Dynamic Message Sequence Charts

Martin Leucker[1,2⋆], P. Madhusudan[2⋆⋆], Supratik Mukhopadhyay[2]

[1] Dept. of Computer Systems, Uppsala University, Sweden
[2] Dept. of Computer and Information Science, University of Pennsylvania, USA
{leucker, madhusudan, supratik}@cis.upenn.edu

Abstract. We introduce a formalism to specify classes of MSCs over an unbounded number of processes. The formalism can describe many interesting behaviours of dynamically changing networks of processes. Moreover, it strictly includes the formalism of Message Sequence Graphs studied in the literature to describe MSCs over a fixed finite set of processes. Our main result is that model-checking of MSCs described in this formalism against a suitable monadic-second order logic is decidable.

1 Introduction

In the early stages of design of a communication system, an emerging practice is to specify the system by describing the possible scenarios. A popular notation to describe such scenarios is that of message sequence charts (MSCs). This ITU standardized notation ([8]) describes a snap-shot of messages sent and received by various components of a distributed system in graphical notation.

There is now a growing interest in analyzing MSCs and applying the formal analysis techniques of model-checking to them. The motivation is that a designer could specify the scenarios using a collection of MSCs and verify them against certain requirements. Detection of errors at such an early stage in design can have considerable pay-offs.

In [1], the authors studied model-checking of message-sequence graphs (MSGs) against linear-time specifications. MSGs are a compact way of specifying an infinite collection of MSCs as a regular combination of atomic MSCs using concatenation, choice and repetition. Since an MSC describes a partially-ordered behaviour, a natural question to ask is whether all the linearizations of all the MSCs in the collection satisfy the linear specification. As shown in [1], this problem is undecidable unless the MSGs are heavily restricted.

In [5], the model-checking problem for MSCs was studied for structural specifications expressed in monadic second-order logic (MSO). MSO specifications describe properties of the partial-order defined by the MSC rather than of their linearizations. It was shown that the model-checking problem is decidable for MSGs, without any restriction on them. The idea that structural specifications can make model-checking effective has been extended for a larger class of MSC languages than MSGs [6].

⋆ supported by DARPA MOBIES F33615-00-C-1707, NSF CCR-9988409, NSF CISE-9703220, and the European Research Training Network on "Games"
⋆⋆ supported by NSF awards CCR99-70925 and ITR/SY 0121431

All the classes of MSC languages considered have, however, a serious limitation—they describe behaviours of only a *fixed* finite set of processes. While this is sufficient for many protocols, there are a number of domains where the number of processes cannot be considered fixed. Prime examples of such protocols are those that arise in the areas of mobile computing and ad-hoc networks. For example, a mobile phone may interact with an arbitrary number of transmitters and the description of switching from one to the other is naturally modelled only with a varying number of transmitters.

In this paper, we propose a formalism to describe MSC languages where the underlying set of processes need not be bounded. The formalism is given by a grammar which allows the user to concatenate MSCs (as in an MSG) and also gives the ability to *split* processes into two teams. Both teams can then independently interact with new processes that are spawned, and then join again, upon which they can continue interacting with each other. In order to contain ourselves in a decidable fragment, we constrain that at any point, the number of interesting processes for which the user is specifying a behaviour is fixed. Our main result is that monadic-second order specifications can be model-checked effectively for such a set of dynamic MSCs specified in this grammar.

Our formalism is quite natural and can describe many interesting behaviours involving unboundedly many processes. Like process calculus [7], it uses recursion to describe scenarios over an arbitrary number of processes. Also, our formalism is a proper extension of that of MSGs, which can be formulated in our grammar easily.

The main technical arguments for the decidability of model-checking stem from the rich theory of decidable classes of graphs against MSO formulas studied by Courcelle and others (see [3, 2]). The classes of graphs we generate are similar in spirit to that of series-parallel graphs, for which similar decidability results are known. We see the main contribution of this paper as that of identifying a meaningful and natural class of MSC languages that allow depiction of scenarios of unboundedly many processes, and applying existing techniques in order to verify them against powerful structural specifications.

2 Fork-and-join MSCs and Monadic Second-Order Logic

Let P, P', ... denote finite sets of *process names* (or just *processes* in short). We let $p, q, p_1, p', q', p'_1, \ldots$ range over processes. A *message alphabet* is a finite set Γ_M. Its elements are usually denoted by $a, a_1, \ldots$. For notational convenience, we fix Γ_M for the rest of the paper. For a set of processes P, let $Ac(P) := \{(p!q, a), (p?q, a) \mid p, q \in P, a \in \Gamma_M\}$ denote the set of *actions* over P. An action $(p!q, a)$ should be read as "p sends a message a to process q", and $(q?p, a)$ represents the corresponding receive action, which is then executed by process q. We denote by $Ac_p(P)$ the set of actions that $p \in P$ *participates* in, defined as $Ac_p(P) = \{(p!q, a), (p?q, a) \mid q \in P, a \in \Gamma_M\}$.

A message-sequence chart (MSC) is a partially ordered set of send and receive events, with a matching function that identifies the send events with corresponding receive events:

Definition 1. *Let P be a set of processes. A* Message Sequence Chart (MSC) *over P is a tuple $m = (P, E, \{\leq_p\}_{p\in P}, \lambda, \mu)$ where*

- *E is a finite set of* events*;*
- *$\lambda : E \to Ac(P)$ is a* labelling function *that identifies for each event an action. Let $E_p = \{e \in E \mid \lambda(e) \in Ac_p(P)\}$ denote the set of events which p participates in. Also, let $E_{\mathcal{S}} = \{e \in E \mid \lambda(e) = (p!q, a)$ for some $p, q \in P, a \in \Gamma_M\}$ and $E_{\mathcal{R}} = \{e \in E \mid \lambda(e) = (p?q, a)$ for some $p, q \in P, a \in \Gamma_M\}$ denote the set of* send *and* receive events *of E, respectively;*
- *$\mu : E_{\mathcal{S}} \to E_{\mathcal{R}}$ is a* matching function *that associates with each send event its corresponding receive event. Therefore, we require μ to be a bijection and for $e, e' \in E$, if $\mu(e) = e'$ then $\lambda(e) = (p!q, a)$ and $\lambda(e') = (q?p, a)$ for some $p, q \in P, a \in \Gamma_M$;*
- *$\leq_p$ is a total order on E_p for each $p \in P$. Let $\widehat{\leq} = (\bigcup_{p\in P} \leq_p) \cup \{(e, e') \mid e, e' \in E$ and $\mu(e) = e'\}$. Let $\leq = (\widehat{\leq})^*$ be the reflexive and transitive closure of $\widehat{\leq}$. Then $\leq$ denotes the* causal ordering *of E in the MSC and we require it to be a partial order on E.*

Note that in the above definition, we do not require E_p to be nonempty, for any $p \in P$.

An MSC m is considered equivalent to an MSC m' if it differs from m' only on the process and event sets—i.e. if the process and event sets of m can be relabelled (with appropriate relabelling of the actions) to yield m'.

We want to identify certain processes of MSCs when composing them. Let $\Pi_k = \{\pi_1, \ldots, \pi_k\}$ be a set of k *process identifiers*. To simplify the presentation, we fix Π_k, for some k, for the rest of the paper.

Definition 2. *A* named *MSC over P is a tuple (m, β) where m is an MSC over P and $\beta : \Pi_k \to P$ is an injective mapping, which assigns to every process identifier a process.*

Note that P must comprise at least k processes. We usually denote a named MSC by M or M'. Figure 1 illustrates two named MSCs M_1 and M_2. We are now ready to define the sequential composition for named MSCs.

When two named MSCs are concatenated, the processes corresponding to the same process identifier get identified and their events get causally related; the other processes are simply added as separate processes.

Definition 3. *Let $M = ((P, E, \{\leq_p\}_{p\in P}, \lambda, \mu), \beta)$ and $M' = ((P', E', \{\leq'_p\}_{p\in P'}, \lambda', \mu'), \beta')$ be two named MSCs. Since processes and events can be renamed, we may assume that for all $p \in P, p' \in P'$,*

- *$p = \beta(\pi)$, $p' = \beta'(\pi)$ for some $\pi \in \Pi_k$ implies $p = p'$ (hence $\beta = \beta'$) and*
- *$p \notin \beta(\Pi_k), p' \notin \beta'(\Pi_k)$ implies $p \neq p'$.*

Also, assume $E \cap E' = \emptyset$. The concatenation *$M.M'$ of M and M' is the named MSC $((P'', E'', \{\leq''_p\}_{p\in P''}, \lambda'', \mu''), \beta'')$ where* [3]

[3] For two functions h and l over disjoint domains, $h \cup l$ denotes the function over the combined domain defined in the expected manner.

– $P'' = P \cup P'; \quad E'' = E \cup E'$
– $\leq''_p = \begin{cases} \leq_p & \text{for } p \in P \setminus \beta(\Pi_k) \\ \leq'_p & \text{for } p \in P' \setminus \beta'(\Pi_k) \\ \leq_p \cup \leq'_p \cup (E_p \times E'_p) & \text{for } p \in \beta(\Pi_k) \end{cases}$
– $\lambda'' = \lambda \cup \lambda'; \quad \mu'' = \mu \cup \mu'; \quad \beta'' = \beta \ (= \beta')$.

It is easy to see that whenever we substitute named MSCs with equivalent ones, their compositions yield equivalent named MSCs.

The idea of the join composition is to take the union of the processes of two named MSCs and to identify a new set of processes for the process identifiers. Let $\langle \Lambda, \Delta \rangle$ denote that Λ and Δ form a partition of the set of process identifiers Π_k, i.e. $\Lambda \cup \Delta = \Pi_k$ and $\Lambda \cap \Delta = \emptyset$.

Definition 4. *Let $M = ((P, E, \{\leq_p\}_{p \in P}, \lambda, \mu), \beta)$ and $M' = ((P', E', \{\leq'_p\}_{p \in P'}, \lambda', \mu'), \beta')$ be two named MSCs. Because of renaming, we may assume that $P \cap P' = \emptyset$ and $E \cap E' = \emptyset$. Let $\langle \Lambda, \Delta \rangle$ be a partition of Π_k. The* join *of M and M' with respect to $\langle \Lambda, \Delta \rangle$ is denoted by $\mathsf{join}_{\langle \Lambda, \Delta \rangle}(M, M')$ and is the named MSC $((P'', E'', \{\leq''_p\}_{p \in P''}, \lambda'', \mu''), \beta'')$ defined by*

– $P'' = P \cup P''; \quad E'' = E \cup E'$,
– $\{\leq''_p\}_{p \in P''}$ *is defined by* $\leq''_p = \leq_p$ *for* $p \in P$ *and* $\leq''_p = \leq'_p$ *for* $p \in P'$,
– $\lambda'' = \lambda \cup \lambda'; \quad \mu'' = \mu \cup \mu'$,
– $\beta''(\pi)$ *yields* $\beta(\pi)$ *for* $\pi \in \Lambda$ *and* $\beta'(\pi)$ *for* $\pi \in \Delta$.

Note that no event of M is causally related to any event of M' in $\mathsf{join}_{\langle \Lambda, \Delta \rangle}(M, M')$. Also, it is obvious that the operation is robust for equivalent named MSCs.

Concatenation and join of named MSCs is illustrated in Figure 1 below.

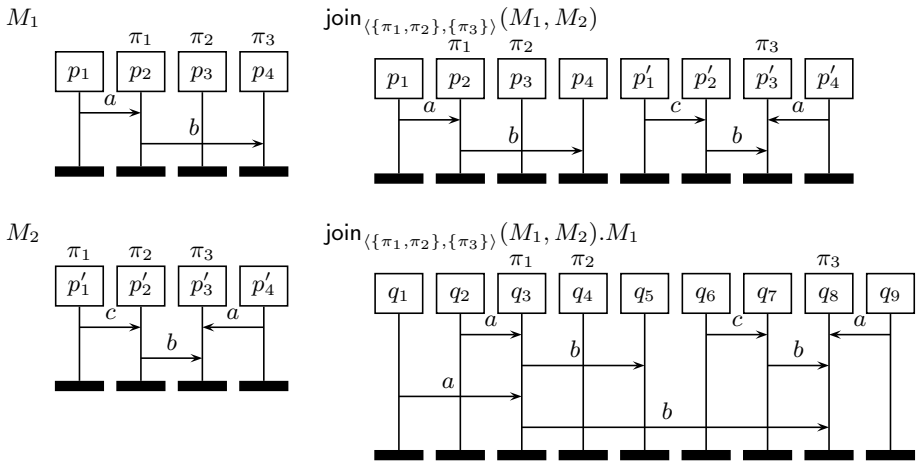


Fig. 1. Concatenation and join of MSCs

Let $\mathcal{M}$ be a finite set of named MSCs (named with process identifiers Π_k). Let Σ be an alphabet and $\overline{\ }: \Sigma \to \mathcal{M}$ a bijection between Σ and $\mathcal{M}$. Let us fix these for the rest of the paper.

A *term* is an expression given by:

$$term ::= b \mid term.term \mid \mathsf{split}_{\langle \Lambda, \Delta \rangle}(term, term)$$

where $b \in \Sigma$ and $\langle \Lambda, \Delta \rangle$ is a partition of Π_k.

To any term t, we can now associate a named MSC $[t]$ inductively as follows: $[b] = \bar{b}$, $[t_1.t_2] = [t_1].[t_2]$ and $[\mathsf{split}_{\langle \Lambda, \Delta \rangle}(t_1, t_2)] = \mathsf{join}_{\langle \Lambda, \Delta \rangle}([t_1], [t_2])$.

For example, $b.\mathsf{split}_{\langle \Lambda, \Delta \rangle}(b_1, b_2).b'$ is a term (where $b, b_1, b_2, b' \in \Sigma$). In this term, we start with the MSC $\bar{b}$. Then the processes Λ and Δ split. In the first branch, the processes in Λ interact with a new set of processes (instantiated with processes identifiers in Δ) and interact with them as specified in $\bar{b_1}$. Similarly, in the other branch, new processes with identifiers in Λ are created and interact with the processes labelled Δ in $\bar{b}$, according to $\bar{b_2}$. Then, the original processes join and interact as specified in $\bar{b'}$.

A set of terms hence represents a set of MSCs obtained using concatenation and join operations of the MSCs in $\mathcal{M}$. We specify sets of terms using a grammar:

Definition 5. *Let $\mathcal{N}$ be a finite set of non-terminals and $S_0 \in \mathcal{N}$ a start symbol. A* fork-and-join MSC grammar *is a tuple $(\mathcal{R}, S_0, \Sigma, \mathcal{M}, \bar{\ })$ where $\mathcal{R}$ is a set of (context-free grammar) rules of the form $S \to nterm$ where $nterm$ is a term with non-terminals, given by the grammar*

$$nterm ::= b \mid S \mid nterm.nterm \mid \mathsf{split}_{\langle \Lambda, \Delta \rangle}(nterm, nterm)$$

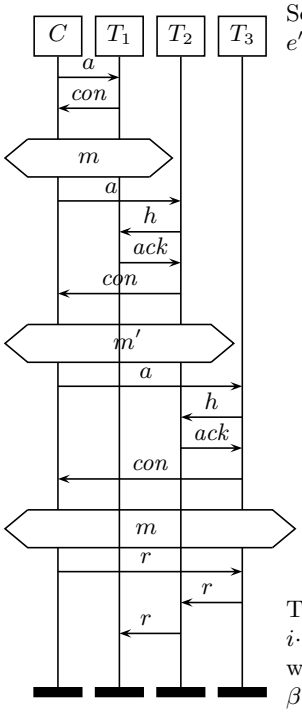
where $b \in \Sigma$, $S \in \mathcal{N}$ and $\langle \Lambda, \Delta \rangle$ is a partition of Π_k.

The set of terms that are derived using a fork-and-join MSC grammar $\mathcal{G}$ is denoted $T(\mathcal{G})$.

Definition 6. *Let $\mathcal{G} = (\mathcal{R}, S_0, \Sigma, \mathcal{M}, \bar{\ })$ be a* fork-and-join MSC grammar*. The* language *of $\mathcal{G}$ is denoted by $\mathcal{L}(\mathcal{G})$ and is defined as*

$$\mathcal{L}(\mathcal{G}) = \{m \mid \text{there is } t \in T(\mathcal{G}) \text{ and an assignment } \beta \text{ with } (m, \beta) \in [t]\}$$

Example 1. Imagine a scenario in which a car C travels from a source to a destination on a route guided by a number of transmitters. The number of transmitters is not fixed. A typical scenario for three transmitters is depicted in Figure 2. Initially the car C sends an "a" (approach) signal to the first transmitter T_1. On receiving a connect signal "con" from T_1, the car and the transmitter interact using a protocol which is described by some MSC m. As the car moves away from a transmitter T_i and approaches the next transmitter T_{i+1}, it sends the approach signal to T_{i+1}. T_{i+1}, on receiving this, requests T_i to hand-over ("h") the control. When T_{i+1} receives an acknowledgement, it sends a connect signal "con" to C, upon which they start their protocol m. Once the car reaches its destination, it informs its current transmitter and this message is relayed back to the first transmitter. In the figure below, we describe the above scenarios in terms of a fork-join grammar.



Set $k = 3$, $\Pi = \{c, t_1, t_2\}$, and $\Sigma = \{i, m, m', n, n', b, b', e, e', \varepsilon\}$. The names in Σ denote the following named MSCs:

- ε denotes the empty MSC (i.e. the one with no events)
- i is the named MSC where c sends "a" to t_1 and t_1, on receiving this, sends back "*con*" to c,
- The MSC m is the protocol that the car establishes with the transmitter t_1,
- n describes this sequence of messages: c sends "a" to t_2, t_2 sends "h" to t_1, t_1 sends *ack* to t_2 and t_2 sends "*con*" to c,
- b describes t_2 sending "r" to t_1,
- e describes c sending "r" to t_1,
- m',n', b', e' are the MSCs obtained from m, n, b and e, respectively, by switching the roles of t_1 and t_2.

$$\begin{aligned}
\mathcal{S} &\to i \cdot m \cdot \mathcal{L}_0 \\
\mathcal{L}_0 &\to n \cdot m' \cdot \mathsf{split}_{\langle\{t_1\},\{c,t_2\}\rangle}(\varepsilon, \mathcal{L}_1) \cdot b \\
\mathcal{L}_0 &\to e \\
\mathcal{L}_1 &\to n' \cdot m \cdot \mathsf{split}_{\langle\{t_2\},\{c,t_1\}\rangle}(\varepsilon, \mathcal{L}_0) \cdot b' \\
\mathcal{L}_1 &\to e'
\end{aligned}$$

This generates, for example, the term $i{\cdot}m{\cdot}n{\cdot}m'{\cdot}\mathsf{split}_{\langle\{t_1\},\{c,t_2\}\rangle}(\varepsilon, n'{\cdot}m{\cdot}\mathsf{split}_{\langle\{t_2\},\{c,t_1\}\rangle}(\varepsilon, e){\cdot}b'){\cdot}b$, which depicts the figure on the left, with $\beta = \{c \mapsto C, t_1 \mapsto T_1, t_2 \mapsto T_2\}$.

Fig. 2. A typical scenario for three transmitters and the grammar generating them

Monadic Second-Order Logic

We assume a supply $eVar = \{x, y, \ldots\}$ of individual (first-order) variables, which are interpreted over events of an MSC, and a supply $EVar = \{X, Y, \ldots\}$ of (second-order) set variables, which are interpreted over sets of events. Since the number of processes is unbounded, we extend our logic to quantification over processes as well. Thus, let $pVar = \{u, v, \ldots\}$ be a supply of (first-order) process variables and $PVar = \{U, \ldots\}$ be a set of (second-order) variables interpreted over sets of processes.

The syntax of monadic second-order logic (MSO) over MSCs is given by

$$\begin{aligned}
\varphi ::= (u, x) \xrightarrow{a} (v, y) \mid x \leq_u y \mid x \in X \mid u \in U \mid \\
\neg\varphi \mid \varphi \vee \varphi \mid \exists x\, \varphi \mid \exists X\, \varphi \mid \exists u\, \varphi \mid \exists U\, \varphi
\end{aligned}$$

where $u, v \in pVar$, $U \in PVar$, $x, y \in eVar$, $X \in EVar$, and $a \in \Gamma_M$. The notions of free and bound variables are introduced as usual.

The intuitive meaning of $(u, x) \xrightarrow{a} (v, y)$ is that x is a send-event of process u and y is the corresponding receive event in process v and the message sent is a. $x \leq_u y$ holds iff y follows x in the total order of the process given by u. Let $m = (P, E, \{\leq_p\}_{p \in P}, \lambda, \mu)$ be an MSC. Let $\mathcal{I}$ be an interpretation function which assigns to every x, X, u and U, an event, a set of events, a process and a set of processes, respectively. The satisfaction relation $m \models_{\mathcal{I}} \varphi$ for a formula $\varphi \in$ MSO is inductively defined as follows:

– $m \models_{\mathcal{I}} (u,x) \xrightarrow{a} (v,y)$ iff $\lambda(\mathcal{I}(x)) = (\mathcal{I}(u)!\mathcal{I}(v), a)$, $\lambda(\mathcal{I}(y)) = (\mathcal{I}(v)?\mathcal{I}(u), a)$, and $\mu(\mathcal{I}(x)) = \mathcal{I}(y)$,
– $m \models_{\mathcal{I}} x \leq_u y$ iff $\mathcal{I}(x) \leq_{\mathcal{I}(u)} \mathcal{I}(y)$.

The remaining constructs are defined as usual. For a formula without free variables, we write $m \models \varphi$ instead of $m \models_{\mathcal{I}} \varphi$. Note that the causal order relation $\leq$, though not explicitly present, can be expressed using our syntax (see [4]). Also, the logic cannot distinguish between equivalent MSCs.

We can now state the main result of the paper:

Theorem 1. *The problem of checking, given a fork-and-join MSC grammar $\mathcal{G}$ and an MSO formula φ, whether all the MSCs corresponding to terms in $T(\mathcal{G})$ satisfy φ, is decidable.*

3 Model checking

Trees and automata: A *tree* is a graph $T = (N, Edg)$ where N is a finite subset of $\{0,1\}^*$ of nodes such that (i) N is prefix-closed and (ii) if $x \in N$, then either both $x.0$ and $x.1$ are in N or neither is in N, and $Edg = \{(x, x.i) \mid x, x.i \in N, i \in \{0,1\}\}$. Thus the trees we consider are finite binary trees where every node has either zero or two children.

For a finite set of labels Γ, a Γ-labelled tree is a pair (T, λ) where $T = (N, Edg)$ is a tree and $\lambda : N \to \Gamma$ is a labelling function that assigns a label to every node of the tree.

A tree-automaton over Γ-labelled trees is a tuple $\mathcal{A} = (Q, init, \delta, Q_f)$ where Q is a finite set of states, $init : \Gamma \to 2^Q$ is a function that associates a set of initial states with every label, $Q_f \subseteq Q$ is a set of final states and $\delta : Q \times Q \times \Gamma \to 2^Q$ is a bottom-up transition function that associates a set of possible states to a node depending on its label and the states associated with its children.

A run of such an automaton over a Γ-labelled tree (T, λ), with $T = (N, Edg)$ is a function $\rho : N \to Q$ such that for every leaf v in T, $\rho(v) \in init(\lambda(v))$, and for every internal node v, $\rho(v) \in \delta(\rho(v.0), \rho(v.1), \lambda(v))$. Such a run is said to be accepting if $\rho(\epsilon) \in Q_f$, i.e. if the root is labelled by a final state. A labelled tree is accepted by $\mathcal{A}$ if there is an accepting run of $\mathcal{A}$ over it. The language of trees accepted by $\mathcal{A}$, $\mathcal{L}(\mathcal{A})$, is the set of trees it accepts. A set of Γ-labelled trees is said to be regular if there is an automaton whose language is this set.

Parse trees and i-trees: We work with trees that represent the parse-trees of terms. Formally, for a given alphabet Σ, the set of *parse trees* over Σ is the set of all Γ-labelled trees, where $\Gamma = Ops \cup \Sigma$ with $Ops = \{.\} \cup \bigcup_{\langle \Lambda, \Delta \rangle} \{\mathsf{split}_{\langle \Lambda, \Delta \rangle}\}$, in which all the leaves are labelled with letters in Σ and all the internal nodes are labelled with letters in Ops.

For a term t over Σ, we can clearly associate a unique parse tree over Σ. Also, every parse tree over Σ corresponds to a term over Σ. It is also easy to see that the set of parse trees over Σ is regular. We assume henceforth that the automata we construct work only over parse trees—this is an acceptable assumption since

we can construct an automaton that accepts the set of all parse trees and take its intersection with the automata we construct.

In addition to having a tree represent a term, we work with trees that represent terms along with an interpretation of a set of variables. Let V be a finite set of first-order and second-order process and event variables. Recall that every $b \in \Sigma$ is associated with an MSC $\bar{b}$. A *partial interpretation* of V over b is a function I that maps some first-order (second-order) process variables in V to a process (a set of processes) in $\bar{b}$, and some first-order (second-order) event variables to an event (a set of events) in $\bar{b}$. In other words, it is an interpretation of some subset of variables $V' \subseteq V$ over $\bar{b}$. An *interpretation tree*, or an i-tree over V is a Γ-labelled tree where $\Gamma = Ops \cup \{(b, I) \mid b \in \Sigma, I \text{ is a partial interpretation of } V \text{ over } b\}$. We call the underlying term the tree represents as the term associated with the i-tree, i.e. the term associated with the parse tree obtained when each leaf labelled (b, I) is instead labelled b.

If an i-tree over V is associated with a term t, then the i-tree is supposed to represent both t and an interpretation for the variables in V to the processes and events in $[t]$, the MSC associated with t. For an interpretation function $\mathcal{I}$ of V over $[t]$, it is clear how to associate an i-tree that represents it: For each leaf in the parse tree of t labelled b, map a process variable p to a process in $\bar{b}$, provided the process is the same process interpreted by $\mathcal{I}$ in $[t]$. Similarly, process set variables, event and event set variables can be interpreted locally at each leaf. We refer to this i-tree as the one that corresponds to $\mathcal{I}$. Note that in this i-tree, a first-order event variable gets interpreted at only one leaf. However, a first-order process variable could get interpreted at many leaves—this is because a process in $[t]$ is formed using events of many MSCs at the leaves of the i-tree.

It is clear from the above that not every i-tree over V which is associated with a term t may correspond to an interpretation $\mathcal{I}$ over V—in particular, the first-order process variables defined at various leaves may not correspond to a single process in $[t]$. Also, we clearly require that a first-order event variable is given an interpretation in only one of the leaves. We say an i-tree is *legal* if there is an interpretation $\mathcal{I}$ over V that it corresponds to. Our main task now is to identify the set of legal i-trees.

Let us add an additional layer of labelling to the nodes of an i-tree as follows. The labelling set will be the set Ξ where each element of Ξ is of the form $(\mathrm{IP}, \mathrm{IE}, \eta, \zeta)$ where:

- IP ("interpreted process variables") and IE ("interpreted event variables") are subsets of first-order process and event variables of V, respectively,
- $\eta \subseteq \mathrm{IP} \times \{U \mid U \text{ is a second-order process variable in } V\}$,
- $\zeta : \mathrm{IP} \rightharpoonup \Pi_k$ is a *partial* function that associates some first-order process variables of IP to process identifiers.

When we label a node v of an i-tree with $(\mathrm{IP}, \mathrm{IE}, \eta, \zeta)$, the intuition is that in the subterm represented by the sub-tree rooted at v, IP and IE are the set of first-order process and event variables that have been given an interpretation in the MSC associated with the sub-term. Also, $(u, U) \in \eta$ iff in the interpretation of u in the sub-term, the process u has been declared to be an element of U.

The meaning of ζ is that $\zeta(u) = \pi_i$ iff in the named MSC corresponding to the subterm, the process identifier π_i is assigned the same process as the one u is interpreted with.

The labelling of leaves of an i-tree is straightforward—we label a leaf labelled (b, I) with $(\mathrm{IP}, \mathrm{IE}, \eta, \zeta)$ where IP and IE are the set of all first-order process variables and first-order event variables interpreted by I, respectively; η is the set of all (u, U) where $u \in \mathrm{IP}$ and the process associated with u belongs to the set of processes associated with U, and $\zeta(u) = \pi_i$ iff u is interpreted as a process with identifier π_i.

We can now label the tree bottom-up, starting at the leaves. If v is a node of the tree and its children $v.i$ are labelled $(\mathrm{IP}_i, \mathrm{IE}_i, \eta_i, \zeta_i)$, where $i \in \{0, 1\}$, then we can label v as follows. There are two cases, depending on whether the node v is labelled '.' or '$\mathsf{split}_{\langle \Lambda, \Delta \rangle}$'.

If v is labelled '.', then we say that the labels of its children are consistent if (i) for every $u \in \mathrm{IP}_0 \cap \mathrm{IP}_1$ and $U \in V$, $(u, U) \in \eta_0$ iff $(u, U) \in \eta_1$ and (ii) for every $u \in \mathrm{IP}_0 \setminus \mathrm{IP}_1$, ζ_0 is undefined on u and for every $u \in \mathrm{IP}_1 \setminus \mathrm{IP}_0$, ζ_1 is undefined on u, (iii) for every $u \in \mathrm{IP}_0 \cap \mathrm{IP}_1$, both ζ_0 and ζ_1 are defined on u and $\zeta_0(u) = \zeta_1(u)$ and (iv) $\mathrm{IE}_0 \cap \mathrm{IE}_1 = \emptyset$.

Intuitively, (i) says that, since the MSCs associated with the children of v are going to be concatenated, if a process variable is defined in both MSCs, then the sets of process set variables they are declared to belong to must be the same. Conditions (ii) and (iii) say that for every interpreted process variable u, either u is interpreted only in one of the named MSCs and is not associated with any process identifier, or u is interpreted in both named MSCs and are associated with the same process identifier. The reason behind these conditions is immediate when one observes that when two named MSCs are concatenated, processes whose event-lines get concatenated are exactly those which are associated with process identifiers. The last condition ensures that a first-order event variable is not interpreted in both the subtrees.

If v's children are consistent, it is clear that we can label v as $(\mathrm{IP}, \mathrm{IE}, \eta, \zeta)$ where $\mathrm{IP} = \mathrm{IP}_0 \cup \mathrm{IP}_1$, $\mathrm{IE} = \mathrm{IE}_0 \cup \mathrm{IE}_1$, $\eta = \eta_0 \cup \eta_1$ and $\zeta(u) = \zeta_0(u)$ if $u \in \mathrm{IP}_0$ and $\zeta_1(u)$, otherwise.

Let us now turn to the case when v is labelled '$\mathsf{split}_{\langle \Delta_0, \Delta_1 \rangle}$'. We say that the labels of its children are consistent if $\mathrm{IP}_0 \cap \mathrm{IP}_1 = \emptyset$ and $\mathrm{IE}_0 \cap \mathrm{IE}_1 = \emptyset$. Hence, all that we require is that the first-order process and event variables interpreted in the component MSCs be disjoint. If the labels of v's children are consistent, then we can label v as $(\mathrm{IP}, \mathrm{IE}, \eta, \zeta)$, where $\mathrm{IP} = \mathrm{IP}_0 \cup \mathrm{IP}_1$, $\mathrm{IE} = \mathrm{IE}_0 \cup \mathrm{IE}_1$, $\eta = \eta_0 \cup \eta_1$ and ζ is given as follows: for every $u \in \mathrm{IP}_i$, if $\zeta_i(u) \in \Delta_i$, $\zeta(u) = \zeta_i(u)$, where $i \in \{0, 1\}$. It is easy to see that this update maintains the semantics of the labelling according to the join operation of the two MSCs.

We say a labelling of the nodes of an i-tree is consistent if it can be labelled using the above rules (i.e. every leaf is labelled as described above and every internal node's children are labelled consistently and respects the above labelling rule). It is now easy to observe the following:

Proposition 1. *An i-tree over V is legal iff it has a consistent labelling in which the label of the root is (IP, IE, η, ζ) with IP∪IE spanning all the first-order process and event variables of V.*

The following is now immediate:

Lemma 1. *The set of all legal i-trees is regular.*

Proof. The labelling set Ξ above can be taken to be the set of the states of a tree-automaton working over i-trees. It is easy to engineer such an automaton that accepts an i-tree iff the i-tree admits a consistent labelling. □

In the sequel, we assume that a formula which uses a variable, has at most one quantification involving it. In the style of the well-known automata theoretic approach to decidability [9], we can now show:

Lemma 2. *For any MSO formula φ with free-variables V, the set of all legal i-trees t over V such that the MSC represented by t satisfies φ under the interpretation of V defined by t, is regular.*

Proof. The proof will be by induction on the structure of the formula φ:

For the atomic formula of the kind $(u, x) \xrightarrow{a} (v, y)$, when reading a legal i-tree, the automaton can check the formula by checking if there is some leaf labelled (b, I), where both x and y are interpreted by I, and u and v are interpreted as the processes x and y belong to, respectively, and x and y are matching send and receive events of a message labelled a in $[b]$. It is easy to construct an automaton which accepts a tree iff it has such a leaf.

For the atomic formula $x \leq_u y$, the automaton is more complex. Recall the labelling used in defining the consistency of an i-tree. It is easy to verify that $x \leq_u y$ iff x and y are both interpreted (at leaves) as events of a process that is interpreted as u (in the leaves) and one of the following hold:

- there is a leaf (say labelled (b, I)) where both x and y are interpreted and the event interpreted for x is causally before the event interpreted for y in $[b]$, or
- Let the bottom-most internal node of the tree which is labelled $(\mathrm{IP}, \mathrm{IE}, \eta, \zeta)$ with $x, y \in \mathrm{IE}$ be v and let its children $v.i$ be labelled $(\mathrm{IP}_i, \mathrm{IE}_i, \eta_i, \zeta_i)$. Then v is labelled '.' and $x \in \mathrm{IE}_0$ and $y \in \mathrm{IE}_1$.

Intuitively, $x \leq_u y$ iff they belong to the process interpreted as u and if they are both interpreted at a leaf and they are causally related in the MSC defined at the leaf, or they get causally related by a concatenation operation. We can design an automaton which finds the consistent labelling and accepts the tree iff the above property is satisfied.

The atomic formula "$x \in X$" can be checked easily by checking them at the leaf where x is interpreted (say labelled (b, I)) and checking if $I(x) \in I(X)$. The formula "$u \in U$" can be handled similarly.

For the formula $\neg\varphi$, we can take the automaton for φ, complement it and take its intersection with the automaton accepting all legal i-trees. For formulas

formed using disjunction, $\varphi \vee \varphi'$, we can take the automata for φ and φ', tinker with them if necessary so that they now work on i-trees over the free variables in $\varphi \vee \varphi'$, and then construct an automaton that accepts the union of these two tree languages.

The formulas formed with existential quantification of the form $\exists W \varphi(W)$ (where W is a first- or second-order, event or process variable) can be handled by taking the automaton for $\varphi(W)$ (call this $\mathcal{A}_\varphi$) and then building an automaton accepting i-trees over V' (where V' is the set of free variables of φ but with W removed from it). This automaton first guesses an appropriate extension of the interpretation at the leaves to include an interpretation of W and then proceeds to simulate $\mathcal{A}_\varphi$ on this extended labelled tree. It hence accepts a legal i-tree over V' iff there is a legal i-tree over $V' \cup \{W\}$ that extends the labelling to include an interpretation of W, and this tree is accepted by $\mathcal{A}_\varphi$. □

For a sentence φ (without free variables), the i-trees are over the empty set and hence are isomorphic to parse-trees. Invoking the above lemma, we have:

Theorem 2. *For any MSO sentence φ, the set of all parse-trees over Σ that correspond to terms that satisfy φ is regular. Moreover, one can effectively construct an automaton $\mathcal{A}_\varphi$ that accepts these trees.*

In the model-checking problem, we are given a fork-and-join MSC grammar $\mathcal{G} = (\mathcal{R}, S_0, \Sigma, \mathcal{M}, \overline{\ })$ and an MSO formula φ. The following is easy to observe:

Lemma 3. *The set of all parse trees corresponding to terms in $T(\mathcal{G})$ is regular.*

Proof. One can first rewrite the rules in $\mathcal{G}$ (but preserving the language of terms) such that each rule is of the form $S \to b$, $S \to T$, $S \to nterm.nterm$ or $S \to \mathsf{split}_{\langle \Lambda, \Delta \rangle}(nterm, nterm)$. (This may require more nonterminals.) Then one can show that a parse tree of a term belongs to $T(\mathcal{G})$ iff there is a labelling of the internal nodes with nonterminals such that the root is labelled with S_0 and if a node is labelled S, then the label of its children along with the parse tree label of the node is according to some rule in the grammar. It is easy to build such an automaton which checks whether there is such a labelling. □

To solve the model-checking problem, we first construct, using Lemma 2, an automaton $\mathcal{A}_{\neg\varphi}$ that accepts precisely the parse-trees over Σ which correspond to MSCs that satisfy $\neg\varphi$. We also construct an automaton $\mathcal{A}_\mathcal{G}$ that accepts the parse trees of terms in $T(\mathcal{G})$ (using Lemma 3). The problem then boils down to checking whether $\mathcal{L}(\mathcal{A}_{\neg\varphi}) \cap \mathcal{L}(\mathcal{A}_\mathcal{G})$ is nonempty, which is decidable. This establishes Theorem 1.

4 Discussion

We have presented a formalism to specify languages of MSCs over unboundedly many processes, and shown that MSO-model checking is decidable for this class. Note that there have been similar efforts to extend model-checking for systems

with a fixed number of processes to unboundedly many processes in the area of verification of *parameterized systems*. In that setting, however, it turns out that the model-checking problem gets quickly undecidable and while there are many efforts to find decidable fragments, there is no formalism such that all problems expressed in the formalism admit an effective model-checking procedure. In this light, the fact that there is a decidable formalism for specifying scenarios for unboundedly many processes is interesting.

It is easy to see that MSGs can be modelled in our framework even without the split operator. However, even discarding the split, our framework can be seen to be more powerful than MSGs, as it allows a context-free grammar to describe the concatenations of the atomic MSCs.

A number of extensions are worthy of study. First, there is an extension of MSGs, called compositional MSGs, where in the specification of an MSC, a send-event can be defined without a matching receive—the matching receive can be defined after an arbitrary delay. Model-checking for MSGs equipped with this feature (CMSGs) specifications is known to be decidable [6]. It turns out that we can extend our results to the class where the atomic named MSCs $\mathcal{M}$ are allowed to have such unmatched send-events. Also, MSO logic can be enriched with modulo counting quantifiers, preserving decidability. See [4] for details.

A future direction is to define structural temporal logics for unboundedly many processes and adapting our procedures to that fragment in order to yield interesting yet efficient algorithms for model-checking. Extensions of the presented formalism to handle infinite MSCs, aimed at analyzing liveness properties, would also be interesting.

References

1. R. Alur and M. Yannakakis. Model checking of message sequence charts. In *Proc. 10th Int'l. Conf. on Concurrency Theory*, LNCS 1664, p. 114–129. Springer, 1999.
2. D. Caucal. On infinite transition graphs having a decidable monadic theory. In *Proc. the 23th International Colloquium on Automata, Languages and Programming (ICALP'96)*, LNCS 1099, p. 194–205, Springer, 1996.
3. B. Courcelle. On the expression of graph properties in some fragments of monadic second-order logic. In *Descriptive complexity and finite models*, volume 31.DIMACS Series in Discrete Mathematics and Theoretical Computer Sciences, June 1997.
4. M. Leucker, P. Madhusudan, and S. Mukhopadhyay. Dynamic message sequence charts. Technical Report MS-CIS-02-27, University of Pennsylvania, 2002.
5. P. Madhusudan. Reasoning about sequential and branching behaviours of message sequence graphs. In *Proc. 27th International Colloquium on Automata, Languages and Programming (ICALP'01)*, LNCS 2076, p. 396–407. Springer, 2001.
6. P. Madhusudan and B. Meenakshi. Beyond message sequence graphs. In *Proc. 21st Conference on Foundations of Software Technology and Theoretical Computer Science*, LNCS 2245, p. 256–267. Springer, 2001.
7. R. Milner. *A Calculus for Communicating Processes*, LNCS 92, Springer, 1980.
8. ITU-TS. ITU-TS Recommendation Z.120: Message Sequence Chart 1996 (MSC96). Technical report, ITU-TS, Geneva, 1996.
9. W. Thomas. Languages, automata and logic. In *Handbook of Formal Languages*, volume 3, Beyond Words. Springer, 1997.