

Testing Collaborative Agents Defined as Stream X-Machines with Distributed Grammars

Tudor Bălănescu¹, Marian Gheorghe², Mike Holcombe², and Florentin Ipate¹

¹ Faculty of Sciences, University of Pitesti, Romania

fipate@ifsoft.ro, balanesc@phobos.cs.unibuc.ro

² Department of Computer Science, Sheffield University, UK

{m.gheorghe,m.holcombe}@dcs.shef.ac.uk

Abstract. This paper presents a method for testing software collaborative agents. The method is based on the stream X- machine model using as basic processing relations sets of production rules and is built around an existing stream X- machine testing strategy already. The paper investigates different variants of the 'design for test' conditions required by this strategy and their applicability to the case of collaborative agents.

1 Introduction

Although in philosophical literature the notion of an agent as a cognitive subject has been around for a long time, in the last decade this area has also become a major topic of research within artificial intelligence and computer science [18]. As might be expected, within the latter areas, the concept of an agent generally has a more technical meaning, although there is no general consensus on its definition. *Software agents* may be classified according to their mobility or reaction to their environmental stimuli or the attributes they should exhibit. A number of three primary attributes has been identified [22]: *autonomy, learning and cooperation*. *Collaborative agents* emphasise autonomy - agents can operate on their own - and cooperation - agents have the ability to cooperate with others. Both theoretical aspects pointing out the logic underpinning the collaborative agents as well as the practical issues showing how they are applied to large-scale applications have been investigated [21]. In this context we outline the importance of evaluating collaborative agent systems. The problems related to the verification and validation of these systems to ensure they meet their functional specifications are still outstanding. A specific model of multi agent paradigm called *cooperating distributed grammar systems*, has been devised [8], investigated and developed [20]. In the context of formal software specification, the *X- machine* model introduced by Eilenberg [9] has been reconsidered [12] and proposed as a possible specification language for dynamic systems. Since then a number of further investigations have demonstrated that this idea is of great potential value for both fundamental and practical issues in software engineering [14]. An important aspect approached for *stream X- machines*, a variant of the more general X- machine model, has been that of devising a test method

that has been proved to find *all faults* of the implementation [15]. Cooperating distributed grammar systems (CDGS) and stream X- machines (SXM) share several common properties as pointed out in [8]:

- the information needed to solve a problem is partitioned into separate and independent knowledge sources (the production rule sets of CDGS and the processing relations/functions of SXM);
- the global database on which various knowledge sources may make changes (the common sentential form of CDGS and the global memory of SXM);
- the control of the opportunistic responses of the knowledge sources to make modifications in the global database (various derivation control strategies of CDGS and the underlying automata of SXM).

It is worth noting that the first and last of these properties emphasise the autonomous behaviour that is expected for the agents, the second issue stresses the cooperation aspects requested for multi-agent systems. The similar properties of both models lead to the definition of the *X- machine model with underlying distributed grammars* (SXMDG) ([4], [10]). A further step towards testing SXMDG has been made by investigating deterministic SXMDG for some specific derivation strategies and regular rules [5]. As testing is crucial for validating an implementation against the specification, this paper will investigate further the testing aspects of SXMDG in the context of building reliable collaborative software agents. The SXMDG model benefits from both components involved and exposes new characteristics of interest: a) the precision and simplicity of the production rules and the derivation strategies of CDGS is used in defining the processing relations of SXM; b) the well established computational power of various CDGS can be transferred to SXMDG ([4], [10]); c) the already proven power of the SXM model in testing the behaviour of an implementation against its specification can be transferred and adapted, as we are going to show in this paper, to SXMDG; d) the property of SXMDG models, inherited from SMSs, of using basic relations/functions that process an input symbol in order to yield a suitable output symbol makes the model suitable for modelling collaborative agents as well as reactive agents [17]; e) various grammar types are used in defining different natural activities that can lead to a promising way of organising system design in a formal manner in the context of SXMDG. The paper presents a method of testing stream X- machines with underlying distributed grammars by adapting the general theory of stream X- machine testing to the characteristics of this new model. In particular, the paper defines a set of 'design for test' conditions which ensure that the behaviour of collaborative agents is under control and observable at any moment. These conditions may sometimes require that extra information is added to the system, but any testing method that can find *all faults* will impose certain restrictions on the behaviour of the system under testing [6]. The paper is structured as follows: section 2 introduces the main concepts concerning cooperating distributed grammar systems, stream X- machines and stream X- machines based on distributed grammars; section 3 introduces the concepts related to stream X- machine testing; the results that refer to the application of the 'design for test' conditions to stream X- machines

based on distributed grammars are presented in section 4; conclusions are drawn in the last section.

2 Basic Definitions

Only the basic definitions and notations will be provided in this section. For further details concerning CDGS and SXM the reader may refer to [8] and [14], respectively.

Definition 1. A CDGS is a system $\Gamma = (N, A, S, P_1, \dots, P_n)$ where N is the set of nonterminal symbols, A is the set of terminal symbols, S is the start symbol, and $P_i, 1 \leq i \leq n$ are the sets of production rules.

Note 1. Let us denote by DM the set of all the derivation modes, i.e.

$$DM = \{t\} \cup \{\leq r, = r, \geq r \mid r \geq 1\}$$

Definition 2. A SXM is a system $X = (\Sigma, A, Q, M, \Phi, F, q_0, T, m_0)$ where Σ and A are finite sets called the input set and output set respectively; Q is the finite set of states, M is, the possibly infinite, set of memory symbols; $T \subseteq Q$, is the set of final states and $q_0 \in Q$ is the initial state; $m_0 \in M$ is the initial memory value; F is a partial function called the transition function defined as $F : Q \times \Phi \rightarrow 2^Q$ and Φ is the finite set of processing relations; each processing relation is defined as $\varphi : M \times \Sigma \rightarrow 2^{A^* \times M}$

Definition 3. A SXMDG is, for each $d \in DM$, a system

$$X_d = (\Sigma, A, N, Q, M, P, \Phi_d, F, q_0, T, m_0)$$

where $\Sigma, A, Q, M, F, q_0, T, m_0$ are as in Definition 2 and N is a finite set called the set of nonterminal symbols such that $\Sigma \subseteq N \cup A$; $M = (N \cup A)^*$, P is a set of n sets of production rules defined over $N \cup A, P = \{P_1, \dots, P_n\}$ and Φ_d is defined as being $\Phi_d = \{\varphi_1, \dots, \varphi_n\}$ with each φ_i associated to $P_i \in P$ as follows

$$(x, m') \in \varphi_i(m, \sigma) \text{ if and only if } m\sigma \xRightarrow{d}_{P_i} x m'$$

$\sigma \in \Sigma, x \in A^*, m, m' \in M$ and m' does not begin with a symbol from A . As only nonempty output strings will be expected from each φ_i , in what follows the set A^* will be replaced by $A^+ = A^* - \{\lambda\}$.

An SXMDG acts as a translator which processes the input strings and works out sequences of symbols according to the rules belonging to the sets P_i .

Note 2. For a language $L \subseteq \Sigma^*$, and an SXMDG X_d we denote by $f_d(L)$ the language computed by X_d when L is the input set. For any string $x \in f_d(L)$ there exist: an input string $\sigma_1 \dots \sigma_p \in L$ where $\sigma_i \in \Sigma$ for all $1 \leq i \leq p$; p output strings $x_1, \dots, x_p, x_i \in A^+$ for all $1 \leq i \leq p$ and $x = x_1 \dots x_p$; p processing relations $\varphi_{j_1}, \dots, \varphi_{j_p}$; p states q_1, \dots, q_p with $q_p \in T$; p memory values m_1, \dots, m_p with $m_p = \lambda$; such that $q_i \in F(q_{i-1}, \varphi_{j_i})$ and $(x_i, m_i) \in \varphi_{j_i}(m_{i-1}, \sigma_i)$ for all $1 \leq i \leq p$.

Example 1. Let us consider the following SXMDG

$$X_{=2} = (\Sigma, A, N, Q, M, P, \Phi_d, F, q_0, T, m_0)$$

where $\Sigma = \{C, D, c\}$, $A = \{a, b, c\}$, $N = \{B, C, B', C', D\}$, $P = \{P_1, P_2, P_3\}$, with $P_1 = \{B \rightarrow aB', C \rightarrow bC'\}$, $P_2 = \{B' \rightarrow aB, B' \rightarrow a, C' \rightarrow bC, C' \rightarrow b\}$, $P_3 = \{D \rightarrow aD, D \rightarrow a\}$; $\Phi = \{\varphi_1, \varphi_2, \varphi_3\}$, with $\varphi_1(B, C) = (a, B'bC')$, $\varphi_1(Bb^rCc^s, c) = (a, B'b^{r+1}C'^{s+1})$, $r, s \geq 1$, $\varphi_2(B'b^rC'^{s-1}, c) = \{(a, Bb^{r+1}C^s), (ab^{r+1}c^s, \lambda)\}$, $r, s \geq 1$, $\varphi_3(\lambda, D) = (a^2, \lambda)$; $Q = \{1, 2, 3\}$, $F(1, \varphi_1) = 2$, $F(2, \varphi_2) = 1$, $F(1, \varphi_3) = 3$, $T = \{3\}$, $q_0 = 1$, $m_0 = B$. For the input set $L = \{Cc^{2n-1}D \mid n \geq 1\}$ the language computed by $X_{=2}$ is $f_{=2} = \{a^{2n}b^{2n}c^{2n-1}a^2 \mid n \geq 1\}$.

Indeed, for getting $a^{2n}b^{2n}c^{2n-1}a^2$ we use the input string $Cc^{2n-1}D$, the processing relations $(\varphi_1\varphi_2)^n\varphi_3$ and the states $q_{2k-1} = 2$, $1 \leq k \leq n$; $q_{2k} = 1$, $0 \leq k \leq n$; $q_{2n+1} = 3$; such that for the input strings $\sigma_1 = C$; $\sigma_i = c$, $2 \leq i \leq 2n$; $\sigma_{2n+1} = D$; the following output symbols $x_i = a$, $1 \leq i \leq 2n - 1$; $x_{2n} = ab^{2n}c^{2n-1}$; $x_{2n+1} = a^2$; and memory values $m_0 = B$; $m_{2k-1} = B'b^{2k-1}C'^{2k-2}$, $1 \leq k \leq n$; $m_{2k} = Bb^{2k}Cc^{2k-1}$, $1 \leq k \leq n - 1$; $m_{2n} = m_{2n+1} = \lambda$, are obtained. As it may be seen the SXMDG is powerful enough as it translates a regular language into a non-context-free language by using regular rules.

3 Testing SXM

This section introduces concepts related to the testing method based on SXMs. The idea behind this method is to compare the SXM model of the implementation with the SXM specification of the same system ([15], [14]) and to generate a test set whose successful application will establish that these two machines have identical behaviour. For a SXM $X = (\Sigma, A, Q, M, \Phi, F, q_0, T, m_0)$, the *finite state machine associated* (FSM) with X is defined as $A(X) = (\Phi, Q, F, q_0)$. Note that the set Φ in the definition of $A(X)$ contains in fact labels associated with the partial relations belonging to Φ . For simplicity no distinction is made between the actual relations and the associated labels. The idea of the SXM testing method is to prove that the two stream X- machines (one representing the specification and the other the implementation) have identical input- output behaviour by showing that their associated automata are equivalent. Thus, the generation of the test set used to prove the equivalence of the two X- machines involves two steps: a) the generation of a set of sequences of basic processing relations that tests the equivalence of the two associated automata; b) the translation of the basic processing relations into sequences of inputs. The generation of the set of processing relations (denoted by Z) is based on Chow's W- method [7] that constructs a test set from a FSM specification. In the case of the SXM model, this is the FSM associated with the specification. The generation of Z consists mainly of the generation of two sets. The first set $C \in \Phi^*$, called a *state cover* of $A(X)$, has the property that for any state $q \in Q$ there is $p \in C$ such that p is a path in $A(X)$ from the initial state q_0 to q . The next set $W \in \Phi^*$, called a *characterisation set* of $A(X)$, has the property that if q and q' are two arbitrary

distinct states then there is a $p \in W$ such that p distinguishes between q and q' . The string p is said to distinguish between q and q' if p is a path in $A(X)$ that starts in q and p is not a path that starts in q' or vice-versa. A FSM with these properties is called *minimal*. This property does not restrict the generality of our discussion since for any automaton there is a minimal FSM equivalent to it [11]. If the difference between the number of states of the implementation and the number of states of the specification can be estimated and is denoted by k then the set Z we are looking for is: $Z = C(\Phi^{k+1} \cup \Phi^k \dots \Phi \cup \{\lambda\})W$. Once the set Z has been set up it will be translated into a set of sequences of inputs that can be applied to the two machines. This is the second step of our construction and a *test function* is defined for this purpose. The scope of a test function is to test whether a certain path exists or not in the machines using appropriate input symbols. A test function $test : \Phi^* \rightarrow \Sigma^*$ is constructed as follows:

- $test(\lambda) = \lambda$
- for any $p_s = \varphi_1 \dots \varphi_s \in \Phi^*, s \geq 1$, if $p_{s-1} = \varphi_1 \dots \varphi_{s-1}$ is a path of $A(X)$ that starts in q_0 then $test(p_s)$ is a sequence of inputs that exercise p_s from the initial memory m_0 ; otherwise $test(p_s) = test(p_{s-1})$.

In other words, for any sequence of basic processing relations p , $test(p)$ exercises the longest prefix of p that is also a path of the machine state plus the function that follows after this prefix, if any. Note that a test function is not uniquely determined, a stream X- machine may have many test functions. Once the set Z and the function $test$ have been determined the test set is $Y = test(Z)$. In order to compute the test function the SXM should be *deterministic* and satisfy some properties called '*design for test*' conditions. A deterministic SXM is one in which for any $\varphi, \varphi' \in \Phi$ and $q \in Q$ if $F(q, \varphi) \neq \emptyset$ and $F(q, \varphi') \neq \emptyset$ then $dom \varphi \cap dom \varphi' = \emptyset$, where $dom \varphi$ denotes the domain of φ . The design for test conditions ensure that the behaviour of the machines will have to be detectable under all conditions. These are *output distinguishability* and *memory completeness* [14] or *weak output distinguishability* and *strong memory completeness* [2]. A SXM is *output-distinguishable* if for any memory value m , input symbol σ and any two processing relations $\varphi, \varphi' \in \Phi$ if there exist an output string $x \in A^+$ and memory values $m_1, m_2 \in M$ with $(x, m_1) \in \varphi(m, \sigma)$ and $(x, m_2) \in \varphi'(m, \sigma)$ then $\varphi = \varphi'$ and $m_1 = m_2$. A SXM is *memory-complete* if for any basic processing relation φ and any memory value m there exists an input σ such that $(m, \sigma) \in dom \varphi$. A SXM is *weak output-distinguishable* if for all $\varphi \in \Phi$ there exists an input symbol $\sigma_\varphi \in \Sigma$ such that for all $\varphi' \in \Phi$ it results $\varphi = \varphi'$ and $m_1 = m'_1$ if there exist $m \in M$ and $x \in A^+$ such that $(x, m_1) \in \varphi(m, \sigma_\varphi)$, $(x, m'_1) \in \varphi'(m, \sigma_\varphi)$. A SXM is *strong memory-complete* if it is weak output-distinguishable and if for any $\varphi \in \Phi$, $m \in M$ it follows that (m, σ_φ) is in the domain of φ . Furthermore, the (weak) output distinguishability and (strong) memory completeness properties can be relaxed without affecting the validity of the testing method [16].

Definition 4. Let $\Sigma_1 \subseteq \Sigma$ be a subset of the input alphabet of a given SXM X . We define by $V(m_0, \Sigma_1) = \{m_0\} \cup \{m \mid \exists \sigma_1 \dots \sigma_k \in \Sigma_1^+, \exists \varphi_1 \dots \varphi_k \in \Phi^+\}$

and $(x, m) \in (\varphi_1 \dots \varphi_k)(\sigma_1 \dots \sigma_k, m_0)$ for some $x \in A^+$. Then we define (weak) output distinguishability and (strong) memory completeness with respect to $V(m_0, \Sigma_1)$.

It is easy to see that a SXM that is (weak) output distinguishable and (strong) memory complete is also (weak) output distinguishable and (strong) memory complete with respect to any $V(m_0, \Sigma_1) \subseteq M$. However, the SXM testing method still remains valid if the usual conditions are replaced by similar conditions with respect to $V(m_0, \Sigma_1)$, where $\Sigma_1 = \Sigma$ in the case of memory-completeness and output-distinguishability and $\Sigma_1 = \Sigma_\Phi = \{\sigma_\varphi \mid \varphi \in \Phi\}$ in the case of strong memory-completeness and weak output-distinguishability [16]. That is, the sufficient conditions in which the testing method is valid are either memory-completeness and output-distinguishability with respect to $V(m_0, \Sigma)$ or strong memory-completeness and weak output-distinguishability with respect to $V(m_0, \Sigma_\Phi)$. For the example presented the sets C, W are $C = \{\lambda, \varphi_1, \varphi_3\}$, $W = \{\varphi_2, \varphi_3\}$. For $k = 0$, it follows $Z = C(\Phi \cup \{\lambda\})W$, and thus

$$Z = \{\lambda, \varphi_1, \varphi_3\} \{\lambda, \varphi_1, \varphi_2, \varphi_3\} \{\varphi_2, \varphi_3\} =$$

$$\{\varphi_2, \varphi_1\varphi_2, \varphi_2\varphi_2, \varphi_3\varphi_2, \varphi_1\varphi_1\varphi_2, \varphi_1\varphi_2\varphi_2, \varphi_1\varphi_3\varphi_2, \varphi_3\varphi_1\varphi_2, \varphi_3\varphi_2\varphi_2, \varphi_3\varphi_3\varphi_2,$$

$$\varphi_3, \varphi_1\varphi_3, \varphi_2\varphi_3, \varphi_3\varphi_3, \varphi_1\varphi_1\varphi_3, \varphi_1\varphi_2\varphi_3, \varphi_1\varphi_3\varphi_3, \varphi_3\varphi_1\varphi_3, \varphi_3\varphi_2\varphi_3, \varphi_3\varphi_3\varphi_3\}$$

Please note that $X_{=2}$ is deterministic and (weak) output distinguishable but is not (strong) memory-complete. There is a further problem that appears when the application of the test set is evaluated. For any sequence $\sigma_1 \dots \sigma_n$ in the test set, we have to establish that the outputs produced for any of the inputs $\sigma_1, \dots, \sigma_n$ to the implementation coincide to those in the specification. This is achieved if either the outputs produced by all prefixes $\sigma_1 \dots \sigma_i$, $1 \leq i \leq n$, are recorded or the machine meets an additional property, *output-delimited property*, that allows all these intermediary outputs to be deduced from those produced by the overall sequence $\sigma_1 \dots \sigma_n$. A SXM is *output delimited* if for any two strings of relations $\varphi_1 \dots \varphi_j$ and $\varphi'_1 \dots \varphi'_j \in \Phi^*$, any two output strings $x_1 \dots x_j$ and $y_1 \dots y_j$ with each $x_i, y_i \in A^+$, from $x_1 \dots x_j = y_1 \dots y_j$, $(x_1, m_1) \in \varphi_1(m_0, \sigma_{\varphi_1})$, $(y_1, m'_1) \in \varphi'_1(m_0, \sigma_{\varphi'_1})$, $(x_i, m_i) \in \varphi_i(m_{i-1}, \sigma_{\varphi_i})$, $(y_i, m'_i) \in \varphi'_i(m'_{i-1}, \sigma_{\varphi'_i})$ for all $2 \leq i \leq j$ it follows that $x_i = y_i$, for all $1 \leq i \leq j$.

Thus, for an output delimited type an arbitrary path produces the same output as $\varphi_1 \dots \varphi_j$ on the same input sequence $\sigma_{\varphi_1} \dots \sigma_{\varphi_j}$ only if it produces the same output as φ_i at each stage i , $1 \leq i \leq j$. Any type $\Phi = \{\varphi : M \times \Sigma \longleftarrow A \times M\}$ containing only relations producing single symbols is output delimited.

4 Testing SXMDG

(Weak) output distinguishability and (strong) memory completeness as well as their more relaxed counterparts are addressed in this section. Output-distinguishability will take place for SXMDG having a LL(1) like property [1]. For a string x , $first_1(x)$ denotes the set $\{a \mid a \in A, \exists r \geq 1, x \Longrightarrow^r ay\}$, $r \in DM$.

Lemma 1. *Any SXMDG X_d , $d \in DM'$ with the property that for any two rules $S \rightarrow x_1 \in P_i$, $S \rightarrow x_2 \in P_j$, $i \neq j$ it follows $first_1(x_1) \cap first_1(x_2) = \emptyset$, then X_d is output-distinguishable.*

Proof. If two distinct relations φ_i and φ_j , $i \neq j$, are defined for the same memory value and the same input then distinct output values will be generated.

According to lemma 1 the SXMDG $X_{=2}$ defined in example 1 is output-distinguishable. For a given derivation mode $d \in DM$, for each set of productions P_i we denote by $T_i^d = \{B \mid B \in N \text{ and if } B \Longrightarrow_{P_i}^d \alpha \text{ then } \alpha \in A^+\}$ the set of all nonterminals producing only terminal strings when the derivation mode d is applied. For a nonterminal $B \in N$ and a set of production rules P_i we denote by $L^d(B, P_i) = \{x \mid x \in A^+, B \Longrightarrow_{P_i}^d x\}$ the language generated by B applying production rules from P_i and using only once the derivation mode d .

Lemma 2. *Let X_d be an SXMDG and consider $\Sigma_1 = T_1^d \cup \dots \cup T_n^d$. If $m_0 \in \Sigma$, then $V(m_0, \Sigma_1) = \{m_0, \lambda\}$*

Proof. Direct, from the definition 4.

Lemma 3. *Let us consider an SXMDG X_d having n sets of production rules P_1, \dots, P_n and terminal initial memory value ($m_0 \in A^*$). If there exist n non-terminals $B_1 \in T_1^d, \dots, B_n \in T_n^d$ such that $L^d(B, P_i) \cap L^d(B, P_j) = \emptyset$ for $i \neq j$, then:*

1. *the associated type $\Phi_d = \{\varphi_1, \dots, \varphi_n\}$ is weak output distinguishable with respect to the memory $\{m_0, \lambda\}$*
2. *if all P_i contains only regular rules and the derivation mode d is $= r$ then the type Φ is output delimited.*

Proof. 1. It is sufficient to consider $\sigma_{\varphi_i} = B_i$ and to apply Lemma 2.

2. If d is $= r$ then all the output strings have the length k . Such types are output delimited [2].

Example 2. Let us consider

$$P_1 = \{S \rightarrow aS, S \rightarrow aA, A \rightarrow bB, B \rightarrow c\},$$

$P_2 = \{S \rightarrow cS, S \rightarrow cB, B \rightarrow bA, A \rightarrow a, A \rightarrow bB, B \rightarrow c\}$. For the derivation mode $= 2$ we have: $T_1^{=2} = \{A\}$, $T_2^{=2} = \{A, B\}$. We can choose $B_1 = A, B_2 = B$.

If $m_0 = a$ then according to Lemma 2 we have $V(a, \{A, B\}) = \{a, \lambda\}$. The type $\Phi = \{\varphi_1, \varphi_2\}$ is also output delimited, as stated by Lemma 3.

The memory completeness restriction imposes important hard conditions for the processing relations of SXMDG.

Lemma 4. *If X_d , $d \in DM'$ is a (strong) memory-complete SXMDG then for any $P_i \in P$, $B \in N$*

- *there exists a terminal derivation $B \Longrightarrow_{P_i}^t x$, $x \in (A \cup N)^*$*

– there exist $\sigma \in \Sigma$ and a derivation $B\sigma \Longrightarrow_{P_i}^d x, x \in (A \cup N)^*, d \in DM - \{t\}$

Proof. According to (strong) memory-completeness restriction any memory value m and an input σ should have a derivation in any of P_i components starting from $m\sigma$. In particular m may be any nonterminal. In the case $d = t$ each nonterminal should have a terminal derivation and when $d \neq t$ the string $m\sigma$ should start a d derivation in P_i .

Note 3. The processing relation φ_1 defined by the SXMDG $X_{=2}$ presented in example 1, for any input σ and memory $H \in N, H \neq B$ and $H \neq C$ does not have (m, σ) in its domain. This result shows that the solution of extending the definition of the processing relations by considering new input symbols is no longer valid in this case [14]. Fortunately, as we have shown in the previous section, a weaker condition is sufficient. This refers only to those memory values that may be computed starting from the initial value m_0 (lemma 3). Furthermore, we will use the variant of design for test conditions that suit best our example, these are strong memory-completeness and weak output distinguishability with respect to some $V(m_0, \Sigma_1)$.

Note 4. These design for test conditions can be met as follows: a) add to the production rule sets a number of extra rules so that the conditions of lemma 3 are satisfied (NB: this may change the resulting language); in the case of example 1 considered the rules $B' \rightarrow a \in P_1, B \rightarrow a \in P_2$ are added; consequently the inputs associated with the processing relations will be the next nonterminals: $\sigma_{\varphi_1} = B, \sigma_{\varphi_2} = B', \sigma_{\varphi_3} = D$; b) consider (for testing purposes) $m_0 = \lambda$ and $V(\lambda, \{B, B', D\}) = \{\lambda\}$; then it is easy to see that the SXMDG is strong memor-complete and weak output-distinguishable with respect to $V(\lambda, \{B, B', D\})$.

With these observations in place an algorithm for testing SXMDG in the case $= r, r \geq 1$, may be described as follows:

- compute the set Z according to Chow’s method as stated in the previous section;
- check whether the conditions described in lemma 3 are fulfilled by the current SXMDG, $X_{=r}$; if these conditions are not satisfied then add the necessary rules according to note 4;
- compute the input test set $test(Z)$.

The set $test(Z) = \{test(y_z) \mid z \in Z\}$, where for $z = \varphi_1 \dots \varphi_n, y_z = \varphi_1 \dots \varphi_k$, where $k = n$ if z is a path of the SXM and $k = m + 1$, otherwise, where m is such that $\varphi_1 \dots \varphi_m$ is the longest prefix of z that is also a path of the SXM. Then $test(y_z) = \sigma_{\varphi_1} \dots \sigma_{\varphi_k}$.

5 Conclusions

In this paper we present a method for testing collaborative software agents modelled by SXMDG systems. The method is a variant of the method originally developed for SXM models. A number of alternative design for test conditions have been considered. The paper shows that the "traditional" design for testing conditions (memory-completeness and output-distinguishability) are not applicable to SXMDS models and that the "usual" recommendation of extending the definition of the processing relations so as to accept new input values is not suitable to this case. On the other hand, the paper considers slightly different design for test conditions (strong memory-completeness and weak output-distinguishability with respect to $V(m_0, \Sigma_1)$) that can be easily imposed on a SXMDS. These conditions seem to be coherent with the behaviour of collaborative agents as their status must be observable and controllable at some discrete moments, defined by the states of the machine. Some problems remain to be further investigated. On the theoretical side, the computational power of the SXMDG having production rules with the property stated in lemma 3 need to be established. From a practical point of view it will be necessary to apply the method developed in this paper to some complex collaborative agent systems in order to prove its usefulness and suitability. A number of approaches dealing with agent models ([13], [17]) outline the parallel behaviour of the agents have been based around communicating stream X- machine concepts [3]. We think that some of the models used in biology or bio-informatics and dealing with communities of agents may be better approached by integrating a membrane like computation [19] with X- machine systems. Some work is in progress and will be reported in a further paper.

References

1. A. Aho, J. Ullman, *The Theory of Parsing, Translation and Compiling*, Vol. I: Parsing, Prentice-Hall, Englewood Cliffs, N.J, 1972.
2. T. Balanescu, Generalized stream X-machines with output delimited type, *Formal Aspects of Computer Science*, accepted, 2001.
3. T. Balanescu, T. Cowling, H. Georgescu, M. Gheorghe, M. Holcombe, C. Vertan, Communicating Stream X- Machines are more than X- Machines, *J. of Universal Computer Sci.*, 5, 9(1999), 492-507.
4. T. Balanescu, H. Georgescu, M. Gheorghe, Stream X- machines with underlying Distributed Grammars, submitted 2001.
5. T. Balanescu, M. Gheorghe, M. Holcombe, A Subclass of Stream X-machines with Underlying Distributed Grammars, in *Proceedings of the International Workshop Grammar Systems 2000*, R. Freund and A. Kelemenova (eds), Bad Ischl, Austria, 2000, 93-111.
6. G. Bernot, M. Gaudel, B. Marre, Software testing based on formal specifications: a theory and a tool, *Software Engineering Journal*, 6(1991), 387-405.
7. T.S. Chow, Testing software design modelled by finite-state machines, *IEEE Transactions on Software Engineering*, 4, 3(1978), 178-187.

8. E. Csuhaj-Varju, J. Dassow, J. Kelemen, Gh. Paun, *Grammar Systems. A Grammatical Approach to Distribution and Cooperation*, Gordon and Breach, London, 1994.
9. S. Eilenberg, *Automata, languages, and machines*, Vol. A, Academic Press, 1974.
10. M. Gheorghe, Generalized Stream X- machines and Cooperating Distributed Grammar Systems, *Formal Aspects of Computer Science*, accepted, 2001.
11. A. Gill, *Introduction to the Theory of Finite-State Machines*, McGraw-Hill, 1962.
12. M. Holcombe, X- machines as a basis for dynamic system specification, *Software Engineering Journal* 3(1988), 69-76.
13. M. Holcombe, Computational models of cells and tissues - machines, agents and fungal infection, presented at UCL, London 15-16 Feb, 2001, submitted to *Briefings in Bioinformatics*.
14. M. Holcombe, F. Ipate, *Correct Systems: Building a Business Process Solution*, Springer Verlag, 1998.
15. F. Ipate, M. Holcombe, An Integration Testing Method That is Proved to Find all Faults, *Intern. J. Computer Math*, Vol 69 (1997), 159-178.
16. F. Ipate, M. Holcombe, Generating test sets from non-deterministic stream X-machines, *Formal Aspects of Computer Science*, accepted, 2001.
17. P. Kefalas, Modelling an Agent Reactive Architecture with X- Machines, *Technical Report*, CITY Liberal Studies, 2000.
18. *Formal Models of Agents*, (J-J. Ch Meyer, P-Y. Schobbens eds), Springer Verlag, LNAI 1760, 1999.
19. Gh. Paun, Computing with membranes, *Journal of Computer and System Sciences*, 61, 1 (2000), 108-143.
20. *Grammatical models of multi-agent systems*, (Gh. Paun, A. Salomaa eds), Topics in Computer Mathematics 8, Gordon and Breach Science, Amsterdam, 1999.
21. A. S. Rao and M. P. Georgeff, BDI Agents: From Theory to Practice, in *Proceedings of the 1st International Conference on Multi-Agent Systems*, San Francisco, USA, 1995, 312-319.
22. M. Wooldridge, N.R. Jennings, Intelligent Agents - Theory and Practice, *Knowledge Engineering Review*, 10, 2(1995), 115-152.