Message Passing Automata: Investigation on the Growth in Complexity

Christiano H. Rezende,Gustavo S. Viana *Member, IEEE*

***Abstract*—In this paper, we implement two configurations of message passing automata: the asynchronous, that the controller has no need to wait for the controlled machines to execute its independent tasks, but the later have to wait the controller signal to start executing their tasks; and the synchronous, where the controller waits for all machines tasks being started to execute their own. Based on the increase of machines in the model, and comparing with experimental results, we investigate the growth in complexity, regarding the number of states, and conclude that the synchronous configuration has less states overall, although the same complexity of the asynchronous, however the scaling of the asynchronous model is easier.**

***Index Terms*—Message Passing, Growth in Complexity, Automata, Discrete Event Systems, Supervisory Control.**

I. Introduction

MESSAGE passing is a technique for invoking behavior on a computer. The invoking program sends a message to a actor that, after receiving the message, execute some task. In automata theory, a message passing automaton is an structure where one controller enables the execution of a sequence of events in another automaton. An implementation of this structure was explored in [1] and works as a guideline for this paper.

The objective of this study is implement a message passing automaton with asynchronous and synchronous configuration, as presented in [1], and to investigate how the increase of machines in the two configurations impacts the growth in complexity regarding the number of states of the automata.

The remainder of the paper is organized as follows. In Section II, we review some basic concepts on DES. In Section III, we show the problem of message passing, presenting the asynchronous configuration and analyzing the growth in complexity regarding the number of states and the machines. In Section IV, we explore the synchronous composition, describing how is built and analyzing the growth in complexity in this configuration. At least, in Section V, we remarks some conclusions and outline some future researches in this theme.

II. Background preliminaries

A. Discrete event systems

Discrete Event Systems (DES) are dynamic systems with discrete state space whose state evolution is entirely determined by the occurrence of asynchronous events over time [2]. In this paper, we adopt automata as the model formalism to describe DES behavior, and, to this end, we present now a brief review of automaton theory. Readers not familiar with automaton theory are referred to [2] for further details.

Let $G = (X, \Sigma, f, \Gamma, x_0, X_m)$ denote a deterministic finite automaton that models a DES, where X is the finite set of states, Σ is the finite set of events, $f : X \times \Sigma \to X$ is the transition function, partially defined over its domain, function $\Gamma : X \to 2^{\Sigma}$ (where 2^{Σ} denotes the power set of Σ) determines the set of active events of the states of G, *i.e.*, $\Gamma(x) = \{\sigma \in \Sigma : f(x, \sigma) \text{ is defined}\}$, x_0 is the initial state, and X_m is the set of marked states. Throughout the text, Σ^* denotes the Kleene closure of Σ, which is the set of all finite strings formed by events in Σ including the empty string, denoted by ε. The transition function f is extended to $f : X \times \Sigma^* \to X$ in the following manner: $f(x, \varepsilon) = x$ and $f(x, s\sigma) = f(f(x, s), \sigma)$, for all $x \in X$, $s \in \Sigma^*$ and $\sigma \in \Sigma$. The languages generated and marked by G are, respectively, $L(G) = \{s \in \Sigma^* : f(x_0, s) \text{ is defined}\}$ and $L_m(G) = \{s \in L(G) : f(x_0, s) \in X_m\}$. The prefix-closure of a language L is defined as $\overline{L} = \{s \in \Sigma^* : (\exists t \in \Sigma^*)[st \in L]\}$, and, if $L = \overline{L}$, L is said to be prefix-closed.

Let G_1 and G_2 denote two finite state automata. The union between G_1 and G_2 produces an automaton whose generated language is $L_m(G) = L_m(G_1) \cup L_m(G_2)$. Being $L_m(G_1)$ and $L_m(G_2)$ regular languages, $L_m(G)$ can be obtained by creating a new initial state and connecting it to the initial states of G_1 and G_2 thought ε-transitions. The result is a nondeterministic automaton marking the union between $L_m(G_1)$ and $L_m(G_2)$.

Let Σ_s and Σ_l be two sets of events such that $\Sigma_s \subseteq \Sigma_l$. The natural projection [3] from Σ_l^* to Σ_s^* is a function that removes from strings in Σ_l^* those events that are not in Σ_s, and is denoted as $P : \Sigma_l^* \to \Sigma_s^*$. The inverse projection P^{-1} is the function that recovers those strings in Σ_l^* that generate a given projection in Σ_s^*, being formally defined as $P^{-1}(t) = \{s \in \Sigma_l^* : P(s) = t\}$. The projection and the inverse projection operations are both extended to languages by applying P and P^{-1} to all strings in the language.

The accessible (or reachable) part of automaton G, denoted by $Ac(G)$, is the automaton obtained by removing from G all states that are not reachable from the initial state and those transitions that either initiate or end at the removed states. The coaccessible (or coreachable) part of automaton G, denoted by $CoAc(G)$, is the automaton obtained by removing from G all states from which it is not possible to reach some marked state and their respective transitions. The parallel composition [2] between two automata G_1 and G_2 is denoted as $G_1 \| G_2$, and is performed to synchronize the behaviors of two automata by allowing common events to occur only when they are in the active event sets of the current states of both automata and private events, *i.e.,*, events that belong to only one of the automata, to freely occur. The languages generated and marked by $G_1 \| G_2$ are, respectively, $L(G_1 \| G_2) = P_1^{-1}[L(G_1)] \cap P_2^{-1}[L(G_2)]$ and $L_m(G_1 \| G_2) = P_1^{-1}[L_m(G_1)] \cap P_2^{-1}[L_m(G_2)]$, where

$P_i : (\Sigma_1 \cup \Sigma_2)^* \rightarrow \Sigma_i^*$, for $i = 1, 2$.

B. Supervisory control of DES

Let $G = (X, \Sigma, f, \Gamma, x_0, X_m)$ be the automaton that models the "uncontrolled behavior" of a DES. In some cases, this behavior is unsatisfactory and needs to be "modified" by some control action. The task of restricting the behavior of a DES can be done by using a structure called supervisor. A supervisory control system is a feedback control system that, through the action of a supervisor S acts by enabling (or disabling) event occurrences based on the strings generated by plant G so as to make the closed-loop behavior equal to a given applicable language requirement $K \subseteq L(G)$. In that case, we say that the closed-loop system S/G (read S controlling G) is such that $L(S/G) = K$.

The supervisor is a mapping $S : L(G) \rightarrow 2^{\Sigma}$, where, for a given string $s \in L(G)$, $\Gamma[f(x_0, s)] \cap S(s)$ are the events that are enabled by S at state $f(x_0, s)$. The set of events Σ may be partitioned as $\Sigma = \Sigma_c \dot{\cup} \Sigma_{uc}$, where Σ_c and Σ_{uc} are, respectively, the sets of controllable events, which can be disabled by the supervisor, and uncontrollable events, whose occurrence cannot be preempted by the supervisor. In this regard, we say that a supervisor is admissible if for all $s \in L(G)$, $\Gamma[f(x_0, s)] \cap \Sigma_{uc} \subseteq S(s)$. The language achieved by the closed-loop system S/G is, therefore, recursively characterized as: *(i)* $\varepsilon \in L(S/G)$, and *(ii)* $s\sigma \in L(S/G) \Leftrightarrow s \in L(S/G) \wedge s\sigma \in L(G) \wedge \sigma \in S(s)$. The language marked by the closed-loop system is equal to the part of $L_m(G)$ that remains after the supervisory control action, *i.e.* $L_m(S/G) = L(S/G) \cap L_m(G)$. A supervisor is said to be nonblocking if $L(S/G) = \overline{L_m(S/G)}$.

A realization of a supervisor S is an automaton $R = (X_r, \Sigma, f_r, \Gamma_r, x_{0_r}, X_r)$ whose states are all marked, and is such that, for all $s \in L(G)$, $S(s) = \Gamma_r[f_r(x_{0_r}, s)]$. Thus, the closed-loop behavior is characterized by $R \| G$, since $L(S/G) = L(R \| G)$ and $L_m(S/G) = L_m(R \| G)$.

III. Asynchronous Configuration

A. Model

Suppose a module M_0 wishes to enable an event go_1 "remotely" in controlled machine M_1 by sending a message en - enable. The machines should not execute go_1 before receiving the message, but the module M_0 need not wait for M_1 to execute other tasks, that we will illustrate as an event $task$. Also, if M_1 has just executed go_1, it must not do so again until it has execute task ret_1 - return, and once again receive an en. To couple M_0 and M_1 as describes, we define a 'mailbox' MB_1 that will act as the supervisor of this configuration.

For this, we define automaton $M_0 = (X_0, \Sigma_0, f_0, \Gamma_0, x_{0_0}, X_m)$, where the states in $X_0 = \{0, 1, 2\}$ can be interpreted as 'busy', 'idle' and 'communicating', $\Sigma_1 = \{rdy, en, task\}$ with rdy stating for ready, transition function f_0 and the set of active events Γ_0 are defined according to the desired sequence, the state 0 is the initial state x_{0_0} and the only state in the set of marked states X_{m_0}. The automaton M_0 is illustrated in figure 1. Although module M_0 intends to control the behavior of M_1, it has no direct influence on the later, thus we characterize M_0 as part of the plant with uncontrolled behavior, alongside M_1.

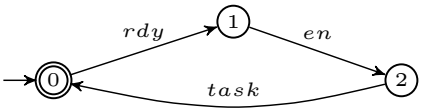

Fig. 1: Controller module M_0.

Similarly, we define automaton $M_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0_1}, X_m)$, where the states in X_1 are the locations machine M_1 can be (rest or advanced), $\Sigma_1 = \{go_1, ret_1\}$, transition function f_1 and the set of active events Γ_1 are defined according to the connectivity, the rest position is the initial state x_{0_1} and also the only state in the set of marked states X_{m_1}. The automaton M_1 is illustrated in figure 2.

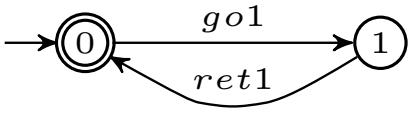

Fig. 2: Machine M_1.

Lastly, we define mailbox $MB_1 = (X_{MB_1}, \Sigma_{MB_1}, f_{MB_1}, \Gamma_{MB_1}, x_{0_{MB_1}}, X_m)$ as the supervisor of the system. Since MB_1 has events $\sigma \in \Sigma_{MB_1}$ common to Σ_0 and Σ_1, it couples M_0 and M_1 by enforcing the desired specification, that enables go_1 only after the occurrence of event en. The generated automaton of supervisor MB_1 is shown on figure 3.

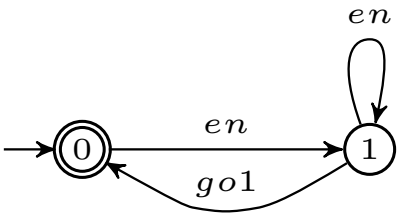

Fig. 3: Supervisor MB_1.

The closed-loop behavior of the asynchronous configuration can be characterized by the parallel composition of M_0, M_1 and MB_1, that we will nominate $Sync_1$. This composition has $|X| = 12$ states, $|\Gamma| = 21$ transitions and $|\Sigma| = 5$ events.

We can verify if $L_a(K) \subseteq L_m(Sync_1)$ by doing the following procedure. We elaborate automaton Vio_1 that marks strings that violates the specification, as shown in figure 4, in this model we need that go_1 occurs only after en and that ret_1 happens after go_1. Afterward, we marks all states of $Sync_1$, *i.e.*, $L_m(Sync_1) = L(Sync_1)$, and make the parallel composition $Sync_{1m} \| Vio_1$. If $L_m(Sync_{1m} \| Vio_1) = \emptyset$, we ensure that our model doesn't contain substrings that violates the specifications, to verify this we take the coaccessible part of $Sync_{1m} \| Vio_1$, if it is an empty automaton $L_m(Sync_{1m} \| Vio_1) = \emptyset$.

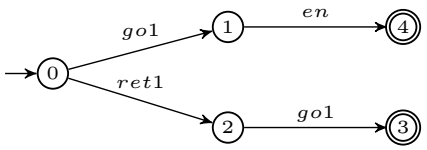

Fig. 4: Specification violation recognizer Vio_1.

B. Model Growth

Until now we explored the asynchronous configuration with only one machine being controlled but this approach can be extended to two or more controlled machine $M_1, M_2, \ldots, M_n$. By increasing the number of controlled modules, we need to increase the number of supervisors MB_n coupling M_0 to M_n. This configuration is illustrated in figure 5.

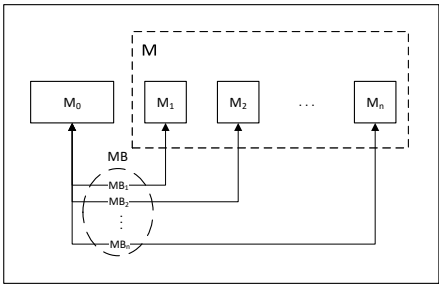

Fig. 5: Asynchronous configuration model scheme.

We will now explore how to add one more machine to the model, but the same method is used when adding more machines.

First of all, all machines M_n has the same automaton than M_1 only relabeling events go_n and ret_n with $n \in \{1, 2, \ldots, n\}$. We can consider all controlled machines as one compounded machine $M_{1,2}$ by doing a parallel composition $M_1 \| M_2$, as presented in figure 6.

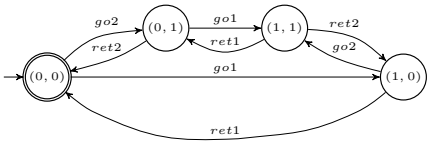

Fig. 6: Compounded machine $M_{1,2}$.

Equally, all supervisors MB_n add new specification analogue to MB_1, by relabeling the event go_n with $n \in \{1, 2, \ldots, n\}$. As the new supervisors add independent specifications, we can compound them in a single supervisor $MB_{1,2}$ by doing a parallel composition $MB_1 \| MB_2$, as presented in figure 7.

Hence, we have the same configuration as one machine, module M_0, grouped machines $M_{1,2}$ and supervisor $MB_{1,2}$. By doing the parallel composition $M_0 \| M_{1,2} \| MB_{1,2}$ we achieve a automaton with with $|X| = 48$ states, $|\Gamma| = 120$ transitions and $|\Sigma| = 7$ events.

Moreover, the following of the restriction can analogously be verified by creating an automaton that marks substrings that violates the specifications. Since we already elaborated automaton Vio_1 previously, we just need to add new substrings

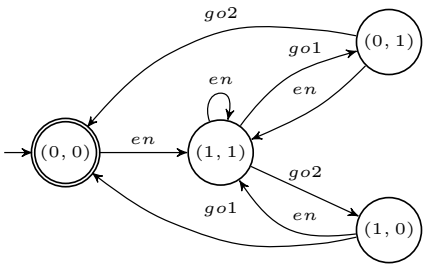

Fig. 7: Compounded supervisor $MB_{(1,2)}$.

regarding the violations to the new machines, thus we need to create a similar automaton, say Vio_2, that marks those substrings, and take the union $Vio_1 \cup Vio_2$ of the automata, resulting in automaton $Vio_{1,2}$ shown in figure 8, that will be used in the procedure described previously.

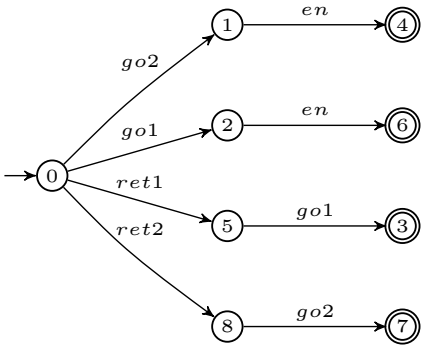

Fig. 8: Specification violation recognizer $Vio_{1,2}$.

Due to the nature of parallel composition, that synchronizes common events while letting independent events occur, if G_1 and G_2 have no common events, than we have the worst case of $G_1 \| G_2$ having $n \times m$ states, with n and m being the number of states in G_1 and G_2 respectively. This can be observed when we get the parallel composition M_n.

As we increase the number of machines, we get a growth in $M_{(1,\ldots,n)}$ states that is $\mathcal{O}(2^n)$. The same applies to $MB_{(1,\ldots,n)}$, that has a growth in states that is $\mathcal{O}(2^n)$. Assuming we maintain the same M_0, that has 3 states, we can imply that the growth in complexity of this configuration is $\mathcal{O}(3 \times 2^n \times 2^n) = \mathcal{O}(2^{2n})$, therefore, $|X| = \mathcal{O}(4^n)$, as we can compare with experimental results using $n = \{1, \ldots, 5\}$ shown in table I.

TABLE I: Asynchronous growth experimental data.

# of Machines	# of States	# of Transitions	# of Events
1	12	21	5
2	48	120	7
3	192	624	9
4	768	3072	11
5	3072	14592	13

IV. Synchronous Configuration

A. Synchronous Model

Now, suppose M_0 should not progress past state 2 (communicating) until all enabled events go_n have occurred, *i.e.*, M_0 waits for its message to perform its $task$ event. Firstly,

we focus in the model with two machines. We can re-model M_0 as $N_2 = (X_2, \Sigma_2, f_2, \Gamma_2, x_{0_2}, X_m)$, with $\Sigma_2 = \{en, task, go_1, go_2\}$, and the structure presented in figure 9. Moreover, since N_2 waits, there is logically no need for a supervisor MB at all.

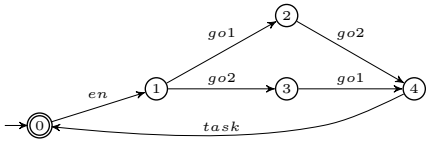

Fig. 9: Controller module N_2.

Therefore, we can elaborate the synchronous model closed-loop behavior by the parallel composition of N_2 and $M_{1,2}$, that we previously presented, that we will nominate $Sync_2$. This composition has $|X| = 20$ states, $|\Gamma| = 36$ transitions and $|\Sigma| = 6$ events. Due to the synchronization restriction of N_2, there is no need to verify if $L_m(Sync_2)$ violates the wanted requirements, since N_2 acts as the supervisor of this configuration.

B. Model Growth

In the later configuration, we extended the model by simply increasing the number of controlled machines and corresponding supervisor, however, in the synchronous configuration we cannot just increase the number of controlled machines, we also need to modify the controller module N_n to include all the new events $go_n \in \Sigma_{M_{1,\dots,n}}$, with $n \in \{1, 2, \dots, n\}$, an ensure that all go_n has occurred in whichever combination before N_n execute event $task$. This configuration is illustrated in figure 10.

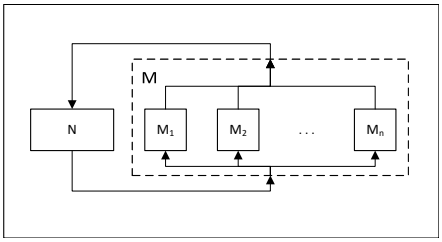

Fig. 10: Synchronous configuration model scheme.

Notice that, when we expand N_n, the increase in the number of states resulting from the combination of events go_n is $\mathcal{O}(2^n)$, although the model have one more state (the initial one), asymptotically it does not affect our analysis, as $(2^n + 1) \in \mathcal{O}(2^n)$.

Therefore, we can evaluate the synchronous configuration complexity since we use the same $M_{(1,\dots,n)}$ as the later configuration, that is $\mathcal{O}(2^n)$. By calculating the complexity, we can expect that the growth is $\mathcal{O}(2^n \times 2^n) = \mathcal{O}(2^{2n})$, thus, $|X| = \mathcal{O}(4^n)$, as compared with experimental results shown in table II.

Comparing the two configurations experimental results of the complexity growth, is noticeable that, although having the same growth in complexity, the increase in number of states is much smaller in the synchronous configuration. This is due to the fact that, since the number of states in M_0 does not scale with the number of machines in the asynchronous configuration, asymptotically it does not impact the analysis of the growth, as it is treated as a constant.

TABLE II: Synchronous configuration experimental data.

# of Machines	# of States	# of Transitions	# of Events
2	20	36	6
3	72	172	8
4	272	832	10

V. Concluding Remarks

In this paper we addressed two configurations of message passing models, the asynchronous, that the controller has no need to wait for the machines to realize it independent tasks, but the machines have to wait the enable event to execute their tasks, and the synchronous, where the controller waits for all machines tasks being started to execute their own, yet the machines still have to wait for the controller.

Regarding the growth in complexity, we analyzed that both configurations growth in the number the states are $\mathcal{O}(4^n)$. Although the same growth in complexity, we also noticed that the synchronous configuration has less states than the asynchronous with the same number of machines. Nevertheless, we can still consider the asynchronous configuration instead of the synchronous, because of the ease to scale, as is only needed to replicate the machines M_n and supervisor MB_n automata for the parallel composition, instead of remodeling the controller on each increase of machine, that can be an exhaustive process.

A future work regarding the theme include finding a procedure to generate the controller of the synchronous configuration, as all 2^n combinations of go_n is needed to generate the automaton, or another method to identify the occurrence of each go_n once.

References

[1] W. M. Wonham, K. Cai *et al.*, "Supervisory control of discrete-event systems," 2019.
[2] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*, 2nd ed. New York: Springer, 2008.
[3] P. J. G. Ramadge and W. M. Wonham, "The control of discrete event systems," *Proceedings of the IEEE*, vol. 77, no. 1, pp. 81–98, 1989.