Reasoning about Adaptivity of Agents and Multi-Agent Systems

Graeme Smith*, J.W. Sanders†‡ and Kirsten Winter*

*School of Information Technology and Electrical Engineering, The University of Queensland, Australia

†African Institute for Mathematical Sciences (AIMS), South Africa

‡Department of Mathematical Sciences, Stellenbosch University, South Africa

***Abstract*—Although adaptivity is a central feature of agents and multi-agent systems (MAS), there is no precise definition of it in the literature. What does it mean for an agent or for a MAS to be adaptive? How can we reason about and measure the ability of agents and MAS to adapt? In this paper, we provide a formal definition of adaptivity of agents and MAS aimed at addressing these issues.**

The definition is independent of any particular mechanism for ensuring adaptivity. It is qualified by the environmental actions to which the agents adapt, and quantified by the number of actions needed for adaptivity. It is formalised using a simple extension to labelled transitions systems allowing it to be applied to specifications of MAS in a wide range of existing formal notations. We show by a simple example how it can be used to detect design flaws which lead to situations in which a system is unable to adapt.

***Keywords*-adaptivity, multi-agent systems, formal methods, team automata**

I. Introduction

In Biology, adaptivity refers to the long-term gradual adjustment of a species to cope better with its environment. In Informatics, it can mean also an abrupt system change in response to disruption from the environment, or failure of one of the system's components. Multi-agent systems (MAS) exhibit both forms of adaptivity. Agents adapt gradually to their environment using, for example, machine learning techniques, and the distributed nature of MAS is exploited to make them robust against both external disturbances and agent failures.

To arrive at a definition of adaptivity for agents and MAS, we begin by asking what is the fundamental feature of an adaptive system. One suggestion is its ability to "change its behaviour" to suit its environment [10]. This notion of changing behaviour at first conjures up visions of systems which are somehow more advanced than standard computer programs. It must be pointed out, however, that changing behaviour is illusory since a system, when viewed at a certain level of abstraction as a simple state machine, does not actually change what it is capable of doing. As shown in [10], it simply moves to a new state in which different actions and environmental interactions are possible. This is true even of approaches to evolutionary computing [6] and machine learning [16]: underlying such a system is just a computer program.

So rather than changing behaviour we could focus on a system's ability to produce different behaviour based on its interactions with its environment. However, this ability is common to all systems which we would regard as reactive. For example, a thermostat which turns a heater on when the temperature drops below 20 degrees Celsius and turns it off when the temperature rises above 24 degrees Celsius fits this definition. However, we would not generally regard a thermostat-controlled heater as an adaptive system.

To see what distinguishes an adaptive system from a merely reactive one, we appeal to the notion of *legitimate states* of a system introduced by Dijkstra [3]. In this work, 'legitimacy' is defined by a state invariant capturing those states in which the system behaves as it was intended. Dijkstra's paper is concerned with self-stabilisation of distributed systems. His examples consist of token ring networks in which a legitimate state is one in which there is exactly one token present. He presents several algorithms which, given an arbitrary number of tokens initially, end up in a legitimate state.

Dijkstra's systems are *closed* in the sense that they do not interact with an external environment. The notion of legitimate states must for our purposes be extended to *open* (or reactive) systems by considering the state to include both that of the system and its environment. Under this definition, a legitimate state of the thermostat-controlled heater would be any in which the environment is such that the thermostat and heater could operate correctly, *e.g.*, an environment in which power is supplied to the heater.

An important feature of Dijkstra's self-stabilising networks is that even if they start in states which are not legitimate states, they are guaranteed to reach legitimate states after a finite number of system actions. A reactive system such as the thermostat may not operate at all when not in a legitimate state. Importantly, it does not perform actions which allow it to reach a legitimate state. An adaptive system, on the other hand, is a reactive system which, like Dijkstra's self-stabilising token rings, is always able to reach legitimate states from illegitimate ones.

In other words, an adaptive system is one which, when placed in a particular environment, has a defined set of legitimate states and when in an illegitimate state reaches a legitimate state again. Indeed, the period before the system reaches the legitimate state is the time when the system is adapting.

Following Dijkstra's definition, a system in a legitimate state is not able to enter an illegitimate state on its own accord. That is, defined transitions of the system from legitimate states enter only other legitimate states. A system is placed in an illegitimate state by an *external* action, *i.e.*, an action that is not regarded as part of the system's specification. This action may represent a change in the environment due to an unforeseen disturbance, or the passing of a threshold point in an environment that is gradually changing over time. Alternatively, an external action may represent a change to the system itself. In the case where the system is an agent, this may be caused, for example, by the action of a software virus changing internal data. In the case where the system is a MAS, it may be caused by the failure of a component agent.

In each case we can reason about adaptivity as the ability to "recover" from the external event, *i.e.*, the ability of the system to reach a legitimate state. Since systems will, in general, be adaptive to only a subset of all possible external events, we qualify our definition of adaptivity with respect to an external event. We also quantify our definition of adaptivity with respect to the number of actions required for the system to adapt. This provides us with a metric for comparing different adaptivity mechanisms.

To formalise our definition of adaptivity we begin by presenting a formal model of agents and MAS in Section II. This model is based on a simple extension of labelled transition systems which has been advocated for modelling reactive systems. It is independent of any specific formal notation, yet can be readily mapped to, and hence used with, a wide range of existing formal notations. Section III provides our formal definition of adaptivity for agents and MAS. It builds on a definition of adaptivity for closed systems presented in [17]. We show by a simple example how it can be used to detect design flaws which lead to situations in which a system is unable to adapt. In Section IV, we demonstrate the use of the definition with Object-Z [18], a formal notation that has been advocated for modelling MAS [11]. We relate our definition of adaptivity to a range of informal definitions occurring in the literature in Section V before concluding with a discussion of related work in Section VI.

II. An Agent Model

In order to present a formal definition of adaptivity, we begin by providing formal representations of agents and MAS. Since we consider agents as being artifacts that are realised by software, they can – on a low level of abstraction – be represented as labelled transition systems (LTS). An LTS comprises a (possibly infinite) set of states, a (possibly infinite) set of initial states, and a collection of actions which cause (possibly nondeterministic) state transitions. Similar concepts have been used in the agent literature before. For example, formalisms with an underlying transition systems semantics such as Z and Object-Z have been suggested for modelling agents and MAS [4], [11]. Also, Hunter and Delgrande [14] use transition systems which they extend with a metric function to capture "plausibility" amongst belief states. In this work we assume such a metric can be encoded in the transitions system.

Definition 1: An LTS is a 4-tuple $S = (Q, I, \Sigma, \delta)$ where

- Q is the (possibly infinite) set of states of the agent.
- $I \subseteq Q$ is the non-empty set of initial states.
- Σ is the set of actions (or labels).
- $\delta \subseteq Q \times \Sigma \times Q$ is the agent's set of labelled transitions.

A *behaviour* of an LTS, S, is a possibly infinite sequence alternating between states and actions $q_0\ a_1\ q_1\ a_2\ q_2\ \ldots$ where for all $i > 0$, $a_i \in \Sigma$ such that $(q_{i-1}, a_i, q_i) \in \delta$.

Let $\mathcal{B}(S)$ denote the behaviours of S starting from an initial state of S, *i.e.*, where $q_0 \in I$, and $\mathcal{B}(S, Q')$ denote behaviour of S starting in a state $q_0 \in Q'$ where $Q' \subseteq Q$. Let $st(b, i)$ denote the ith state of b, and let $act(b, i)$ denote the *ith* action.

To facilitate reasoning about environmental interaction, we use a simple extension of LTS in which actions are partitioned into three sets: *internal* actions, *input* actions (externally observable actions controlled by the environment), and *output* actions (externally observable actions controlled by the component).

Such a partitioning of actions has been proposed for modelling reactive systems. It is central to the *I/O automata* approach of Lynch and Tuttle [15], and *interface automata* of de Alfaro and Henzinger [2]. In each of these approaches, combined automata interact by synchronising on common-named input and output actions. All automata with a given action are involved in each synchronisation on that action.

The main difference between I/O automata and interface automata is that the former are *input-enabled* meaning that input actions can never be refused. This is not the case with interface automata where the restrictions on the type of input actions and when they can occur is used to model assumptions on the system's environment.

An approach similar to interface automata has also been proposed for modelling groupware systems by Ellis [7]. This approach has been formalised and further developed by Beek *et al.* [1]. The automata are referred to as *component automata*, and component automata which are formed as the composition of other component automata as *team automata*. The major difference with the aforementioned approaches to reactive systems is that in a team automaton not all of the composed automata with a given action need to be involved in a synchronisation on that action. This flexibility has been shown to be well suited to formalising notions of

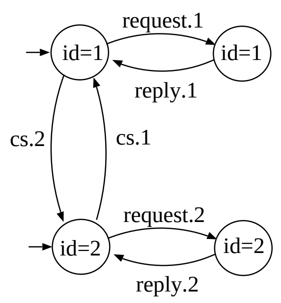

Figure 1. Component automaton of the client agent

coordination, cooperation and collaboration in a distributed setting [1]. In the remainder of this section, we show how agents and MAS can be modelled using component and team automata.

A. Agents as Component Automata

Agents are modelled as component automata [1].

Definition 2: An agent is an LTS, $A = (Q, I, \Sigma = \Sigma_{int} \cup \Sigma_{inp} \cup \Sigma_{out}, \delta)$, where Σ_{int}, Σ_{inp} and Σ_{out} are pairwise disjoint, and

- Σ_{int} is the set of *internal* actions. Such actions are controlled by the agent and are not externally observable.
- Σ_{inp} is the set of *input* actions. Such actions are externally observable and are controlled by the agent's environment.
- Σ_{out} is the set of *output* actions. Such actions are externally observable and are controlled by the agent.

Example 1: Consider an agent *Client* which is aware of a number of servers in its environment with which it can interact. The state of the client includes the set of server identifiers and the identifier of the server with which it is currently interacting. It has a set of internal actions *cs.id* which allows it to change the server with which it is interacting to that with identifier *id* (initially the client is interacting with any server of which the agent is aware), a set of output actions *request.id* representing a request to the server with identifier *id*, and a set of input actions *reply.id* representing a reply from the server with identifier *id*.

Assume there are two available servers with identifiers 1 and 2. An LTS that models the client can be depicted as in Figure 1, where incoming arrows mark initial states. In this model, the client changes server only when it has not made a request.

To represent this system as a component automata, we simply partition its actions as follows.

$$\begin{aligned} \Sigma_{int} &= \{cs.i \mid i \in 1 \,..\, 2\} \\ \Sigma_{inp} &= \{reply.i \mid i \in 1 \,..\, 2\} \\ \Sigma_{out} &= \{request.i \mid i \in 1 \,..\, 2\} \qquad \diamond \end{aligned}$$

Given this partitioning, the fact that the input action *reply.id* occurs only after *request.id*, for $id \in 1 \,..\, 2$, is an assumption that has been made about the client's environment. It is not something the client could itself enforce.

B. Multi-agent systems as Team Automata

When component automata are composed, they potentially synchronise on common-named actions. Hence to prevent unwanted synchronisations, a precondition for composing a group of agents is that no internal action of one agent is present as an action (internal or external) of another agent.

Let A_i denote the agent $(Q_i, I_i, \Sigma_i = \Sigma_{i,int} \cup \Sigma_{i,inp} \cup \Sigma_{i,out}, \delta_i)$, for $i \in 0 \,..\, n$. A composition of the agents $A_0, \ldots, A_n$ is possible if

$$\forall\, i \in 0 \,..\, n \bullet (\Sigma_{i,int} \cap \bigcup_{j:0..n \setminus \{i\}} \Sigma_j) = \varnothing. \qquad (1)$$

Given such a composable set of agents, a multi-agent system (MAS) is modelled as a special kind of component automaton called a *team automaton* [1]. A state q of the team automaton is a tuple of the possible states of the agents, $q \in \prod_{i:0..n} Q_i$. We let q_j, for $j \in 0 \,..\, n$, denote the jth element of the tuple q.

Definition 3: A MAS comprising agents $A_0, \ldots, A_n$ is an LTS, $M = (Q, I, \Sigma = \Sigma_{int} \cup \Sigma_{inp} \cup \Sigma_{out}, \delta)$, where Σ_{int}, Σ_{inp} and Σ_{out} are pairwise disjoint, and

- $Q = \prod_{i:0..n} Q_i$.
- $I = \prod_{i:0..n} I_i$.
- $\Sigma_{int} = \bigcup_{i:0..n} \Sigma_{i,int}$.
- $\Sigma_{out} = \bigcup_{i:0..n} \Sigma_{i,out}$.
- $\Sigma_{inp} = (\bigcup_{i:0..n} \Sigma_{i,inp}) \setminus \Sigma_{out}$.
- $\delta \subseteq Q \times \Sigma \times Q$ such that
  - for all $(q, a, q') \in \delta$, there exists a $j \in 0 \,..\, n$ such that $(q_j, a, q'_j) \in \delta_j$ and for all $i \in 0 \,..\, n$ with $i \neq j$, $(q_i, a, q'_i) \in \delta_i$ or $q_i = q'_i$
  - for all $q, q' \in Q$ and $a \in \Sigma_{int}$, if there exists a $j \in 0 \,..\, n$ such that $(q_j, a, q'_j) \in \delta_j$, then $(q, a, q') \in \delta$.

The internal and output actions of M are those of the agents. The input actions are those of the agents which are not also output actions. In the case where an input action of one agent is the same as that of an output action of another agent, the input is assumed to be caused by the output action and hence is not an input action for the MAS. The fact that the output action is not also removed from the system allows team automata to be further composed with other component or team automata, *e.g.*, to act as the environment of a component in a further composition.

The transitions of M are such that the following hold.

(i) Each transition involves a non-empty subset of agents engaging in the action a. The state of each agent not involved in the action remains unchanged.

(ii) There is a MAS transition for each agent transition corresponding to an internal action.

Not all agents with action a need to be involved in a system transition corresponding to a. This allows different interaction strategies to be captured [1]. However for consistency, we require that any output action of the system involve at least one agent output action, *i.e.*, there should not be a system action a involving only an agent which has a as an input action when there are other agents which have a as an output action. More formally

$$\forall (q, a, q') \in \delta \bullet a \in \Sigma_{out} \Rightarrow \exists j \in 0 \,..\, n \bullet a \in \Sigma_{j,out} \wedge (q_j, a, q'_j) \in \delta_j. \quad (2)$$

Furthermore, given a transition (q, a, q') of a MAS such that $q_j = q'_j$ for some $j \in 0 \,..\, n$, if the jth agent has a transition (q_j, a, q_j) then the agent undergoes this action, otherwise (*i.e.*, if $(q_j, a, q_j) \notin \delta_j$) it undergoes no action. This *maximal* interpretation suggested by Beek *et al.* [1] removes any ambiguity concerning which agents participate in a particular MAS transition.

Example 2: To continue Example 1 above we assume that each server is defined as $Server_i = (Q_i, I_i, \Sigma_{i,int} \cup \Sigma_{i,inp} \cup \Sigma_{i,out}, \delta_i)$ with $\Sigma_{i,int} = \varnothing$, $\Sigma_{i,out} = \{reply.i \mid i \in 1 \,..\, 2\}$ and $\Sigma_{i,inp} = \{request.i \mid i \in 1 \,..\, 2\}$. The behaviour of the two servers is modelled abstractly in Figure 2. (For simplicity, we assume that a server deals with only one client at a time).

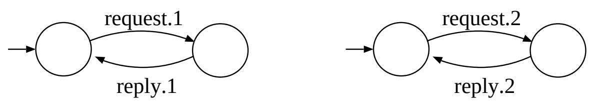

Figure 2. Component automata of the server agents

The agents *Client*, $Server_1$ and $Server_2$ can be composed since (1) holds.

Given $Client = (Q_{Client}, I_{Client}, \Sigma_{Client}, \delta_{Client})$, one team automaton that can be composed from *Client* and the servers $Server_1$ and $Server_2$ is $M = (Q, I, \Sigma_{int} \cup \Sigma_{inp} \cup \Sigma_{out}, \delta)$ with

- $Q = Q_{Client} \times \prod_{i:1..2} Q_i$.
- $I = I_{Client} \times \prod_{i:1..2} I_i$.
- $\Sigma_{int} = \{cs.i \mid i \in 1 \,..\, 2\}$.
- $\Sigma_{inp} = \varnothing$.
- $\Sigma_{out} = \{reply.i, request.i \mid i \in 1 \,..\, 2\}$.
- $\delta = \{(q, a, q') \in Q \times \Sigma \times Q \mid (q_0, a, q'_0) \in \delta_{Client} \wedge (\exists j \in 1 \,..\, 2 \bullet (q_j, a, q'_j) \in \delta_j \wedge (\forall i \neq j \bullet q_i = q'_i))\}$.

Since all common-named actions synchronise, and all actions which are enabled in a component can occur, the definition satisfies (2).

It is possible, by restricting δ in such compositions, to limit when operations are enabled, or to limit the agents which synchronise on an action. For example, if we had included two client agents, then we would expect only one to be involved in each request, reply and change-server action. $\diamond$

III. Adaptivity

In this section we provide formal definitions of adaptivity for agents and MAS based on their team automata representations as defined in Section II. We base our definitions on Dijkstra's notion of legitimate states [3] which we extend to include both the state of the system under consideration (agent or MAS) and its environment. The definitions qualify adaptivity with respect to the external action to which the system adapts, and quantify it with respect to the number of actions required to adapt.

Since agents and MAS are represented by automata, the definition of adaptivity for each of them is identical. We begin by defining adaptivity in the special case of *closed systems*, *i.e.*, where the system does not interact with its environment, in Section III-A. This definition is applicable to MAS which do not rely on environmental interaction for their operation. It is based on a definition in [17]. We then extend the definition to *open systems*, *i.e.*, where interaction with the environment is central to the system's operation, in Section III-B. This definition is applicable to agents as well as MAS that interact with their environment.

A. Adaptivity of closed systems

A closed MAS M can be modelled by a team automata with no input actions. The team automata of Example 2 is an example of such a closed system. Let $\mathcal{Q}(M)$ denote the set of legitimate states of M. By definition, all transitions from legitimate states lead to legitimate states. That is, given $M = (Q, I, \Sigma, \delta)$

$$(q, a, q') \in \delta \wedge q \in \mathcal{Q}(M) \Rightarrow q' \in \mathcal{Q}(M). \quad (3)$$

A MAS is *well-formed* if the initial states of M are legitimate states, or if M is guaranteed to reach a legitimate state in a finite number of actions. That is,

$$I \subseteq \mathcal{Q}(M) \vee (\forall b \in \mathcal{B}(M) \bullet \exists i \geq 0 \bullet st(b, i) \in \mathcal{Q}(M)). \quad (4)$$

In the case where a finite number of actions are required to reach a legitimate state, the MAS undergoes an initial (self-)configuration process.[1]

Let Q be the set of states of M and Z be an external action defining the set of transitions $\zeta \subseteq Q \times Z \times Q$ on M. Such an external action can move the MAS from a legitimate state to an illegitimate one. M can adapt to the external action, if it can return to a legitimate state.

Definition 4: A closed MAS M is Z-adaptive if, after an occurrence of Z which places the MAS in an illegitimate state, the MAS is guaranteed to reach a legitimate state in

[1]Self-configuration can itself be viewed as a type of adaptivity in which the external action is the system initialisation.

a finite number of transitions under the assumption of no further occurrences of Z.

That is, for all $b \in \mathcal{B}(M)$ such that there exists an $i \geq 0$ such that $st(b,i) = q$ and for all illegitimate states q' such that $(q, Z, q') \in \zeta$ the following holds.

$$\forall b' \in B(M, \{q'\}) \bullet \exists j > 0 \bullet st(b', j) \in \mathcal{Q}(M) \tag{5}$$

Note that we are concerned only with cases where Z places the MAS in an illegitimate state. If Z places the MAS in a legitimate state, the MAS is robust against Z, but we do not regard this as adapting (since there is no deflection from its normal behaviour).

A closed MAS M is n-Z-adaptive for some $n > 0$, if it can adapt within at most n transitions. That is, for all $b \in \mathcal{B}(M)$ such that there exists an $i \geq 0$ such that $st(b,i) = q$ and for all illegitimate states q' such that $(q, Z, q') \in \zeta$ the following holds.

$$\forall b' \in B(M, \{q'\}) \bullet \exists j \in 1 \mathinner{..} n \bullet st(b', j) \in \mathcal{Q}(M) \tag{6}$$

The following theorems follow directly from these definitions.

Theorem 1: If M is n-Z-adaptive, it is also m-Z-adaptive for any $m \geq n$.

Theorem 2: If M is n-Z-adaptive for some $n > 0$, then it is also Z-adaptive.

Note that the inverse of Theorem 2 does not hold. It is possible, due to nondeterminism in a MAS, that there is no minimum number of transitions required to reach a legitimate state. For example, consider a MAS that after Z is repeatedly able to choose between two actions a and b, and reaches a legitimate state after choosing b. If we assume fairness (so that b must eventually be chosen) the MAS is Z-adaptive. However, there is no n for which it is n-Z-adaptive.

B. Adaptivity of open systems

An agent provides an example of an *open* system since its behaviour typically depends on its environment. The environment controls the agent's input actions, and may restrict the occurrence of its output actions when synchronisation is required. Similarly, a MAS can be an open system. In this section we will discuss adaptivity of agents, although the results are also directly applicable to open MAS.

To reason about an agent, we need to model the interactions with its environment. In interface automata, this is taken care of by the restrictions placed on the types of observable (input and output) actions and when they can occur [2]. The same is true for component and team automata.

With open systems, there are two kinds of external actions. The first change the state of the agent. They are identical to the external actions of closed systems and adaptivity to these actions can be reasoned about in the same way.

The second kind of external actions change the state of the system's environment, and possibly also the system's state. In the setting of component automata, this would manifest itself as (possibly) different restrictions on the observable actions.[2]

To facilitate modelling such an external action, we need to extend the agent's state with one or more auxiliary variables which the external action changes. These auxiliary variables, representing some facet of the environment, can be used to restrict when particular observable actions can occur. They are also used in defining the legitimate states.

Example 3: Consider the client agent of Section II. A possible external action that could occur in its environment is a server going down. Let Z_1 be the external action that causes a server with which the client is interacting to go down when the client is in the state where it has not performed a request. Let Z_2 be an external action similar to Z_1 which occurs when the client has performed a request and is waiting for a reply.

To reason about the adaptivity of the client, we extend it with an auxiliary variable *down* which is the set of servers which are currently down. This set is initially empty. The transitions are restricted such that if the extended client is in a state where server i is down, transitions corresponding to *cs.i*, *request.i* and *reply.i* cannot occur. If it is in a state where server i is not down, the transitions can occur. The transitions do not change whether a given server is up or down.

Figure 3 shows part of the restricted client automaton. We show the states in which both servers are up, and also the states in which server 1 is down. We choose the legitimate states to be those where the client can perform request and reply actions. These states are shaded in the figure. The external actions are shown using dotted arrows between states.

After a single occurrence of Z_1, the client is able to perform the change-server action restoring it to a legitimate state. Hence, the client is Z_1-adaptive. In fact, since it requires only one action to reach a legitimate state, it is 1-Z_1-adaptive. In a more detailed specification where, for example, the client was required to log in to the new server, the client would be n-Z_1-adaptive for some $n > 1$. Hence, the quantification of adaptivity with respect to number of actions is dependent on the level of abstraction. It should, therefore, be used only for comparing adaptive responses at the same level of abstraction.

In the case of Z_2, there is no possibility of performing the change-server action. The client is therefore not Z_2-adaptive. This may correspond to a design flaw which our reasoning allows us to detect and rectify if desired, *e.g.*, by introducing a timeout when waiting for a response. $\diamond$

[2]We assume all input actions possible in the environment are included in the agent automaton, as are all of the agent's possible output actions. Hence, no observable actions will be introduced or removed by such an external action.

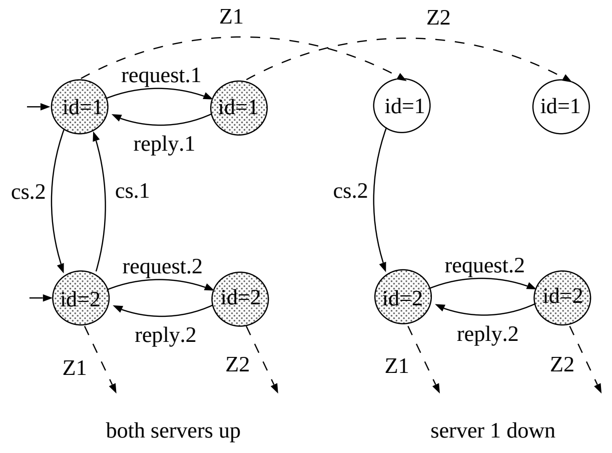

Figure 3. Team automata of the restricted client

As can be seen from Example 3, the specifier must, based on an understanding of the system and its environment, determine the available transitions from states corresponding to different values of the auxiliary variables. It is possible that these transitions change the values of the auxiliary variables. Although the agent cannot change these variables directly, it can interact with its environment to instigate their change. For example, if the agent of the above example was able to call a maintenance agent to fix the server, then as a consequence of this call it would return to a state where the server was no longer down. Whether the agent can do this and how the environment responds is up to the specifier.

To be adaptive to an external action Z, an agent must (i) be guaranteed to reach a legitimate state in a finite number of transitions, and (ii) have at least one behaviour which reaches a legitimate state without the auxiliary variables being changed. The second condition precludes agents which rely on the auxiliary variables changing to reach a legitimate state. For example, an agent may call a maintenance agent but have no other strategy for dealing with a server that is down. We would not regard such an agent as adaptive.

The approach is formalised as follows. We enhance the agent $A = (Q, I, \Sigma = \Sigma_{int} \cup \Sigma_{inp} \cup \Sigma_{out}, \delta)$ with a set of auxiliary variables of type E. The cross product $Q \times E$ captures the state space of A extended with these auxiliary variables. Thus, the enhanced version of the agent is defined as $A' = (Q \times E, I \times I_E, \Sigma, \delta')$ where $I_E \subseteq E$ and $\delta' \subseteq (Q \times E) \times \Sigma \times (Q \times E)$. We require that the behaviours of A' when restricted to Q correspond to behaviours of the original agent A. That is,

$$\forall\, b \in \mathcal{B}(A') \bullet b\,|_Q \in \mathcal{B}(A) \tag{7}$$

where $b\,|_Q$ denotes the behaviour b restricted to the state Q of the original agent A. Hence, A' behaves identically to A in the absence of external actions. This is true in Example 3 since initially no servers are down.

Let Z be an external action defining the set of transitions $\zeta \subseteq (Q \times E) \times Z \times (Q \times E)$ on A'.

Definition 5: An agent (or open MAS) A is Z-adaptive, if its enhancement A' on which Z is defined is, after an occurrence of Z which places it in an illegitimate state, able to reach a legitimate state in a finite number of transitions, and at least one behaviour reaches a legitimate state without changing the extension to the state of A.

That is, for all $b \in \mathcal{B}(A')$ such that there exists an $i \geq 0$ such that $st(b, i) = q$ and $(q, Z, q') \in \zeta$ the following holds.

$$\forall\, b' \in \mathcal{B}(A', \{q'\}) \bullet \exists\, j > 0 \bullet st(b', j) \in \mathcal{Q}(A') \tag{8}$$

and

$$\begin{aligned} &\exists\, b' \in \mathcal{B}(A', \{q'\}) \bullet \\ &\quad \exists\, j > 0 \bullet st(b', j) \in \mathcal{Q}(A') \wedge \\ &\quad\quad (\forall\, k \leq j \bullet st(b', k)\,|_E = q'\,|_E) \end{aligned} \tag{9}$$

where $s\,|_E$ restricts a state s of A' to the variables of E.

An agent (or open MAS) A which is Z-adaptive is n-Z-adaptive for some $n > 0$, if it can adapt within at most n transitions. That is, for all $b \in \mathcal{B}(A')$ such that there exists an $i \geq 0$ such that $st(b, i) = q$ and $(q, Z, q') \in \zeta$, the following holds along with condition (9) above.

$$\forall\, b' \in \mathcal{B}(A', \{q'\}) \bullet \exists\, j \in 1 \,..\, n \bullet st(b, j) \in \mathcal{Q}(A') \tag{10}$$

Theorems 1 and 2 of Section III-A remain true and follow directly from these definitions.

IV. Reasoning about adaptivity using Object-Z

The notions of component and team automata introduced in Section II have provided a convenient setting for the definition of adaptivity but they are inconvenient for expressing any but non-trivial examples. Fortunately several formal notations have been developed over the years for the purpose of expressing state-based examples of realistic proportions. The fact that our agent and MAS models are LTS makes it straightforward to use our definitions with such notations.

In this section we consider the Object-Z specification language [18] as an example; other notations could also be used. Object-Z is an object-oriented extension of the well known Z specification language [20]. Its notions of classes and objects are ideal for capturing descriptions of agents. Both Z and Object-Z have been advocated for the description of agents by other researchers in the field [4], [11].

A. Overview of Object-Z

Classes in Object-Z are represented by a named box which has a state schema, and zero or more operations. They may also have an initial state schema describing the class's initial states. In the absence of such a schema, all states allowed by the state schema are potential initial states.

For example, given a type *ID* of server identifiers and *ClientState* $= \{idle, waiting\}$, we can define the *Client* of Section II-A by the following class.

The state schema of class *Client* declares a variable *server* of type *ID* denoting the current server being used, and a variable *state* denoting the client's current state. The initial state schema states that initially the state is *idle*. This schema does not restrict the variable *server* which may be any value from the type *ID*.

Client

$server : ID$
$state : ClientState$

Init
$state = idle$

ChangeServer
$\Delta(server)$

$state = idle$
$server' \neq server$

Request
$\Delta(state)$
$id! : ID$

$state = idle$
$server = id!$
$state' = waiting$

Reply
$\Delta(state)$
$id? : ID$

$state = waiting$
$id? = server$
$state' = idle$

The class has three operations: *ChangeServer*, *Request* and *Reply*. Operation *ChangeServer* allows the value *server* to change. This is indicated by including the variable in a Δ-list (read "delta-list"). The *predicate part* of the operation, *i.e.*, the part below the horizontal line, restricts the values of the state variables both before and after the operation and any input and output variables declared in the *declaration part*, *i.e.*, the part above the horizontal line. State variables after an operation are represented by the variable name decorated with a prime, *e.g.*, $server'$. The predicate part of *ChangeServer* states that the value of *server* after the operation is not equal to its value before the operation. *Request* changes the state from *idle* to *waiting* and outputs its server's id (variables ending in ! denote output variables). *Reply* changes the state from *waiting* back to *idle* and inputs its server's id (variables ending in ? denote inputs).

Operations in Object-Z are *guarded*, *i.e.*, they can occur only when the restrictions on the state before the operation can be met. Hence, *ChangeServer* and *Request* can occur only when $state = idle$. Similarly, *Reply* can occur only when $state = waiting$ and $server = id?$.

Given a type $ServerState = \{idle, processing\}$, we can similarly define a class *Server*. Initially, the server's state is *idle*. In this state it can perform a *Request* operation, which corresponds to a client sending a request. This operation changes the server's state to *processing*. In this state, it can perform a *Reply* operation corresponding to the server sending a reply to a client. This operation returns the state of the server to *idle*.

Server

$id : ID$
$state : ServerState$

Init
$state = idle$

Request
$\Delta(state)$
$id? : ID$

$state = idle$
$id? = id$
$state' = processing$

Reply
$\Delta(state)$
$id! : ID$

$state = processing$
$id! = id$
$state' = idle$

The system of one client and a number of servers could then be defined by a class *System*. (Note that the actual number of servers is left unspecified here and can be any number greater than or equal to 2.)

System

$client : Client$
$servers : \mathbb{F}\, Server$

$\#servers \geq 2$
$\forall\, s_1, s_2 : servers \bullet s_1 \neq s_2 \Rightarrow s_1.id \neq s_2.id$

Init
$client.\textsc{Init}$
$\forall\, s : servers \bullet s.\textsc{Init}$

$ChangeServer \mathrel{\widehat=} client.ChangeServer$

$Request \mathrel{\widehat=} client.Request \parallel$
$\quad ([\!]\; s : servers \bullet s.Request)$

$Reply \mathrel{\widehat=} ([\!]\; s : servers \bullet s.Reply) \parallel client.Reply$

The state schema declares a single *Client* object (variable *client*) and a finite set of *Server* objects (variable *servers*). The predicate part of the state schema states that each server has a different id. Initially, the client and each of the servers are in their initial states (as defined by their classes).

The operation *ChangeServer* specifies that the client performs a *ChangeServer* operation. The operation *Request* specifies one server *s* (whose *Request* operation is enabled with the server identifier of the client) performing a *Request* operation in parallel with the client. The choice operator $[\!]$ is used to select one of the server objects, and the parallel operator $\parallel$ to place the client and server operation in parallel. The latter conjoins its argument operations and removes the

? and ! decorations from common-named inputs and outputs occurring in different operations. This effectively identifies these variables and hence equates their values. In this case, it equates $id!$ of *client.Request* with $id?$ of *s.Request* by renaming them both to *id*. This restricts the choice of the server s to the one whose *Request* operation is enabled with the server identifier output by the client.

The operation *Reply* similarly specifies one server s (whose *Reply* operation is enabled with the server identifier of the client) performing a *Reply* operation in parallel with the client.

B. Adaptivity in Object-Z

To reason about the adaptivity of a class such as *Client*, we follow the approach detailed in Section III.

We begin by enhancing *Client* with additional state variables denoting the part of the environment affected by an external action Z. This can be done using Object-Z's notion of inheritance. When a class in Object-Z inherits another, it merges the definitions in the state schema, initial state schema and any common-named operations of the inherited class with its own. By merge, we mean that it forms the union of the declarations and conjunction of the predicates of the respective schemas. Hence, we can enhance class *Client* as in Example 3 as follows.

$Client'$

$Client$

$down : \mathbb{F}\, Id$

INIT

$down = \varnothing$

$ChangeServer \mathrel{\widehat{=}} [id' \notin down]$

$Request \mathrel{\widehat{=}} [id! \notin down]$

$Reply \mathrel{\widehat{=}} [id? \notin down]$

An auxiliary variable *down* is added to model the set of servers which are currently down. Initially, this set is empty. We then add constraints to each of the operations. To *ChangeServer* we add the constraint that the new server is not a member of *down*. To *Request* and *Reply* we add the constraint that the value of the communicated (input or output) variable is not a member of *down*.

The action Z_1 can then be modelled as an operation which is enabled when the client is idle and results in the server the client is interacting with being added to the set *down*.

$Client_1$

$Client'$

Z_1

$\Delta(down)$

$client.state = idle$

$down' = down \cup \{client.server\}$

Using $Client_1$ it is then possible to reason about the ability of the client to adapt to an occurrence of Z_1. This can be done using standard reasoning techniques for state-based specifications, or using tools adapted for use with Object-Z. For example, we could encode *Client* in the notation of the SAL model checking tools as detailed in [19]. To show that *Client* is Z_1-adaptive, we would check that the following two Computational Tree Logic (CTL) formulae [8] hold. (**AG** p states that p always holds in every behaviour, **AF** p states that p eventually holds in every behaviour, and **E**(p **U** q) states that there exist a behaviour during which p holds until q holds.)

$$\mathbf{AG}(\,(state = idle \wedge server \in down) \Rightarrow \mathbf{AF}(server \notin down)\,)$$

This formula follows directly from Definition 5. The state after Z_1 occurs is $state = idle \wedge server \in down$. The set of legitimate states of *Client* is captured by $server \notin down$.

$$\forall\, i \in ID \bullet \mathbf{AG}(\,(server = i \wedge state = idle \wedge i \in down) \Rightarrow \mathbf{E}\ (i \in down\ \mathbf{U}\ i \in down \wedge server \notin down)\,)$$

This formula states that for all behaviours in which after Z_1 (here captured by the predicate $server = i \wedge state = idle \wedge i \in down$ for some $i \in ID$), there exists a behaviour in which $i \in down$ until $i \in down$ and $server \notin down$. This captures condition (9).

V. Discussion

Given our formal definition it is prudent to examine whether or not it does cover the usual (informal) notions of adaptivity that arise in the literature. The purpose of this section is to relate a representative selection of them, particularly those reflecting the agent paradigm, to the approach and formalisation adopted in this paper.

1. Adaptivity should allow change in system functionality.

The view is that, by adapting, a system or agent may offer new operations on new states. This is already covered by any formalism (including the one used here) which models a system or agent as a state machine. The new functionality is simply the result of the system moving to a new state where new operations are enabled. For example, suppose a system $S_1 = (Q_1, I_1, \Sigma_1, \delta_1)$ adapts to behave like the system $S_2 = (Q_2, I_2, \Sigma_2, \delta_2)$ as a result of undergoing an external action Z. The adapting system consisting of S_1 and S_2 is equivalent to a single system S whose states are formed by the (discriminated) union of the states of S_1 and S_2, whose initial states are I_1, and whose actions are those of the S_1 and S_2. Z is an action which is enabled in a set of states $E \subseteq S_1$ and results in a state in $R \subseteq I_2$.

2. Adaptivity should reflect improved response to change.

The argument is that adaptivity means also that when subsequently confronted with the same external action in the future, the system or agent adapts more quickly and efficiently. Ensuring that repeated occurrences of any external action Z require fewer steps to stabilisation is readily incorporated in our definition by conjoining with it the predicate that says: if a system is n-Z-adaptive (for some $n > 0$), then it is n'-Z-adaptive for a second occurrence of Z for $n' < n$. This can be be extended for further occurrences of Z with the obvious limitation that the number of steps required to adapt cannot decrease below 1.

3. Adaptivity should include self-organisation.

Self-organisation may be viewed, using the terminology of complex systems, as the autonomous convergence to attracting states. For example, an ant colony forages for food by its ants following a combination of antennation and pheromone trails, with some random movement built in. The result is that each trail forms an approximate geodesic between the nest and food supply. If a is a geodesic path, let $V(a)$ denote the neighbourhood of paths which approximate it in the sense that any path lying in $V(a)$ deviates from a by routine random movements (or ‘noise’). If a trail is broken (perhaps a rock falls on it) then after some exploration a new geodesic is established.

If the system states are captured by neighbourhoods of geodesics between the nest and food supplies, and an external action Z removes some area of space (that covered by the rock) then the colony adapts by converging to the attracting state consisting of a new geodesic. (Nondeterminism - or bifurcation as it would be called in this setting - may occur, since there may be more than one new geodesic.) Convergence is most certainly not immediate and is not even easy to estimate.

We capture self-organisation of a system by assuming that certain of its states, say $\{a_i \mid 0 \leq i < \alpha\}$, are the attracting states, by which we mean that each has a ‘domain of convergence’, $D(a_i)$: from any state in $D(a_i)$ convergence to a_i is automatic. We also assume, as in the example, that each a_i has a neighbourhood $V(a_i)$ of normal activity. Z changes the current state of a system to some member of some $D(a_i)$. The system then self-organises by ensuring that paths converge to those in $V(a_i)$.

In that setting, with that extra structure on the system, self-organisation is covered by extending our definition of adaptivity to require that legitimate states are members of $V(a_i)$ for some a_i. (That setting is simplified by assuming $V(a_i) = \{a_i\}$, though that is unrealistic in the ant example.)

4. Adaptivity should include self-optimisation.

According to one agent-based view, a system is adaptive if in response to a change in externally-set parameters (*i.e.*, global variables) the agents are able spontaneously and autonomously to perform calculations (viewed as optimising certain local variables) which result in the system returning to a desired state. This is more a means to achieve adaptivity than a definition of it. It suffices in this case to model the system by ensuring that the relevant local variables are captured when specifying each agent, and the agents' optimising calculation is included (perhaps as part of a cooperative action, or perhaps as an individual internal action).

5. Adaptivity should reflect evolutionary change.

The interaction between a system and its environment may be viewed as a two-person game. On its ith turn, the environment performs an action —according to its (game-theoretic) strategy— which we view as Z_i. There are then two senses in which the system may be thought to adapt. The first is that it is always able to respond by performing actions that return it to a legitimate state before the environment's next interaction. That accords with adaptivity as we have defined it.

The second is a stronger notion, requiring that the system adapt not merely to individual occurrences of the Z_i but to the strategy of which they are the manifestations over finite time. That is possible only approximately because in general the system is not able to learn the environment's strategy in only a finite number of interactions. This is related to learnability, to which we now turn.

6. Adaptivity should include (machine) learning.

Machine learning (see the introductory text by Mitchell [16] and the book on ‘reinforcement learning’ by Sutton and Barto [21]) provides an important paradigm of agent adaptivity. Typically, in supervised learning (positive and negative) examples of a concept Q are provided to enable subsequent approximate classification of specimens into those satisfying Q and those satisfying $\neg Q$. In other words, this kind of adaptivity is of the ‘evolutionary’ kind rather than of the kind triggered by a specific external event. So ‘learning Q’ can be viewed as an emergent property.

In our setting, the system is open because the training examples are provided not in advance by the system, but spontaneously by its environment. The set of legitimate states expresses approximate classification of Q (for example as formalised by Valiant in probably approximately correct

(PAC) learnability [22]). An external action Z corresponds to initialisation of the learning protocol. The agent 'learns Q' iff it reaches a legitimate state after such initialisation.

VI. Conclusion

In this paper, we have proposed a formal definition of adaptivity based on behaviour leading to a set of legitimate states. The definition is general allowing it to be used for a wide range of system designs and with a variety of modelling notations.

Most existing work on adaptive systems has concentrated on implementations. For example, Güdemann *et al.* [12] augment a MAS with an observer/controller (OC) to monitor an invariant that incorporates the agents' roles. When the OC detects failure of the invariant it calculates a new configuration. In our terms, the MAS can be modelled as a closed system whose legitimate states are those satisfying the invariant. The action Z is highly nondeterministic and simply results in the invariant being violated. Calculation of the new configuration ensures that the augmented MAS is adaptive to Z. Similarly, the designs proposed in [9], [13], [5] can be mapped to our approach.

Dolev and Herman [5] additionally consider probabilistic convergence to a legitimate state expressed in terms of 'stable' distribution of states. The consideration of states represented as distributions, and adaptivity as some form of probabilistic convergence, constitutes interesting and important future work.

Acknowledgements

This work was supported by Australian Research Council (ARC) Discovery Grant DP110101211 and the Macao Science and Technology Development Fund under the EAE project, grant number 072/2009/A3.

References

[1] M. Beek, C. Ellis, J. Kleijn, and G. Rozenberg. Synchronizations in team automata for groupware systems. *Computer Support Cooperative Work – The Journal of Collaborative Computing*, 12(1):21–69, 2003.

[2] L. de Alfaro and T. A. Henzinger. Interface automata. In *Ninth Annual Symposium on Foundations of Software Engineering (FSE)*, pages 109–120. ACM Press, 2001.

[3] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, **17**:643–644, 1974.

[4] M. d'Inverno and M. Luck. Development and application of a formal agent framework. In *Proceedings of the First IEEE International Conference on Formal Engineering Methods*, pages 222–231. IEEE Press, 1997.

[5] S. Dolev and T. Herman. Dijkstra's self-stabilizing algorithm in unsupportive environments. In *Proc. Fifth Workshop Self-Stabilizing Systems (WSS 2001)*, pages 67–81, 2001.

[6] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Natural Computing Series. Springer, 2003.

[7] C. A. Ellis. Team automata for groupware systems. In S. Hayne and W. Prinz, editors, *International ACM SIGGROUP Conference on Supporting Groupwork: The Integration Challenge*, pages 415–424. ACM Press, 1997.

[8] E. A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science*, pages 995–1072. Elsevier, 1990.

[9] I. Georgiadis, J. Magee, and J. Kramer. Self-organising software architectures for distributed systems. In *1st Workshop on Self-healing Systems (WOSS '02)*, pages 33–38, 2002.

[10] M. G. Gouda and T. Herman. Adaptive programming. *IEEE Transaction on Software Engineering*, 17(9):911–921, 1991.

[11] P. Gruer, V. Hilaire, A. Koukam, and K. Cetnarowicz. A formal framework for multi-agent systems analysis and design. *Expert System Applications*, 23(4):349–355, 2002.

[12] M. Güdemann, F. Nafz, F. Ortmeier, H. Seebach, and W. Reif. A specification and construction paradigm for organic computing systems. In *Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2008)*, pages 233–242. IEEE Computer Society Press, 2008.

[13] T. Hayes, N. Rustagi, J. Saia, and A. Trehan. The forgiving tree: A self-healing distributed data structure. *CoRR*, abs/0802.3267, 2008.

[14] A. Hunter and J. P. Delgrande. Iterated belief change: A transition system approach. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI05)*, pages 460–465, 2005.

[15] N. Lynch and M. Tuttle. An introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219–246, 1989.

[16] T. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[17] J. W. Sanders and G. Smith. Assuring adaptive behaviour in self-organising systems. In *Self-Organising and Self-Adaptive Systems Workshop (SASOW 2010)*, pages 172–177. IEEE Computer Society Press, 2010.

[18] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 2000.

[19] G. Smith and L. Wildman. Model checking Z specifications using SAL. In H. Treharne, S. King, M. Henson, and S. Schneider, editors, *International Conference of Z and B Users (ZB 2005)*, volume 3455 of *LNCS*, pages 87–105. Springer-Verlag, 2005.

[20] J. M. Spivey. *The Z Notation: a reference manual, second edition*. Prentice-Hall International, 1992.

[21] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[22] L. Valiant. A theory of the learnable. *Communications of the ACM*, 27, 1984.