
3. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.
4. A. Aldini. Design and verification of trusted collective adaptive systems. *ACM Trans. Model. Comput. Simul.*, 28(2):1–27, 2018.
5. S. Balsamo and M. Marzolla. Performance evaluation of UML software architectures with multiclass queueing network models. In *Workshop on Software and Performance*, 2005.
6. L. Bocchi, T.-C. Chen, R. Demangeon, K. Honda, and N. Yoshida. Monitoring networks through multiparty session types. In *Formal Techniques for Distributed Systems*, volume 7892 of *LNCS*. Springer, 2013.
7. L. Bocchi, K. Honda, E. Tuosto, and N. Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *Int. Conf. on Concurrency Theory*, volume 6269 of *LNCS*. Springer, 2010.
8. R. Bruni, A. Corradini, F. Gadducci, H. C. Melgratti, U. Montanari, and E. Tuosto. Data-driven choreographies à la klaim. In *Models, Languages, and Tools for Concurrent and Distributed Programming*, volume 11665 of *LNCS*. Springer, 2019.
9. N. Carriero and D. Gelernter. Linda in context. *CACM*, 32(4):444–458, April 1989.
10. R. Casadei and M. Viroli. Programming actor-based collective adaptive systems. In *Programming with Actors*. Springer, 2018.
11. G. Casale and G. Serazzi. Quantitative system evaluation with java modeling tools. In *International Conference on Performance Engineering*, 2011.
12. D. Castro, R. Hu, S. Jongmans, N. Ng, and N. Yoshida. Distributed programming using role-parametric session types in go: Statically-typed endpoint APIs for dynamically-instantiated communication structures. *PACMPL*, 3(POPL):1–30, 2019.
13. R. De Nicola, T. Duong, O. Inverso, and C. Trubiani. AErlang at Work. In *Int. Conf. on Current Trends in Theory and Practice of Informatics*. Springer, 2017.
14. R. De Nicola, T. Duong, O. Inverso, and C. Trubiani. AErlang: Empowering Erlang with Attribute-Based Communication. In *Int. Conf. of Coordination Models and Languages*, 2017.

15. R. De Nicola, T. Duong, O. Inverso, and C. Trubiani. Aerlang: Empowering erlang with attribute-based communication. *Sci. Comput. Program.*, 168:71–93, 2018.
16. R. De Nicola, M. Loreti, R. Pugliese, and F. Tiezzi. A formal approach to autonomic systems programming: The SCEL language. *ACM Transactions on Autonomous and Adaptive Systems*, 9(2):1–29, 2014.
17. A. Ferscha. Collective adaptive systems. In *Int. Joint Conf. on Pervasive and Ubiquitous Computing and Symposium on Wearable Computers*, 2015.
18. A. Francalanza, C. A. Mezzina, and E. Tuosto. Reversible Choreographies via Monitoring in Erlang. In *Distributed Applications and Interoperable Systems*, volume 10853 of *LNCS*. Springer, 2018.
19. L. Frittelli, F. Maldonado, C. Melgratti, and E. Tuosto. A Choreography-Driven Approach to APIs: the OpenDXL Case Study. In *Int. Conf. of Coordination Models and Languages*, volume 12134 of *LNCS*. Springer, June 2020.
20. D. Gale and L. S. Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 68(1):9 – 15, 1962.
21. S. Gay and A. Ravara, editors. *Behavioural Types: from Theory to Tools*. Automation, Control and Robotics. River, 2009.
22. R. Guanciale and E. Tuosto. An abstract semantics of the global view of choreographies. In *Interaction and Concurrency Experience*, 2016.
23. H. Hüttel, I. Lanese, V. T. Vasconcelos, L. Caires, M. Carbone, P. Deniélou, D. Mostrous, L. Padovani, A. Ravara, E. Tuosto, H. T. Vieira, and G. Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):1–36, 2016.
24. L. Kleinrock. *Theory, Volume 1, Queueing Systems*. Wiley-Interscience, 1975.
25. E. D. Lazowska, J. Zahorjan, G. Scott Graham, and K. C. Sevcik. *Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Inc., Englewood Cliffs, 1984.
26. M. Loreti and J. Hillston. Modelling and analysis of collective adaptive systems with CARMA and its tools. In *Int. School on Formal Methods for the Design of Computer, Communication and Software Systems*, 2016.
27. R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, I and II. *Inf. Comput.*, 100(1):1–40, 1992.
28. R. Neykova, L. Bocchi, and N. Yoshida. Timed runtime monitoring for multiparty conversations. *Formal Aspects of Computing*, 29(5):877–910, 2017.
29. K. V. S. Prasad. A calculus of broadcasting systems. *Sci. Comput. Program.*, 25(2-3):285–327, 1995.
30. C. U. Smith and L. G. Williams. Performance and scalability of distributed software architectures: An SPE approach. *Scalable Computing: Practice and Experience*, 3(4), 2000.
31. E. Tuosto and R. Guanciale. Semantics of global view of choreographies. *J. Log. Algebr. Meth. Program.*, 95:17–40, 2018.
32. A. Vandin and M. Tribastone. Quantitative abstractions for collective adaptive systems. In *Int. School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM)*, 2016.
33. M. Viroli, G. Audrito, J. Beal, F. Damiani, and D. Pianini. Engineering resilient collective adaptive systems by self-stabilisation. *ACM Trans. Model. Comput. Simul.*, 28(2):1–28, 2018.