Team Automata@Work: On Safe Communication

Maurice H. ter Beek[1(✉)], Rolf Hennicker[2], and Jetty Kleijn[3]

[1] ISTI–CNR, Pisa, Italy
maurice.terbeek@isti.cnr.it
[2] Ludwig-Maximilians-Universität München, Munich, Germany
hennicker@ifi.lmu.de
[3] LIACS, Leiden University, Leiden, The Netherlands
h.c.m.kleijn@liacs.leidenuniv.nl

Abstract. We study requirements for safe communication in systems of reactive components in which components communicate via synchronised execution of common actions. These systems are modelled in the framework of team automata in which any number of components can participate—as a sender or as a receiver—in the execution of a communication action. Moreover, there is no fixed synchronisation policy as these policies in general depend on the application. In this short paper, we reconsider the concept of safe communication in terms of reception and responsiveness requirements, originally defined for synchronisation policies determined by a synchronisation type. Illustrated by a motivating example, we propose three extensions. First, compliance, i.e. satisfaction of communication requirements, does not have to be immediate. Second, the synchronisation type (and hence the communication requirements) no longer has to be uniform, but can be specified per action. Third, we introduce final states to be able to distinguish between possible and guaranteed executions of actions.

1 Introduction

For the correct functioning of systems built from reactive components which collaborate by message exchange, it is important to exclude communication failures during execution, like message loss or indefinite waiting for input. This requires a thorough understanding of their synchronisation policies [5,8,11,17,18] to establish compatibility of communicating components [1,3,4,9,10,12,15]. Compatibility in multi-component systems was studied in [12] for services and in [10] for team automata, in both cases with the assumption that systems are full synchronous products of their components. Thus global states are Cartesian products of local states and all system transitions that represent the execution of an action leading from one global state to a next global state, involve all and only those component automata that have that action. A main reason to focus first on this kind of systems is that synchronous product automata are known for

Published by Springer Nature Switzerland AG 2020
S. Bliudze and L. Bocchi (Eds.): COORDINATION 2020, LNCS 12134, pp. 77–85, 2020.
https://doi.org/10.1007/978-3-030-50029-0_5

their appealing compositionality and modularity properties [6,7,14,16,18] and are thus easier to analyse.

Team automata, introduced in [2,5,13], represent a useful model to specify different forms of intended behaviour of reactive systems and they were shown to form a suitable formal framework for lifting the concept of compatibility to a multi-component setting. Explorations on generalising compatibility notions from full synchronous products to arbitrary synchronisation policies in the framework of team automata can be found in [3,4].

In [3], synchronisation types are used to classify synchronisation policies that can be realised in team automata. A synchronisation type is a pair (snd, rcv) that specifies the ranges for the number of senders and the number of receivers taking part in the transitions (communications) of the team automaton. A synchronisation type thus defines the transitions of the team automaton (its synchronisation policy). On the other hand, if at a given global state an appropriate number of components are ready to send (receive) an action, there is the requirement of synchronisation with a suitable number of other components that will receive (send, respectively) that action. Thus for output actions, requirements for reception at a given global state can be formulated. Conversely, locally enabled input actions give rise to responsiveness requirements. In [3], we have introduced a formal notation for expressing communication requirements and we have shown how such requirements can automatically be generated from a given synchronisation type. A team automaton is said to be compliant with a given set of communication requirements if in each reachable state of the team the requirements are met (the communication is safe).

Contribution. In this paper, we discuss, by means of an informal example, situations that can be seen as an impediment to this approach, in the sense that the application of the communication requirements appears to be too restrictive. As a solution, we propose the following three extensions to the idea of safe communication.

Compliance: the notion of compliance is made less restrictive by allowing *intermediate transitions by other components* before a particular communication requirement is fulfilled.

Actions: we propose an individual assignment of synchronisation types to communication actions to fine tune the number of participating sending and receiving components *per action*.

States: it may be the case that (local) enabledness of an action indicates only readiness for communication and not so much that communication is required; to make this distinction between possible and required communication explicit, we propose to add *final states* to components.

Outline. The paper starts with a brief summary of the principal notions of team automata, followed by a discussion of communication safety and compliance that is illustrated by an example from [3]. We point out some issues not covered by the

original definition of communication requirements, based on which we formulate extensions to make compliance a more liberal concept still following our intuition.

2 Team Automata

In the team automata framework, a *system* $\mathcal{S} = \{ \mathcal{A}_i \mid i \in \{1, \ldots, n\}\}$ consists of a (finite) set of reactive components modelled by component automata $\mathcal{A}_i$. Each component automaton has its own set of—local—states (with distinguished initial states), an alphabet of actions that are either input or output (not both)[1] to that component, and a labelled transition relation. The alphabet of actions of $\mathcal{S}$ consists of all actions of the $\mathcal{A}_i$. An action is called *communicating* (in $\mathcal{S}$) if it occurs in some automata of $\mathcal{S}$ as an output action and in some (other) automata of $\mathcal{S}$ as an input action. The state space of $\mathcal{S}$ is the Cartesian product of the state spaces of all $\mathcal{A}_i$, i.e. global states are tuples $\overline{q} = (q_1, \ldots, q_n)$ with local states q_i. The initial states of $\mathcal{S}$ are those global states that have only initial states as their local states. The possible transitions from one global state to another are described by labelled *system transitions*. The label of a system transition from $\overline{q}$ to $\overline{p}$ is an action a from the alphabet of $\mathcal{S}$ such that, for all i, whenever $q_i \neq p_i$, component $\mathcal{A}_i$ has an a-labelled transition from q_i to p_i. Thus any number of components in which a is *locally enabled* at q_i can participate simultaneously in a system transition from $\overline{q}$. An a-labelled transition in which both a component of which a is an output action (a *sender*) and one which has a as an input action (a *receiver*) participate, is a *communication* (via a).

One of the strengths of the team automata approach is that no a priori restrictions are imposed on system transitions. In general, it depends on the application which transitions from the set of all possible system transitions are relevant. Formally, a *synchronisation policy* is a subset δ of the system transitions of $\mathcal{S}$. Such policy δ determines a *team automaton* $\mathcal{T}$ over $\mathcal{S}$ which has as its state space the set of all global states of $\mathcal{S}$ that are reachable by δ from the initial states of $\mathcal{S}$.

In [3], *synchronisation types* are proposed to specify synchronisation policies for team automata. A synchronisation type (snd, rcv) determines ranges for the number of senders and the number of receivers that may take part in communications. For instance, if $snd = [k, l]$ (with $0 \leq k \leq l$) and $rcv = [m, n]$ (with $0 \leq m \leq n$) then at least k and at most l senders and at least m and at most n receivers are allowed. The synchronisation policy δ generated by (snd, rcv), consists of all system transitions that satisfy this constraint. While k, m are always natural numbers, the delimiters l, n can also be given as $*$ which indicates that no upper limit is imposed. Important synchronisation types are, eg., $([1, 1], [1, 1])$ which expresses binary communication, and $([1, 1], [1, *])$ for multicast communication in which exactly one component outputs a communicating action while arbitrarily many (but at least one) components input that action.[2]

[1] For simplicity of presentation, we do not consider internal actions here.

[2] In [3], we have also introduced notations for (strong and weak) broadcast communication and for full synchronisation, amongst others, which are not used here.

3 On Safe Communication

The idea underlying a communication-safe team automaton is that, in every (reachable) global state, whenever a communicating action is enabled (according to the prevailing synchronisation policy) at some of the local states of its components, these components can execute this action from these local states as a communication of the team.

As an example, let us consider a team automaton $\mathcal{T}$ with synchronisation type $([1,1],[1,*])$. Then, to guarantee that at a global state $\overline{q} = (q_1, \ldots, q_n)$, output action a of component $\mathcal{A}_i$, which is locally enabled at q_i, can be received by at least one other component, one would impose a *receptiveness requirement*, written as $\texttt{rcp}(i,a)@\overline{q}$. If $\mathcal{T}$ is compliant with (satisfies) this requirement, it is guaranteed that a can be executed by $\mathcal{T}$ at $\overline{q}$. Note that in case $\mathcal{A}_i$ could also execute another output action b at state q_i, also subject to the receptiveness requirement, the two requirements would be combined through a conjunction, denoted by $\texttt{rcp}((i,a) \wedge (i,b))@\overline{q}$. The reason for this is that components control their output actions and execution of either of them should lead to a reception.

For input actions one could require responsiveness with the intuition that enabled inputs should be served by appropriate outputs. Unlike output actions, however, input actions are controlled by the environment. Guaranteeing that for a choice of enabled inputs, one of them is provided by an output of the environment suffices for the progress of a component waiting for a signal from its environment. Hence, if component $\mathcal{A}_j$ enables input actions a and b in its local state q_j, then the *responsiveness requirements*, denoted by $\texttt{rsp}(j,a)@\overline{q}$ and $\texttt{rsp}(j,b)@\overline{q}$, respectively, would be combined with a disjunction as $\texttt{rsp}((j,a) \vee (j,b))@\overline{q}$.

In general, a team automaton $\mathcal{T}$ over a system $\mathcal{S}$ is called *communication-safe* if it is compliant with all communication requirements (at all states of $\mathcal{T}$) derived from its synchronisation type. We refer to [3] for the formal definition of compliance and the general procedure for deriving communication requirements.

Motivating Example
Consider a distributed chat system where buddies can interact once registered. There are three types of components: clients, servers, and arbiters; see Fig. 1, where input actions are annotated by ? and output actions by !. Initial states are marked with an incoming arrowhead. A server controls new entries into the chat as well as exits from the chat (actions *join* and *leave*, respectively). It also coordinates the main activities in the chat. The overall messaging protocol assumes registered clients to communicate messages to servers (action *msg*) which, upon arbiter approval (action *grant*), send the received messages to clients in the chat (action *fwdmsg*).

The chat system $\mathcal{S}$ considered here consists of two clients, one server, and one arbiter. The team automaton $\mathcal{T}_{chat}$ over $\mathcal{S}$ is defined by the synchronisation type $([1,1],[1,*])$, as the reception of forwarded messages may involve more than one component. Also in [3], this synchronisation type was applied. The states of $\mathcal{T}_{chat}$ are tuples (q_1, q_2, q_3, q_4) in which the first and second entries are client states, the third entry is a server state, and the fourth state is an arbiter state.

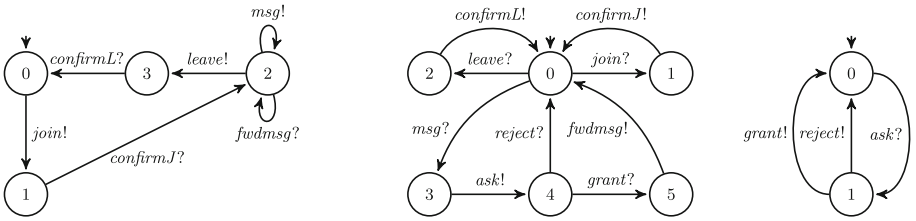

Fig. 1. [from left to right] Component automata for clients, servers, arbiters

An example of a receptiveness requirement, for all (reachable) states of $\mathcal{T}_{chat}$ with the server being in state 5, would be the following:

$$\texttt{rcp}(\mathcal{S}erver, fwdmsg)@(q_1, q_2, 5, q_4)$$

$\mathcal{T}_{chat}$ is compliant with this requirement since one of the clients locally enables *fwdmsg*. This is because, whenever the server is in state 5, at least one of the clients is in its state 2 (and the arbiter must be in state 0).

An example of a responsiveness requirement would be the following:

$$\texttt{rsp}((\mathcal{S}erver, join) \vee (\mathcal{S}erver, leave) \vee (\mathcal{S}erver, msg))@(q_1, q_2, 0, q_4)$$

$\mathcal{T}_{chat}$ is compliant with this requirement too, since whenever the server is in state 0 then each of the clients is in state 0 or in state 2. Hence there is always a client in a state that enables one of the required outputs for communication with the server (*join* in state 0 and *msg* or *leave* in state 2).

Extending Communication Safety

Using this example to illustrate our motivations, we will now introduce three useful extensions of the concept of communication-safety discussed so far.

Compliance: Intermediate Communications. First consider state $(2, 0, 5, 0)$ of $\mathcal{T}_{chat}$, where the second client locally enables the execution of its output action *join*. According to the synchronisation type that defines $\mathcal{T}_{chat}$, the output action *join* can be executed from local state 0 with the receptiveness requirement at this state being as follows:

$$\texttt{rcp}(\mathcal{C}lient_2, join)@(2, 0, 5, 0)$$

$\mathcal{T}_{chat}$ is not compliant with this requirement. Output action *join* of the second client has to be received as input by at least one of the other components. The only component with *join* as an input action is the server, but *join* is not enabled at its local state 5. The server can however transit from state 5 to state 0 (by a communication with the first client) after which it is ready to execute *join* in a communication with the second client. Hence, we propose a generalisation of the compliance notion along the following lines:

Given a receptiveness requirement for a component $\mathcal{A}_i$ and the actions $a_1, \ldots, a_k$ at a reachable state $\overline{q} = (q_1, \ldots, q_n)$, there should be a state $\overline{p}$ reachable from $\overline{q}$ by a sequence of zero or more team transitions in which $\mathcal{A}_i$ does not participate, and then, from $\overline{p}$, each of the actions $a_1, \ldots, a_k$ can be executed by $\mathcal{A}_i$ in a team transition.

A similar loosening of responsiveness requirements can be formulated (now requiring that at least one of the actions $a_1, \ldots, a_n$ can be executed by $\mathcal{A}_i$ in a team transition from $\overline{p}$).

Actions: Individual Synchronisation Types. Next, consider again the state $(2, 0, 5, 0)$ of $\mathcal{T}_{chat}$, but now with the following receptiveness requirement:

$$\texttt{rcp}((\mathcal{C}lient_1, leave) \wedge (\mathcal{C}lient_1, msg))@(2, 0, 5, 0)$$

This requirement expresses that the first client in state 2 can (internally) decide to execute either its output action *leave* or its output action *msg*, and for each, there should be at least one other component be ready (possibly after some team transitions not involving the first client, as discussed above) to execute this action as an input action. The server, that is in state 5, has only its output action *fwdmsg* locally enabled. Hence, by the synchronisation type $([1, 1], [1, *])$ of $\mathcal{T}_{chat}$, this requires a communication with a client. That client has to be the second client, which however currently is in state 0 with only output action *join* locally enabled. Consequently, the team automaton does not satisfy the receptiveness requirement of the first client at $(2, 0, 5, 0)$.

If, instead, we would have $([1, 1], [0, *])$ as a synchronisation type for the chat system, then the server would be allowed to move to state 0 by executing its output action *fwdmsg* on its own (rather than in a communication) after which the server would be ready to accept inputs as required. Thus it would be allowed that not every occurrence of *fwdmsg* will be received. However, $([1, 1], [0, *])$ is not an acceptable synchronisation type for other actions (like *join*, *leave*, etc). Indeed, a client performing a *join* action without acceptance by the server should not be permitted. Therefore, we propose to no longer require a uniform synchronisation type for all actions of the system, but rather to assign synchronisation types individually for each single action. In our example, this leads to the following action synchronisation types:

$$\texttt{stype}(join) = \ldots = \texttt{stype}(reject) = ([1, 1], [1, 1])$$
$$\texttt{stype}(fwdmsg) = ([1, 1], [0, *])$$

With this assignment we would also solve another issue with a chatting system which was also mentioned in [3]: Assume that, in order to increase robustness, we were to extend the system and let it consist of two servers and, as before, two clients and the arbiter. In case we would use the synchronisation type $([1, 1], [0, *])$ (or $([1, 1], [1, *])$) for the whole system, a client may send a message to two servers, who both forward the message (upon approval from the arbiter). The assignment of synchronisation types per action would solve

the problem of duplicate message forwarding, because we can now assign to the action *msg* the synchronisation type $([1,1],[1,1])$.

Hence, as exemplified above, the idea is to introduce the syntactic concept of a *synchronisation type specification*. Such a specification is a mapping $\mathtt{stype}$, which assigns to each communicating action a of the system a synchronisation type $\mathtt{stype}(a) = (snd, rcv)$ that determines ranges for the number of senders and receivers that may take part in a synchronisation (communication) on the action a. Each synchronisation type specification $\mathtt{stype}$ over a system $\mathcal{S}$ generates a unique team automaton $\mathcal{T}(\mathtt{stype})$ over $\mathcal{S}$ with a synchronisation policy that comprises all system transitions that—if labelled by a communicating action a—satisfy the synchronisation type $\mathtt{stype}(a)$. It remains to establish what this allows us to say about the communication safety of $\mathcal{T}(\mathtt{stype})$. Communication safety concerns receptiveness and responsiveness. Therefore, the systematic derivation of receptiveness and responsiveness requirements for a team automaton from a given uniform synchronisation type as developed in [3] has to be generalised by deriving receptiveness and responsiveness requirements individually per action.

States: Final States. Finally, assume that the behaviour of a client terminates after leaving the chat. In that case, input action *confirmL* would lead from state 3 to a new state 4. When all clients have terminated, the following responsiveness requirement of the server (in its state 0) could not be satisfied anymore:

$$\mathtt{rsp}((\mathcal{S}erver, join) \vee (\mathcal{S}erver, leave) \vee (\mathcal{S}erver, msg))@(4,4,0,0)$$

This is, however, not a problem if the input actions *join*, *leave*, and *msg* are seen as no more than services offered by the server: whether or not these services are called is irrelevant, clients are free to use or not to use a service. On the other hand, in its local state 4 the server definitely wants to get a response, *reject* or *grant*, from the arbiter. Hence, we need formal means to discriminate the quality of the two server states 0 and 4. Our idea is to declare some states of a component automaton as final states. Similarly to automata theory, where final states are accepting states which, nevertheless, may have outgoing transitions, in our framework a final state would be a state where execution can stop but may also continue.

In the example, state 0 of the server would be declared as final with the consequence that the server is no longer required to continue, i.e. there is no responsiveness requirement that one of its inputs *join*, *leave*, or *msg* must be served. (Still, the server offers these actions as input and thus can satisfy reception requirements from clients.) On the other hand, state 4 of the server should not be final because the server intends to proceed from this state. It is expecting a response from the arbiter which can be formalised, e.g., by the following responsiveness requirement:

$$\mathtt{rsp}((\mathcal{S}erver, reject) \vee (\mathcal{S}erver, grant))@(2,0,4,1)$$

This requirement is indeed fulfilled.

Of course, also the symmetric case has to be considered, i.e. what does the combination of final and non-final states with outputs mean. As an example, consider state 1 of the arbiter with the two outgoing transitions for the output of *grant* and *reject*, respectively. If this state were a final state, then this would mean that the arbiter may internally decide to stop here. Then the above responsiveness requirement of the server would not be satisfied anymore. Therefore, this state should definitely be a non-final state. Now consider state 0 of a client. If this were a final state, then a client might decide to never join a chat. This is not a problem if the server is not expecting any client to join, i.e. if the server's state 0 is declared to be final as discussed above.

Outlook. In summary, the addition of final states to component automata has significant consequences for the derivation of communication requirements and for our compliance notions, which must be adjusted accordingly. This is an important next step of our work. Another issue concerns the modelling of open systems and the composition of open team automata. We are specifically interested in investigating conditions under which communication safety of team automata is preserved by composition. This should eventually lead to a methodology for the modelling and analysis of large distributed systems with a significant communication behaviour.

Acknowledgements. The work of the first author was partially supported by the MIUR PRIN 2017FTXR7S project IT MaTTerS (Methods and Tools for Trustworthy Smart Systems). We thank the reviewers for their useful comments.

References

1. Bauer, S.S., Mayer, P., Schroeder, A., Hennicker, R.: On weak modal compatibility, refinement, and the MIO workbench. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 175–189. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_15
2. ter Beek, M.H.: Team automata: a formal approach to the modeling of collaboration between system components. Ph.D. thesis, Leiden University (2003). http://hdl.handle.net/1887/29570
3. ter Beek, M.H., Carmona, J., Hennicker, R., Kleijn, J.: Communication requirements for team automata. In: Jacquet, J.-M., Massink, M. (eds.) COORDINATION 2017. LNCS, vol. 10319, pp. 256–277. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59746-1_14
4. ter Beek, M.H., Carmona, J., Kleijn, J.: Conditions for compatibility of components. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9952, pp. 784–805. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47166-2_55
5. ter Beek, M.H., Ellis, C.A., Kleijn, J., Rozenberg, G.: Synchronizations in team automata for groupware systems. Comput. Sup. Coop. Work **12**(1), 21–69 (2003). https://doi.org/10.1023/A:1022407907596
6. ter Beek, M.H., Kleijn, J.: Team automata satisfying compositionality. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 381–400. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45236-2_22

7. ter Beek, M.H., Kleijn, J.: Modularity for teams of I/O automata. Inf. Process. Lett. **95**(5), 487–495 (2005). https://doi.org/10.1016/j.ipl.2005.05.012
8. Brim, L., Cerná, I., Vareková, P., Zimmerova, B.: Component-interaction automata as a verification-oriented component-based system specification. ACM Softw. Eng. Notes **31**(2) (2006). https://doi.org/10.1145/1118537.1123063
9. Carmona, J., Cortadella, J.: Input/Output compatibility of reactive systems. In: Aagaard, M.D., O'Leary, J.W. (eds.) FMCAD 2002. LNCS, vol. 2517, pp. 360–377. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-36126-X_22
10. Carmona, J., Kleijn, J.: Compatibility in a multi-component environment. Theor. Comput. Sci. **484**, 1–15 (2013). https://doi.org/10.1016/j.tcs.2013.03.006
11. de Alfaro, L., Henzinger, T.A.: Interface automata. In: ESEC/FSE 2001, pp. 109–120. ACM (2001). https://doi.org/10.1145/503209.503226
12. Durán, F., Ouederni, M., Salaün, G.: A generic framework for n-protocol compatibility checking. Sci. Comput. Program. **77**(7–8), 870–886 (2012). https://doi.org/10.1016/j.scico.2011.03.009
13. Ellis, C.A.: Team automata for groupware systems. In: GROUP 1997, pp. 415–424. ACM (1997). https://doi.org/10.1145/266838.267363
14. Gössler, G., Sifakis, J.: Composition for component-based modeling. Sci. Comput. Program. **55**, 161–183 (2005). https://doi.org/10.1016/j.scico.2004.05.014
15. Hennicker, R., Bidoit, M.: Compatibility properties of synchronously and asynchronously communicating components. Log. Methods Comput. Sci. **14**(1), 1–31 (2018). https://doi.org/10.23638/LMCS-14(1:1)2018
16. Jonsson, B.: Compositional specification and verification of distributed systems. ACM Trans. Program. Lang. Syst. **16**(2), 259–303 (1994). https://doi.org/10.1145/174662.174665
17. Larsen, K.G., Nyman, U., Wąsowski, A.: Modal I/O automata for interface and product line theories. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 64–79. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71316-6_6
18. Lynch, N.A., Tuttle, M.R.: An introduction to input/output automata. CWI Q. **2**(3), 219–246 (1989). https://ir.cwi.nl/pub/18164