A Process-based Semantics for Message Sequence Charts with Data

Chien-An Chen Sara Kalvala Jane Sinclair
Department of Computer Science
University of Warwick
Coventry CV4 7AL, UK
{*cssdc, sk, jane*}*@dcs.warwick.ac.uk*

Abstract

Message Sequence Charts (MSCs) are a graphical language for scenarios of communicating components exchanging messages in a distributed environment. The language has been standardised by the International Telecommunication Union (ITU) and given a formal semantics by means of a process algebra. Nevertheless, little attention has been given to MSCs where a message with data is a building block. In this paper, we investigate the impact that the concept of data flow brings to the conventional semantics, and propose a CCS-like process calculus as an alternative formal framework. The proposed semantics captures the data flow properties while maintaining the expressiveness of the conventional semantics. Equivalence of MSCs is also discussed from the perspective of the corresponding process equivalence.

1. Introduction

Message Sequence Charts (MSCs) describe scenarios where messages are exchanged between communicating entities in a distributed environment. The syntax and static requirements of the language have been standardised by ITU in the Z.120 recommendation [13]. Attempts at a formal semantics of MSCs emerged after the language was standardised for the first time (MSC'92) [2]. Different approaches were adopted for the task, such as automata theory [16], Petri Nets [8] and process algebra [19, 3]. Notably, Mauw et al. [17] proposed a formal framework for the Z.120 standard, which we refer to hereafter as the *standard MSC semantics*. Their method transforms the textual representation of MSCs into a process algebra based on *ACP* (Algebra of Communicating Processes) [1]. Further work on formalising MSCs still proceeds with respect to the evolution of the language, e.g. MSC'96 [12]. The *high-level* MSC, a structural means to composite MSCs, has been formalised by Mauw et el [18]. Similarly, Gehrke et al. [7] extend the standard semantics with the concern of *conditions*, which are a rudimentary form of MSC compositions. In [15], Kosiuczenko presents a semantics to capture *inline expressions*.

Although the semantics of MSCs has been studied using different approaches, relatively little attention has been drawn to MSCs where the messages have data as parameters. MSCs with data have been syntactically legitimate since MSC'92 and have been mentioned in tutorials such as [25, 26]. Nevertheless, none of these documents clearly specify the meaning of a parameter attached to a message. This paper mainly contributes to answering the question: *what does an MSC with data mean?*

The approach we use is inspired by the work of Mauw et al.[17]. We firstly propose a process calculus as a denotational domain, and then define a translation scheme that transforms a textual MSC into a process term. The meaning of an MSC diagram is actually the behaviour of the corresponding process. The process calculus we define is a variant of the asynchronous version of *CCS* (Calculus of Communicating Systems) [20] and π-*calculus* [10, 11]. We use asynchrony to capture temporal ordering of events in an MSC, and value-passing to model data-binding changes while the MSC proceeds. The CCS syntax and semantics are modified accordingly to accommodate the MSC language. This approach gives a new semantics to MSCs with the following advantages:

1. Our proposed semantics maintains the expressiveness of the standard approach while incorporating the feature of data flow via a uniform process-calculus language. This approach has two consequences. First, not only can our framework model the temporal ordering of message sending and receiving, but also the data passed via the message. Second, MSCs without data can also be captured as MSCs with null-valued data.
2. Our proposed semantics is simpler than the standard one in the sense that the pending output event for asynchrony and the handshake mechanism for value-passing are both well-known in the CCS community.

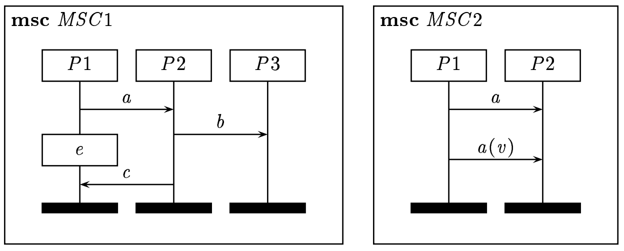

Figure 1. Example of bMSC diagrams.

This paper is organised as follows. Section 2 gives an overview of the MSC language that is relevant to this paper and points out the issues that arise when MSCs are extended with data. In section 3, we propose a process calculus that is capable of modelling value-passing in an asynchronous communication environment. Section 4 is concerned with a translation scheme from an MSC into a process term; readers are also given a simple example illustrating the whole idea and how the mechanism works. In section 5, we define an equivalence relation over MSCs in terms of the process domain. Section 6 shows the direction of future work and presents some conclusions.

2. Message Sequence Charts

This section gives an overview of the MSC language used in this paper. Firstly, we give an intuitive description of an MSC diagram and then introduce MSCs with data; meanwhile, two basic assumptions on data are presented. Then we provide the textual representation of the MSC language. In our discussion, we follow the convention that the letters $a, b, \ldots, e$ denote message names; v represents an actual data value, and x, y are data variables.

2.1. bMSC notation

A *basic* MSC (bMSC) is a building block for an MSC document. As can be seen in Figure 1, *MSC*1 contains three *instances* (or *processes*), namely $P1$, $P2$ and $P3$, represented by vertical axes. Three *messages*, a, b and c, represented by arrows, are exchanged between those instances. The block labelled e along $P1$ is a *local action*, and the frame around the diagram represents the *environment*. *MSC*1 intuitively describes a scenario where $P1$ sends a message a to $P2$ and then executes an action e; after receiving a, $P2$ sequentially sends messages b to $P3$ and c to $P1$.

A temporal ordering is determined along each axis and arrow in the sense that (1) the events along an instance axis occur from top to bottom, and (2) a message must be sent before it is received. Hence, in *MSC*1, the order of receiving b and c is undefined. In this paper, we assume that the communication medium between instances is reliable, i.e. no messages are lost. Furthermore, the setting of a bMSC is generally interpreted to be of *asynchronous* communication.

2.2. bMSCs with data

Messages with data are a legitimate syntax since MSC'92, and have been used extensively in object-oriented methods to represents a remote procedure call and dynamic instance creations, where the data denote the actual values passed through communicating instances. Nevertheless, this incorporation can cause difficulties in interpreting what a *message* means in a bMSC. The problem can be roughly illustrated using *MSC*2 in Figure 1, where $P1$ first send a message a to $P2$ and then sends another message $a(v)$ to $P2$. Without the second message $a(v)$, we can easily interpret the identifier a as a stream of data flowing from $P1$ to $P2$. Similarly, without the first message a, the name a occurring free in $a(v)$ can be regarded as a channel name while the actual stream of data from $P1$ to $P2$ is denoted by v. Yet putting the two messages together as shown in *MSC*2 gives a single identifier a two different meanings. Problems still remain even if we re-label the first message to any name not free in $a(v)$. In a bMSC without any data, between each pair of communicating instances, it is usually interpreted that there exist implicitly at most two (if bidirectional) communication media, where all the sent messages arrive, waiting to be picked up according to a certain queuing mechanism. In this case, $P1$ throws b and v sequentially into the medium and leaves a meaningless.

In the standard MSC semantics [17], a message, say a from Pi to Pj, is treated as two distinct atomic actions, e.g. $out(i, j, a)$ and $in(i, j, a)$. The semantics of a bMSC is the temporal ordering of the actions with respect to the partial order derived from the asynchrony in the bMSC. If we directly extend this framework to bMSCs with data, we could as a first attempt denote a message $a(v)$ from Pi to Pj by $out(i, j, a(v))$ and $in(i, j, a(v))$, interpreted as the sending and receiving of a value v along channel a respectively. We can see that the input event $in(i, j, a(v))$ restricts the coming value to be v. Nevertheless, the value v is controlled by Pi and should not be observable to Pj until it is passed through a. In other words, an input event cannot use the value to be input to prevent the actions being performed. The current method violates the *normal operational semantics of value-passing* [24], which means that an input event should be able to accept *any* value decided by the output event.

In this paper, we adopt a uniform view of message exchanges in bMSCs: each message has a canonical form

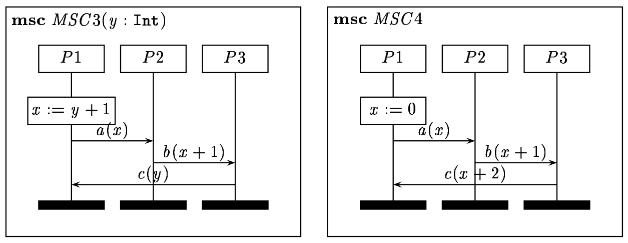

Figure 2. Example bMSCs with data binding.

$a(e)$, where a ranges over channel names and e over data expressions. A message standing alone is also a message $a(e)$ where e is a *null* value and can be simplified as a.

2.3. Assumptions on data

MSCs with data had not been discussed in any depth until the emergence of MSC2000 [13], which officially incorporates a simple data language. Literature such as [5, 6] proposed some problems and design decisions for this incorporation. A summary can be found in the work of Engels [4]. In the following, two requirements on our bMSC language are given and justified accordingly in order to indicate what a *meaningful* bMSC is in our language.

1. *Undefined variables are disallowed.* Undefined variables are not bound to any value and are typically regarded as universally quantified. They cause explosion (or infinity) in possible traces, which we prefer to avoid at this early stage of research. Besides, this requirement coincides with the MSC2000 standard [13]. Two data binding mechanisms, usually appearing in related work such as [4, 9, 14], are illustrated by $MSC3(y)$ in Figure 2. One is *assignment* in a local action, and the other is the *parameterised* bMSC, where actual instantiation for the parameters is presumed.

2. *All variables are local and not sharable.* This means that an instance, say Pi, can only refer to a variable x if and only if either x is local to Pi, or x is passed from some Pj via a message. This requirement fits the MSC spirit and has been adopted in [5]. For example, a naive way to interpret $MSC4$ in Figure 2 is that the messages $a(0)$, $b(1)$ and $c(2)$ are sent sequentially. However, a closer observation suggests that $MSC4$ violates this requirement since what $P3$ receives is the actual value of $x+1$ not x. $P3$ has no knowledge of the value of x, so it cannot send out $c(x+2)$. Thus, only $P1$ and $P2$ can refer to the variable x. Note that the parameterised bMSC is just a syntactical convenience in the sense that a parameter cannot be regarded as a global variable.

`<msc>`	::=	`msc <mscid> (<var>);`
		`<msc body> endmsc`
`<msc body>`	::=	`<> |`
		`<inst def> <msc body>`
`<inst def>`	::=	`instance <instid>;`
		`<inst body> endinstance;`
`<inst body>`	::=	`<> |`
		`<event> <inst body>`
`<event>`	::=	`in <msg> from <address>;|`
		`out <msg> to <address>;|`
		`action <var> := <exp>;`
`<msg>`	::=	`<mid> |`
		`<mid>(<exp>)`
`<address>`	::=	`<instid> |`
		`env`
`<exp>`	::=	`<val> |`
		`<var> |`
		f(`<exp>`$_1, \ldots,$`<exp>`$_n$)

Table 1. Textual syntax of basic MSCs.

2.4. Textual format

In addition to the graphical layout, a pure textual representation is also included in the MSC recommendation Z.120. The reason we introduce the textual bMSC here is to give background for a translation function mapping a textual bMSC to a process term in our process calculus. For the purpose of explanation, however, graphical syntax is used throughout this paper.

The grammar of the textual representation of bMSCs used in this paper is provided in Table 1. The symbol <> denotes an empty string. The terminals, `msc`, `endmsc`, etc, are reserved keywords. The non-terminals `<mscid>`, `<instid>` and `<mid>` represent identifiers for a bMSC, an instance and a message respectively. Since bMSCs with data are used liberally in the literature, the non-terminal `<exp>` requires additional attention. `<val>` accounts for a data value and `<var>` for a data variable. Moreover, f is a n-ary function, applying a list of `<exp>` with n entries with respect to type consistency. Here we use $MSC3(y)$ in Figure 2 as an example to illustrate the textual syntax.

```
msc MSC3(y);
instance P1;             instance P3;
action x := y + 1;       in b(x + 1) from P2;
out a(x) to P2;          out c(y) to P1;
in c(y) from P3;         endinstance;
endinstance;             endmsc

instance P2;
in a(x) from P1;
out b(x + 1) to P3;
endinstance;
```

It is worth noting that, for the present, the MSC language we use is basically limited to communication events and data binding mechanisms, i.e. assignments in local actions and instantiation of parameterised bMSCs. Other constructs like conditions, inline expressions, composition of bMSCs, etc are intentionally left out for simplicity. Likewise, we restrict the message parameter to be monadic, but our framework can be easily extended for polyadic data values.

3. Process calculus theory

In this section, we firstly present the main concerns in capturing the meaning of a bMSC with data; then a process calculus PC_{bMSC} is proposed as a formal framework to implement these ideas.

3.1. Basic concepts

While constructing the formal framework, there are two concepts we intend to realise in the proposed semantics. One is the value-passing behaviour between instances, and the other is the asynchronous communication pattern implied by a bMSC.

Value-Passing. As pointed out in section 2.2, the standard MSC semantics [17] is unsuitable for capturing value-passing properties since the input event is specified as waiting for a specific value which should not be observable by the receiving instance until the value is actually passed. To remedy this flaw, we apply here the key idea of value-passing CCS [20], where the value-passing has been modelled via a reduction of *pure synchronisation* (or *handshake*) of two parallel actions with complement names, e.g. a and $\overline{a}$. The data values are fused with the port names that synchronising processes use. This corresponds to the reduction relation of CCS, i.e.

$$(\overline{a}v.P \mid a(x).Q) \rightarrow (P \mid Q\{v/x\})$$

where a value v is passed through channel a to Q for future use. Notationally, $\overline{a}v$ means sending a value v along a, $a(x)$ receiving any value from a. $Q\{v/x\}$ substitutes all free occurrences of x in Q with v.

Asynchrony. Asynchrony in a bMSC means that the non-blocking output events do not oblige senders and receivers to synchronise when exchanging messages but allow the sender to continue with its tasks while the message reaches its destination. Typically, a buffering mechanism concerned with a certain queuing policy is used to model asynchrony in a distributed environment. Here, however, the communication medium is not described as an observable entity; instead it is realised in the syntax of processes. The approach we use for modelling asynchrony is inspired by the asynchronous CCS and π-calculus [10, 11], where the asynchrony is guaranteed by its syntax, and a process is observed by *asynchronous experiments*. Our work is different from the above in two points. Firstly, we use the syntax of asynchrony to derive the temporal ordering implied by a bMSC. Secondly, the asynchronous observation cannot be applied here, and we need a different approach for the equivalence between processes.

We next flesh out the above two concepts and form a variant of a fragment of asynchronous CCS with value passing. Knowledge of CCS is presumed.

3.2. Syntax

Let $\mathcal{N}$ denote a potentially infinite set, ranged over by alphabets $a, b, \ldots, g$, which function as the names of all communication ports. A tuple $(a, i, j) \in \mathcal{N} \times \mathbb{N} \times \mathbb{N}$ denotes a communication channel a from the instance Pi to Pj for some $i, j \in \mathbb{N}$. For simplicity, we put it in a subscripted form as a_{ij}. The set $\overline{\mathcal{N}}$ is a set of complement names of the corresponding names in $\mathcal{N}$, i.e. $\overline{\mathcal{N}} = \{\overline{a} \mid a \in \mathcal{N}\}$. Similarly, a name in $\overline{\mathcal{N}}$ can also be subscripted as $\overline{a_{ij}}$ to denote a tuple $(\overline{a}, i, j) \in \overline{\mathcal{N}} \times \mathbb{N} \times \mathbb{N}$. We use $\mathcal{E}$ for a set of data expressions ranged over by $e, e_1, \ldots$. Also we let $\mathcal{V}$ be a set of data values ranged over by $v, w, v_1, \ldots$ and $\mathcal{X}$ be a set of data variables ranged over by x, y, z. The names $v, w, \ldots$ are abstractions of certain data values, so their occurrences are bound. In different applications, the set which $\mathcal{V}$ represents can vary, e.g. integers, strings, etc. The set $\mathcal{P}$ of process expressions, where $P, Q, R \in \mathcal{P}$, is then defined by the following syntax,

$$\begin{aligned} P &::= \mathbf{0} \mid \overline{a_{ij}}(e).\mathbf{0} \mid (P) \mid P_1|P_2 \mid \\ &\quad\;\; \pi.P \mid P\backslash L \\ \pi &::= a_{ij}(x) \mid !a_{ij} \mid (x \leftarrow e)_i \\ e &::= null \mid x \mid v \mid f(e_1, \ldots, e_n) \end{aligned}$$

where the symbol $\mathbf{0}$ denotes an inactive process. The *guarded* process $\pi.P$ and the *parallel composition* $P \mid Q$ have the usual meaning as those in CCS. In the *restriction* $P\backslash L$, the set $L \subseteq \mathcal{N} \times \mathbb{N} \times \mathbb{N}$ consists of subscripted port names. Its meaning is tackled operationally in section 3.4. The expression e has identical structure as `<exp>` defined in the previous section, which means that the data being passed can be a value, a variable or a n-ary function applied to its arguments.

In the above syntax, we use decorated names to represent atomic actions required in observing bMSCs. There are four kinds of such names:

- $\overline{a_{ij}}(e)$ is a data expression e pending along a between Pi and Pj and ready to be picked up.
- $a_{ij}(x)$ means that the instance Pj is waiting for *something* from Pi along a port named a.

- $!a_{ij}$ denotes the action of *Pi* sending a message to *Pj* along *a*.
- $(x \leftarrow e)_i$ is the binding of a variable to an expression, occurring on the instance *Pi*.

A set $\mathcal{A}$ consisting of the four kinds of actions can be defined accordingly as $\mathcal{A} = \mathcal{L}(\pi) \cup \{\overline{a_{ij}}(e)\}$ where $\mathcal{L}(\pi)$ denotes the set spanned by π. A non-$\mathbf{0}$ process can only be guarded by a non-overlined action. We use this mechanism to incorporate asynchrony. This mechanism also indicates the difference between $\overline{a_{ij}}(e)$ and $!a_{ij}$, where the former represents a pending message usually guarded by the latter, denoting an output event.

Three binding operators bind decorated names or variables in process expressions. First, the input prefix of $a_{ij}(x).P$, occurring bound, binds all free occurrences of x in P. Second, for all $a_{ij} \in L$, the restriction operator $P\backslash L$ also binds the name a_{ij} and its complement $\overline{a_{ij}}$ in P. Third, all free occurrences of x in P are bound by the binding prefix of $(x \leftarrow e)_i.P$. A name which is not *bound* is called *free*. We then define $bn(P)$ as a set of the bound names in P and $fn(P)$ as the free names.

Notably, the above syntax does not include *recursion* since we focus on bMSC where only finite traces are defined. The framework can be extended accordingly when taking MSC compositions into considerations.

3.3. Structural congruence

A structural congruence [21] can lead to a very compact set of transition rules without losing any expressiveness. The following definition is a fragment of the structural congruence in π-calculus [22].

Definition 1 *The structural congruence $\equiv$ is the smallest equivalence relation over process expressions $\mathcal{P}$ satisfying the following rules. For $P, Q, R \in \mathcal{P}$, $x \in \mathcal{N} \times \mathbb{N} \times \mathbb{N}$ and $L \subseteq \mathcal{N} \times \mathbb{N} \times \mathbb{N}$,*

1. *$P \equiv Q$ if P is α-convertible to Q*
2. *The Abelian monoid laws for parallel composition, i.e. commutativity $P|Q \equiv Q|P$, associativity $(P|Q)|R \equiv P|(Q|R)$ and $\mathbf{0}$ as a unit $P|\mathbf{0} \equiv P$.*
3. *$\mathbf{0}\backslash L \equiv \mathbf{0}$, $(P\backslash\{x\})\backslash L \equiv P\backslash\{x\} \cup L$*
4. *If $x \notin fn(P)$ then $(P \mid Q)\backslash\{x\} \equiv P \mid Q\backslash\{x\}$*

The first rule states that an α-conversion does not affect the behaviour due to the difference in choice of bound names. From Rule 2, we know that the processes in parallel composition are unordered and can be safely presented as $P \mid Q \mid R$ without parentheses. Rule 3 states that a subscripted name in the restriction set can be pulled out to be associated with the process, and Rule 4 means that the scope of restriction can move into a parallel composition.

3.4. Operational semantics

The operational semantics for our process calculus is defined in terms of a *labelled transition system* (LTS), where transition relations are of the form $P \xrightarrow{\alpha} Q$ for a set of actions ranged by α. Here we let $Act = \mathcal{A} \cup \{?a_{ij}e\}$ denote the action-set ranged by α. The set $\mathcal{A}$ has been mentioned in the previous section. Here, we introduce a new action $?a_{ij}e$, a dual of $!a_{ij}$, that represents the interaction between two complement names $a_{ij}(x)$ and $\overline{a_{ij}}(e)$ (denoted by a silent action τ in CCS). Its meaning and justification will be discussed after the presentation of the transition rules.

We then define a function $sn : Act \rightarrow \mathcal{N} \times \mathbb{N} \times \mathbb{N}$, which takes an action and returns its subscripted port name, i.e. $sn(!a_{ij}) = sn(a_{ij}(x)) = sn(\overline{a_{ij}}(e)) = sn(?a_{ij}e) = a_{ij}$ for some $i, j \in \mathbb{N}$. The function is *partial* in the sense that $sn((x \leftarrow e)_i)$ is undefined. Finally, an evaluation function $[\![\_]\!] : \mathcal{E} \rightarrow \mathcal{V}$ evaluating a data expression and returning a data value is given: for all $e \in \mathcal{E}$

$$[\![e]\!] = \begin{cases} v & \text{if } e = v \\ I(x) & \text{if } e = x \\ f([\![e_1]\!], \ldots, [\![e_n]\!]) & \text{otherwise} \end{cases}$$

where the function $I : \mathcal{X} \rightarrow \mathcal{V}$ represents the actual instantiation for the free variable in a parameterised bMSC, e.g. the variable y in $MSC3(y)$.

Definition 2 *The LTS of the processes is a tuple $(\mathcal{P}, \longrightarrow)$ over Act where $\mathcal{P}$ is a set of processes (or states) and $\longrightarrow \subseteq \mathcal{P} \times Act \times \mathcal{P}$ denotes a transition relation. We write $P \xrightarrow{\alpha} Q$ if and only if $(P, \alpha, Q) \in \longrightarrow$. The transition relation $\longrightarrow$ is the smallest set that can be inferred from the rules in Table 2.*

It can be easily observed from the transition rules that the famous τ action, denoting an invisible action in CCS, dose not exist in our framework. Lack of τ action indicates an important paradigm shift between CCS (or π-calculus) and the semantics of MSCs. In CCS, the system is like a black box. The observer observes its behaviour by interacting with its visible actions. What happens within the box is of no concern hence denoted by the symbol τ. In MSC semantics, however, the observer is given enough power to see through the box. The observer observes the system's behaviour by seeing how the components communicate with each other. In brief, it is what happens inside the box that counts for the MSC semantics. These two paradigms can be illustrated by Figure 3, where the dashed lines are invisible to the observer. This concern also justifies the *COM* rule, where the fusion of two complement actions $a_{ij}(x)$ and $\overline{a_{ij}}(e)$ gives us another *visible* action $?a_{ij}[\![e]\!]$ instead of τ. It means *Pi* received a data valued $[\![e]\!]$ from *Pj* along channel *a*.

The prefix rule $PRE1$ and $PRE2$ shows two base cases on the condition of the prefix. If the prefix is an assignment, then a substitution of free x in P with the evaluation

$$PRE1\ (x \leftarrow e)_i.P \xrightarrow{(x\leftarrow [\![e]\!])_i} P\{[\![e]\!]/x\}$$

$$PRE2\ \alpha.P \xrightarrow{\alpha} P\ (\alpha \neq (x \leftarrow e)_i) \qquad PAR\ \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}$$

$$COM\ \frac{P \xrightarrow{a_{ij}(x)} P', Q \xrightarrow{\overline{a_{ij}}(e)} Q'}{P \mid Q \xrightarrow{?a_{ij}[\![e]\!]} P'\{[\![e]\!]/x\} \mid Q'} \qquad RES1\ \frac{P \xrightarrow{\alpha} P'}{P\backslash L \xrightarrow{\alpha} P'\backslash L}\ (sn(\alpha) \notin L)$$

$$RES2\ \frac{P \xrightarrow{\alpha} P'}{P\backslash L \xrightarrow{\alpha} P'\backslash L}\ (\alpha = !a_{ij} \vee \alpha = ?a_{ij}e \textit{ for some } a, i, j)$$

$$STRUCT\ \frac{P' \equiv P, P \xrightarrow{\alpha} Q, Q \equiv Q'}{P' \xrightarrow{\alpha} Q'}$$

Table 2. The operational semantics.

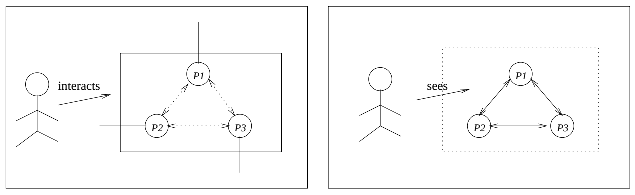

Figure 3. The CCS paradigm and the MSC paradigm.

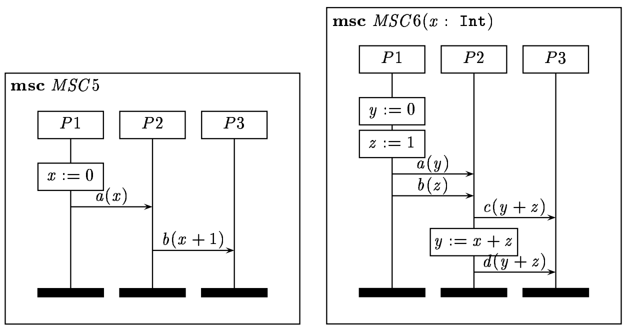

Figure 4. The scope of a variable.

of e is performed after the transition. The *PAR* rule has the usual meaning as that in CCS. The rules RES1 and RES2 say that restriction only applies to the complement actions $a_{ij}(x)$ and $\overline{a_{ij}}(e)$ but *not* to the actions $!a_{ij}$, $?a_{ij}e$ or $(x \leftarrow e)_i$. The *STRUCT* rule corresponds to the structural congruence defined earlier.

4. From bMSCs to PC_{bMSC}

In this section, we first give a brief discussion on the scope of a variable occurring in a bMSC diagram. Then, a mapping function is defined from a textual bMSC to a process expression. Finally, a simple example is given.

4.1. The scope of a variable

One of the assumptions on data in a bMSC is that all the variables are local and not sharable. An instance can only refer to a variable iff either it is local or is passed from another instance via a message. This causes a certain degree of complexity in determining the scope of a variable. A standard case for the above statements can be illustrated by *MSC*5 in Figure 4, where the scope of x covers $a(x)$ and $b(x+1)$. In this case, we can deem $P1$ as the *owner* of the variable x. We use this example to informally introduce two concepts that will be used later.

1. We say that a variable *is bound locally* if the instance which refers to that variable is its owner, e.g. x in $a(x)$ is bound locally.
2. We say that a variable *is bound by its neighbor Pj* for some $j \in \mathbb{N}$ if the instance which refers to that variable is not the owner but has received the variable from Pj. In this case, x in $b(x+1)$ is bound by its neighbor $P1$.

4.2. Mapping bMSCs to processes

The general idea is to represent a bMSC as a single process, obtained by composing in parallel the process representations of component instances. The process is restricted by the subscripted channel names appearing in the bMSC. Within an instance, an output event is split into two atomic actions; one denotes the actual action of message sending, and the other represents a pending message, in any kind of medium, ready to be picked up by the destination instance. An input event is expressed by an action with a variable representing the template used in the subsequent process or computation.

We let the set $\mathcal{D}(\mathtt{X})$ denote the domain of textual content identified by the non-terminal X. For example, an element in $\mathcal{D}(\mathtt{<msc>})$ is a textual bMSC.

Definition 3 *We define the translation function* $[\![\_]\!]_{msc} : \mathcal{D}(\mathtt{<msc>}) \rightarrow \mathcal{P}$, *which maps a textual bMSC to a process term in our* PC_{bMSC} *inductively as*

$$[\![\mathtt{msc\ <mscid>;\ <msc\ body>\ endmsc}]\!]_{msc} = ([\![\mathtt{msc\ body}]\!]_{mscbody})\backslash L$$

where $L \subseteq \mathcal{N} \times \mathbb{N} \times \mathbb{N}$ *consists of subscripted channel names appearing in the considered bMSC. This function means that the process for a bMSC is restricted by all the subscripted channel names appearing in the concerning bMSC. Then the function* $[\![\_]\!]_{mscbody} : \mathcal{D}(\mathtt{<msc\ body>}) \rightarrow \mathcal{P}$, *which puts all the instances in a parallel manner, is defined*

$$[\![\mathtt{<>}]\!]_{mscbody} = \mathbf{0}$$
$$[\![\mathtt{<inst\ def>\ <msc\ body>}]\!]_{mscbody} = [\![\mathtt{<inst\ def>}]\!]_{inst} \mid [\![\mathtt{<msc\ body>}]\!]_{mscbody}$$

The function $[\![\_]\!]_{inst} : \mathcal{D}(\mathtt{<inst\ def>}) \rightarrow \mathcal{P}$ *is defined as*

$$[\![\mathtt{instance}\,Pi\mathtt{;\ <inst\ body>\ endinstance}]\!]_{inst} = [\![\mathtt{<inst\ body>}]\!]^{i}_{body}$$

and with each instance $Pi \in \mathcal{D}(\mathtt{<instid>})$, *a function* $[\![\_]\!]^{i}_{body} : \mathcal{D}(\mathtt{<inst\ body>}) \rightarrow \mathcal{P}$ *is associated. It distinguishes all the events that make up an instance: if the event is a message output, it is split into two sequential atomic actions, i.e. an actual message sending and a pending message. The latter can only occur after the former. Formally*

$$[\![\mathtt{<>}]\!]^{i}_{body} = \mathbf{0}$$
$$[\![E_1;\ E_2;\ \ldots;\ E_n]\!]^{i}_{body} =$$
$$\begin{cases} [\![E_1]\!]^{i}_{out}.([\![E_1]\!]^{i}_{event}.\mathbf{0} \mid [\![E_2;\ \ldots;\ E_n]\!]^{i}_{body}) & \\ \qquad\qquad\qquad \textit{if } E_1 \textit{ is an output event} & \\ [\![E_1]\!]^{i}_{event}.[\![E_2;\ \ldots;\ E_n]\!]^{i}_{body} & \\ \qquad\qquad\qquad \textit{otherwise} & \end{cases}$$

where $E_{1\ldots n} \subseteq \mathcal{D}(\mathtt{<event>})$, *and the function* $[\![\_]\!]^{i}_{out} : \mathcal{D}(\mathtt{<event>}) \rightarrow \mathcal{A}$, *mapping an output event to an action, is defined by*

$$[\![\mathtt{out}\ a(e)\ \mathtt{to}\ Pj]\!]^{i}_{out} = !a_{ij}$$

Finally, the function $[\![\_]\!]^{i}_{event} : \mathcal{D}(\mathtt{<event>}) \rightarrow \mathcal{A}$ *is the base case for three kinds of events. The first one is*

$$[\![\mathtt{in}\ a(e)\ \mathtt{from}\ Pj]\!]^{i}_{event} = \begin{cases} a_{ji} & \textit{if } e = null \\ a_{ji}(x) & \textit{otherwise} \end{cases}$$

where the x occurring free in $a_{ji}(x)$ *has to be a variable that is not bound with respect to its prefix. Formally, it can be formulated as* $x \in \mathcal{X}$ *and* $x \notin bn([\![E_1;\ \ldots;\ E_{k-1}]\!]^{i}_{body})$ *where we let the considered event* $\mathtt{in}\ a(e)\ \mathtt{from}\ Pj$ *in instance Pi be* $E_k \in E_{1\ldots n}$. *The other two cases of this function is*

$$[\![\mathtt{action}\ x := e]\!]^{i}_{event} = (x \leftarrow e\sigma)_i$$
$$[\![\mathtt{out}\ a(e)\ \mathtt{to}\ Pj]\!]^{i}_{event} = \begin{cases} \overline{a_{ij}} & \textit{if } e = null \\ \overline{a_{ij}}(e\sigma) & \textit{otherwise} \end{cases}$$

where $\sigma : \mathcal{E} \rightarrow \mathcal{E}$ *denotes a relabelling function and can be defined inductively as follows*

$$v\sigma = v$$
$$f(e_1, \ldots, e_n)\sigma = f(e_1\sigma, \ldots, e_n\sigma)$$
$$x\sigma = \begin{cases} x & \textit{if x is bound locally or} \\ & \textit{if x is a parameterised variable} \\ x_1 & \textit{if x is bound by its neighbor} \end{cases}$$

where x_1 *has to respect the variable used in the input event that brings x to the instance Pi. Let the considered event* $\mathtt{out}\ a(e)\ \mathtt{to}\ Pj$ *be* $E_k \in E_{1\ldots n}$ *and the input event bringing x to Pi be* $E_s \in E_{1\ldots k-1}$. *The requirement on* x_1 *can be formally expressed as* $[\![E_s]\!]^{i}_{event} = a_{ji}(x_1)$.

This style of mapping function also appears in related work, such as [17, 7], that translates a textual bMSC into a denotational domain. Our work differs from theirs in two senses. First, the function $[\![\_]\!]^{i}_{body}$ implements the idea that an output event has to be split into two atomic actions. The pending message proceeds in parallel with the events following the output event. Second, a complex mechanism is used to build the function $[\![\_]\!]^{i}_{event}$ to ensure the correct data binding as the data language is added into a bMSC. It can be explained via an example as we do next.

4.3. A simple example

We use $MSC6(x)$ in Figure 4 as an example to illustrate the translation and how its corresponding process behaves. This example is particularly designed to show that a liberal use of data variables in a bMSC can lead to complex data binding. The function $[\![\_]\!]^{i}_{event}$ we define earlier ensures the correct binding as the bMSC proceeds. In this case, the occurrences of the variable y in $c(y+z)$ and $d(y+z)$ are bound differently. If the parameterised variable x is instantiated as 1, then the last two messages $P2$ sends to $P3$ will be $c(1)$ and $d(3)$ though that they both send out $y+z$.

As a shorthand, we use identifiers of a bMSC and its instances to represent the whole textual content. For readability, the process for the whole $MSC6(x)$ is expressed via the processes of its components. We also adopt the convention of CCS or π-calculus by writing the process $\alpha.\mathbf{0}$ as α where $\alpha \in Act$ for simplicity. The corresponding process translated from $MSC6(x)$ is in Table 3.

$$
\begin{aligned}
&[\![MSC6(x)]\!]_{msc} = ([\![P1]\!]_{inst} \mid [\![P2]\!]_{inst} \mid [\![P3]\!]_{inst})\backslash\{a_{12}, b_{12}, c_{23}, d_{23}\}\\
&[\![P1]\!]_{inst} = (y \leftarrow 0)_1.(z \leftarrow 1)_1.!a_{12}.(\overline{a_{12}}(y) \mid !b_{12}.(\overline{b_{12}}(z) \mid \mathbf{0}))\\
&[\![P2]\!]_{inst} = a_{12}(x_1).b_{12}(x_2).!c_{23}.(\overline{c_{23}}(x_1 + x_2) \mid (y \leftarrow (x + x_2))_2.!d_{23}.(\overline{d_{23}}(x + x_2) \mid \mathbf{0}))\\
&[\![P3]\!]_{inst} = c_{23}(x_1).d_{23}(x_2)
\end{aligned}
$$

Table 3. The process expression for $MSC6(x)$.

Asynchrony is ensured because the pending message, denoted as an overlined action, does not block what happens after the output event. The data flow is modelled by introducing an intermediate variable, e.g. x_1, in the message-waiting action, $a_{12}(x_1)$. By the transition rule *COM* in Table 2, we note that after the handshake with the pending message, the intermediate variable disappears, and the value is passed. The choice of x_1 and the relabelling mechanism defined in $[\![\_]\!]^i_{event}$ ensure a correct binding.

By applying the operational semantics defined in Table 2, one of the traces derived from the LTS of $[\![MSC6(x)]\!]_{msc}$ with $I(x) = 1$ is,

$$
\begin{aligned}
&\xrightarrow{(y\leftarrow 0)_1}\ \xrightarrow{(z\leftarrow 1)_1}\ \xrightarrow{!a_{12}}\ \xrightarrow{!b_{12}}\ \xrightarrow{?a_{12}0}\ \xrightarrow{?b_{12}1}\ \xrightarrow{!c_{23}}\\
&\xrightarrow{(y\leftarrow 2)_2}\ \xrightarrow{!d_{23}}\ \xrightarrow{?c_{23}1}\ \xrightarrow{?d_{23}3}
\end{aligned}
$$

For brevity, we only show one path in the LTS of the considered process, using the above notation. More paths can be derived. In summary, the semantics of a bMSC here is captured as a set of sequences of input and output events along with data information.

5. Behavioural equivalence

In this section, we discuss the equivalence between bMSCs in terms of their process domain. Briefly, two bMSCs are equivalent if and only if their corresponding process expressions are equivalent. A standard way to give a semantic equivalence to a CCS-like notation is using *bisimulation*. Our work is no exception, but the distinction between *strong* and *weak* bisimulation, based on the ability to observe the invisible action in CCS [20], is no longer applicable here due to the lack of invisible action τ.

A full discussion of bisimulation for our theory will form another paper. For space, we only define strong bisimulation [10, 27] here. In this case, the observer can see every action performed by a process. Moreover, given the current bMSC domain, we argue that the process expressions derived from the bMSCs behave deterministically with respect to their LTSs. This feature enables us to use a simpler model, namely trace semantics, to observe a process and apply the trace equivalence for the equivalence between bMSCs without losing any expressiveness. Also note that a weaker bisimulation is feasible and may be developed in future work.

Definition 4 *A bisimulation is a binary relation $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ over processes satisfying the following two properties. For all $\alpha \in Act$:*

- *if $(P, Q) \in \mathcal{R}$ and $P \xrightarrow{\alpha} P'$, then there exists Q' such that $Q \xrightarrow{\alpha} Q'$ and $(P', Q') \in \mathcal{R}$.*
- *$\mathcal{R}$ is symmetric.*

Two processes P and Q are bisimilar, written $P =_B Q$, if there exists a bisimulation $\mathcal{R}$ such that $(P, Q) \in \mathcal{R}$.

We adopt the convention of van Glabbeek [27] by defining a deterministic process in terms of a *generalised action relation* but in a more convenient style for our proofs:

Definition 5 *Let Act^* denote the set of finite sequences over Act. We write $\varepsilon \in Act^*$ for an empty sequence and $\varphi\rho$ for concatenation of the two sequences $\varphi, \rho \in Act^*$.*

- *The generalised action relation $\Longrightarrow \subseteq \mathcal{P} \times Act^* \times \mathcal{P}$ is defined inductively in terms of $\longrightarrow$. For $P, Q, R \in \mathcal{P}$ and $\alpha \in Act$,*

$$
\begin{aligned}
&P \overset{\varepsilon}{\Longrightarrow} P\\
&P \xrightarrow{\alpha} Q \text{ implies } P \overset{\alpha}{\Longrightarrow} Q \text{ where } \alpha \in Act^*\\
&P \overset{\varphi}{\Longrightarrow} Q \overset{\rho}{\Longrightarrow} R \text{ implies } P \overset{\varphi\rho}{\Longrightarrow} R
\end{aligned}
$$

- *A process $P \in \mathcal{P}$ is deterministic iff*

$$
\begin{aligned}
&\forall P' : \mathcal{P} \text{ such that}\\
&P \overset{\varphi}{\Longrightarrow} P' \wedge P' \xrightarrow{\alpha} R \wedge P' \xrightarrow{\alpha} Q \Rightarrow Q = R
\end{aligned}
$$

Proposition 1 *For all $M \in \mathcal{D}$(`<msc>`), $[\![M]\!]_{msc}$ is deterministic.*

The proof of the above proposition is in the Appendix. We then define the trace observation for a process.

Definition 6 *A trace observation is a function $T : \mathcal{P} \rightarrow \mathbb{P}(Act^*)$ defined as: for all process terms $P \in \mathcal{P}$, $T(P) = \{\varphi : Act^* \mid \exists Q \in \mathcal{P} \text{ such that } P \overset{\varphi}{\Longrightarrow} Q\}$. Accordingly, two processes $P, Q \in \mathcal{P}$ are trace-equivalent, written as $P =_T Q$, if and only if $T(P) = T(Q)$.*

Note that $=_T$ is an equivalence relation by virtue of reflexity, symmetry and transitivity of equivalence on sets.

Lemma 1 *For $P, Q \in \mathcal{P}$ and $\varphi \in Act^*$,*

$$P =_B Q \wedge P \xRightarrow{\varphi} P' \wedge Q \xRightarrow{\varphi} Q' \Rightarrow P' =_B Q'$$

The proof of the above lemma is trivial by mathematical induction on the length of the trace φ. This property is used in the proof of the following proposition:

Proposition 2 *For $M_1, M_2 : \mathcal{D}(\texttt{<msc>})$, $[\![M_1]\!]_{msc} =_B [\![M_2]\!]_{msc}$ if and only if $[\![M_1]\!]_{msc} =_T [\![M_2]\!]_{msc}$*

This proposition states that given the current bMSC domain, bisimulation and trace model coincide. For deterministic processes, this coincidence was found by Park [23], and also documented in [27]. For completeness, we provide our own reasoning in the Appendix.

Definition 7 *For $M_1, M_2 : \mathcal{D}(\texttt{<msc>})$, $M_1 = M_2$ if and only if $[\![M_1]\!]_{msc} =_T [\![M_2]\!]_{msc}$.*

This definition states that two bMSCs are equivalent iff their process expressions are trace-equivalent.

6. Conclusion and future work

In this paper, we present an alternative semantics for a bMSC with data flow. A textual bMSC is firstly transformed into the process domain, and then the operational semantics are used to describe how the process behaves. The process calculus used here is inspired by asynchronous CCS with value-passing. Asynchrony is used to capture the temporal ordering of atomic actions in a bMSC, and value-passing can effectively model the data-binding changes while a bMSC executes.

Jonsson et al. [14] have also developed semantics to MSCs with data. They adopt the *state-based* approach for this task in the sense that the behaviour of an MSC is a sequence of computation steps between *configurations*, which record the current state and data binding of the MSC. A labeled transition system is defined accordingly. The alternative we propose here tackles the problem via a *event-based* view. The behaviour of an MSC is a sequence of atomic actions incorporating the data information. Equivalences between MSCs can also be discussed from the process domain, using different bisimulations.

Here we have only considered a limited set of MSC constructs and defined data variables. Future work can be carried out along these two threads. From the MSC aspect, two important elements can be incorporated into our bMSC language, i.e. *coregions* and inline expressions, which can cause non-determinism in our process domain. In this case, the trace model used in this paper will not be strong enough to distinguish two processes, and therefore new equivalence relations are needed. Furthermore, the composition of bMSCs needs to be introduced to form a complete MSC language. From the data aspect, we plan to demonstrate that the current framework can be extended to capture undefined variables.

References

[1] J. Baeten and C. Verhoef. *Process Algebra*. Cambridge University Press, 1990. Cambridge Tracts in Theoretical Computer Science 18.

[2] CCITT. *CCITT Recommendation Z.120: Message Sequence Chart (MSC)*. Geneva, 1992.

[3] J. de Man. Towards a formal semantics of Message Sequence Charts. In *SDL'93 Using Objects*, Darmstadt, 1993. Elsevier Science Publishers, Amsterdam. Proceeding of the 6th SDL Forum.

[4] A. Engels. Design decisions on data and guards in MSC2000. In S. Graf, C. Jard, and Y. Lahav, editors, *SAM2000, Proceedings of the 2nd Workshop of the SDL Forum Society on SDL and MSC*, pages 33–46, Col de Porte, Grenoble, June 2000.

[5] A. Engels, L. Feijs, and S. Mauw. MSC and data: Dynamic variables. In R. Dssouli, G. von Bochmann, and Y. Lahav, editors, *SDL'99, Proceedings of the 9th SDL Forum*, pages 105–120, Montreal, Canada, June 1999. Elsevier Science Publishers.

[6] L. Feijs and S. Mauw. MSC and data. In Y. Lahav, A. Wolisz, J. Fisher, and E. Holz, editors, *Proceedings of the 1st Workshop of the SDL Forum Society on SDL and MSC*, pages 85–94, Berlin, Germany, June 1998. Humboldt-Universität zu Berlin.

[7] T. Gehrke, M. Huhn, A. Rensink, and H. Wehrheim. An algebraic semantics for Message Sequence Charts documents. In *Proceedings of FORTE/PSTV '98*, Paris, 1998.

[8] J. Grabowski, P. Graubmann, and E. Rudolph. Towards a Petri net based semantics definition for Message Sequence Charts. In *SDL'93 Using Objects*, Darmstadt, 1993. Proceeding of the 6th SDL Forum.

[9] O. Haugen. MSC-2000 interaction diagrams fot the new millennium. *Computer Networks*, 35:721–732, 2001.

[10] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *The Fifth European Conference on Object-Oriented Programming*. Springer-Verlag, July 1991.

[11] K. Honda and M. Tokoro. On asynchronous communication semantics. In *Object-based Concurrent Computing (LNCS 612)*. Springer-Verlag, June 1992.

[12] ITU-TS. *Recommendation Z.120: Message Sequence Chart (MSC)*. Geneva, 1996.

[13] ITU-TS. *Recommendation Z.120: MSC 2000*. Geneva, 2001.

[14] B. Jonsson and G. Padilla. An execution semantics for MSC-2000. In *Proceedings of the 10th SDL Forum*, Copenhagen, Denmark, June 2001.

[15] P. Kosiuczenko. Formalizing MSC'96: Inline expressions. Technical report, Ludwig-Maximilians-Universität München, Institut für Informatik, 1997. Nr. 9705.

[16] P. Ladkin and S. Leue. What do Message Sequence Charts mean? In R. Tenney, P. Amer, and M. Uyar, editors, *Formal Description Techniques VI, IFIP Transactions C*, North-Holland, 1994. Proceeding of the 6th International Conference on Formal Description Techniques.

[17] S. Mauw and M. Reniers. An algebraic semantics of basic Message Sequence Charts. *The Computer Journal*, 37(4):269–277, 1994.

[18] S. Mauw and M. Reniers. Operational semantics for MSC'96. *Computer Networks and ISDN Systems*, 31(17):1785–1799, 1999.

[19] S. Mauw, M. van Wijk, and T. Winter. A formal semantics of synchronous interworkings. In *SDL'93 Using Objects, Proceeding of the 6th SDL Forum*, Darmstadt, 1993. Elsevier Science Publishers, Amsterdam.

[20] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989. ISBN 0-13-114984-9.

[21] R. Milner. Functions as processes. In *Automata, Language and Programming (LNCS 443)*. Springer-Verlag, 1990.

[22] R. Milner. The polyadic π-calculus: a tutorial. Technical report, Laboratory of Foundations of Computer Science, Computer Science Department, University of Edinburgh, 1991.

[23] D. Park. Concurrency and automata on infinite sequences. In *5th GI Conference (LNCS 104)*, pages 167–183. Springer-Verlag, 1981.

[24] S. Reeves and D. Streader. State-based and process-based value-passing. Appearing in ST.EVE Workshop, Pisa, September 2003.

[25] E. Rudolph, P. Graubmann, and J. Grabowski. Tutorial on Message Sequence Charts. *Computer Networks and ISDN Systems - SDL and MSC*, 28(12), 1996.

[26] E. Rudolph, P. Graubmann, and J. Grabowski. Tutorial on Message Sequence Charts (MSC'96). In *Tutorial of the 1st joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification (FORTE/PSTV'96)*, Kaiserslautern, Germany, October 1996.

[27] R. van Glabbeek. The linear time - branching time spectrum I The semantics of concrete, sequential processes. In J. Bergstra, A. Ponse, and S. Smolka, editors, *Handbook of Process Algebra*, chapter 1, pages 3–99. Elsevier, 2001.

Appendix

Proposition 1 *For all $M \in \mathcal{D}(\texttt{<msc>})$, $[\![M]\!]_{msc}$ is deterministic.*

Proof We prove by contradiction. Suppose there exists an $M : \mathcal{D}(\texttt{<msc>})$ such that it is *not* the case that $[\![M]\!]_{msc}$ is deterministic. According to Definition 5, the process $[\![M]\!]_{msc}$ satisfies the following property:

$$\exists P' : \mathcal{P} \text{ such that}$$
$$[\![M]\!]_{msc} \overset{\varphi}{\Longrightarrow} P' \wedge P' \xrightarrow{\alpha} Q \wedge P' \xrightarrow{\alpha} R \wedge Q \neq R$$

The action α has three cases discussed as below:

- $\alpha = !a_{ij}$. The predicate $Q \neq R$ suggests that α denotes two different events in a bMSC, and the predicate $P' \xrightarrow{\alpha} Q \wedge P' \xrightarrow{\alpha} R$ requires no temporal ordering on the two events. In other words, the two events can never be on the same instance axis (without the *coregion* or *inline-expression* constructs). This leads to a contradiction because α has the form $!a_{ij}$, which denotes an output event on the instance *Pi*.
- Similar arguments are applicable when α has the other forms. They all lead to contradictions. $\square$

Proposition 2 *For $M_1, M_2 : \mathcal{D}(\texttt{<msc>})$, $[\![M_1]\!]_{msc} =_B [\![M_2]\!]_{msc}$ if and only if $[\![M_1]\!]_{msc} =_T [\![M_2]\!]_{msc}$*

Proof Due to Proposition 1, we know that for all bMSCs in our MSC domain $\mathcal{D}(\texttt{<msc>})$, their process expressions are deterministic. Hence this proposition can be rephrased as for two deterministic processes $P, Q \in \mathcal{P}$, $P =_B Q \Leftrightarrow P =_T Q$.

- $\Rightarrow$
  To prove $P =_T Q$, we show that $T(P) \subseteq T(Q)$ and $T(Q) \subseteq T(P)$, i.e.

  $$\forall t : Act^* \text{ such that } t \in T(P) \Rightarrow t \in T(Q) \wedge$$
  $$\forall t : Act^* \text{ such that } t \in T(Q) \Rightarrow t \in T(P)$$

  We use mathematical induction on the length of traces t, denoted by $|t|$, in $T(P)$. Here we only show $T(P) \subseteq T(Q)$. Similar arguments can be applied the other way around.
  First, when $|\,t\,| = 0$, t is an empty sequence ε hence $\varepsilon \in T(Q)$. Then we assume that when $|\,t\,| = n$, t has the form $\varphi \in Act^*$ and $t \in T(Q)$. According to Definition 6, we know that there exists a $Q' \in \mathcal{P}$ such that $Q \overset{\varphi}{\Longrightarrow} Q'$. When $|\,t\,| = n + 1$, we have to show that $t \in T(Q)$. Since $T(P)$ is a *prefix-closed* set, t must have the form $\varphi\alpha$ where $\alpha \in Act$. Due to Definition 5, we know $P \overset{\varphi}{\Longrightarrow} P' \xrightarrow{\alpha} P''$ for some P' and P'' in $\mathcal{P}$. From Lemma 1, $Q' =_B P'$, so $Q' \xrightarrow{\alpha} Q''$ for some Q''. Hence $\varphi\alpha \in T(Q)$.
- $\Leftarrow$
  Let $\mathcal{R}$ be the set containing the pairs of trace-equivalent processes, i.e.

  $$\mathcal{R} = \{(P, Q) : \mathcal{P} \times \mathcal{P} \mid P =_T Q\}$$

  Supposing that $P \xrightarrow{\alpha} P_1$ where $\alpha \in Act$, then we have a unique P_1 and $T(P_1) = \{\varphi : Act^* \mid \alpha\varphi \in T(P)\}$ where α is a singleton trace. Due to the assumption that $P =_T Q$, we have $Q \xrightarrow{\alpha} Q_1$ where Q_1 is unique and $T(Q_1) = \{\varphi : Act^* \mid \alpha\varphi \in T(Q)\}$. Hence $(P_1, Q_1) \in \mathcal{R}$. The same argument applies to (P_n, Q_n) where $P \overset{\varphi}{\Longrightarrow} P_n$ and $Q \overset{\varphi}{\Longrightarrow} Q_n$.
  $\mathcal{R}$ is symmetric due to the commutativity of set equivalence. Hence $\mathcal{R}$ is a bisimulation. $\square$