Journal of Universal Computer Science, vol. 5, no. 9 (1999), 610-620
submitted: 10/6/99, accepted: 15/7/99, appeared: 28/9/99 © Springer Pub. Co.

Synchronization Expressions and Languages

Kai Salomaa
Department of Computing and Information Science, Queen's University
Kingston, Ontario K7L 3N6, Canada

Sheng Yu
Department of Computer Science, University of Western Ontario
London, Ontario N6A 5B7, Canada
Email: syu@csd.uwo.ca

Abstract: Synchronization expressions (SEs) were originally developed as practical high-level constructs for specifying synchronization constraints between parallel processes. The family of synchronization languages was introduced to give a precise semantic description for synchronization expressions. In addition to its use for defining the meaning of SEs, the family of synchronization languages is interesting on its own from a formal languages point of view. We consider two variants of the definition of synchronization languages, and survey characterization results for the language families. Synchronization languages also provide us a systematic approach for the implementation and simplification of SEs.

Category: F.4.3

1 Introduction

Synchronization is a crucial part of parallel computation. However, synchronization mechanisms used in most parallel programming languages are at a too low level to be compatible with other constructs of high-level languages. Synchronization expressions (SE) were originally introduced in the ParC project [Govindarajan et al. 91] as high-level constructs for specifying synchronization constraints between parallel processes. Synchronization requests are specified as expressions of tags for statements that are to be constrained by the synchronization requirements. The detailed imposition of the synchronization requirements is left for the compiler in this approach.

The semantics of SEs is defined using synchronization languages. A synchronization language can be viewed as the set of correct executions (as controlled by the SE) of a distributed application where each action is split into two atomic parts, its start and termination. It can be argued [Guo et al. 94, Guo et al. 96] that synchronization languages are a more suitable semantic model for synchronization expressions than traditional models such as traces, Petri nets, or process algebra [Aalbersberg and Rozenberg 88, Hennessy 89, Sassone et al. 96]. An important feature of the semantics defined by synchronization languages is that it splits each action into two parts, the start and the termination. For instance, in trace semantics [Diekert and Métivier 97] parallel execution is interpreted as $a \parallel b = ab + ba$, but in order to control the execution of parallel processes it is necessary to distinguish sequences like ab and ba from real parallel execution. By having the start and termination of an action as separate instantaneous parts such distinctions can be made. Thus SEs provide a natural interpretation of a synchronization control that allows two actions to (concretely) overlap in time. Furthermore, synchronization languages can specify that certain occurrences of given processes are parallel whereas other occurrences need to be executed in a specified sequential order. The relationship of and the differences between synchronization languages and various formalisms used in concurrency will be discussed in [Ciobanu et al. 00].

It was shown in [Guo et al. 94, Guo et al. 96] that synchronization languages are closed under a number of naturally defined rewriting rules and there it was also conjectured that synchronization languages would consist of the subset of regular languages closed under these rules and satisfying the so called start-termination property. The conjecture was proved for languages expressing synchronization between two distinct actions (for a two-letter alphabet) by Clerbout, Roos and Ryl [Clerbout et al. 99, Ryl et al. 97] and they showed that the conjecture is false in general. Furthermore, [Ryl et al. 98] establishes a more general negative result showing that these languages cannot be characterized by any set of rewriting rules.

By generalizing the syntactic definition of SEs and modifying the semantics [Salomaa and Yu 98] considered a new definition of synchronization languages. The new definition has the advantage that it allows us to eliminate the intuitively less well motivated rewriting rules describing closure properties of the synchronization languages and, furthermore, the new set of rules always preserves regularity. This gives us hope to avoid the negative results that were obtained for the original definition of synchronization languages. The approach allows us to obtain an exact characterization for the finite synchronization languages in terms of simple semi-commutation rules [Salomaa and Yu 98] and a characterization of the images of synchronization languages under st-morphisms [Ryl et al. 99]. However, it is not known whether the family of synchronization languages is closed under st-morphisms and, thus, it remains an open question whether the rewriting rules can be used for the general family of synchronization languages.

Besides giving rise to interesting questions in formal language theory, synchronization languages provide us with a systematic method for implementing SEs in a parallel programming language such as ParC. The corresponding synchronization language (SL) describes exactly how an SE is to be implemented in the parallel programming language ParC. Using this formalism, relations such as equivalence and inclusion between SEs can be easily understood and tested. Thus the synchronization languages provide a systematic approach for the implementation and simplification of SEs. In the last section we briefly discuss the practical implementation of SEs, and some difficulties encountered in it.

2 Definitions

Here we recall some definitions and notation related to regular expressions and rewriting systems, for more details the reader may consult [Book and Otto 95, Salomaa 73, Yu 97]. The set of finite words over a finite alphabet Σ is denoted Σ^*, λ is the empty word, and $\Sigma^+ = \Sigma^* - \{\lambda\}$. The catenation of languages $L_1, L_2 \subseteq \Sigma^*$ is $L_1 \cdot L_2 = \{w \in \Sigma^* \mid (\exists v_i \in L_i, i = 1,2)\ w = v_1 v_2\}$ and the catenation of n copies of $L \subseteq \Sigma^*$ is L^n ($n \geq 0$). Note that $L^0 = \{\lambda\}$. The Kleene star of a language L is $L^* = \cup_{i=0}^{\infty} L^i$.

The *shuffle* of words $u, v \in \Sigma^*$ is the language $\omega(u, v) \subseteq \Sigma^*$ consisting of all words that can be written in the form $u_1 v_1 \cdots u_n v_n$, $n \geq 0$, where $u = u_1 \cdots u_n$, $v = v_1 \cdots v_n$, $u_i, v_i \in \Sigma^*$, $i = 1, \ldots, n$. The shuffle operation is extended for languages in the natural way:

$$\omega(L_1, L_2) = \bigcup_{w_1 \in L_1, w_2 \in L_2} \omega(w_1, w_2),$$

where $L_1, L_2 \subseteq \Sigma^*$.

A *string-rewriting system* (or Thue system) over Σ is a finite set R of rules $u \to v$, $u, v \in \Sigma^*$. The rules of R define the single step reduction relation $\to_R \subseteq \Sigma^* \times \Sigma^*$ as follows. For $w_1, w_2 \in \Sigma^*$, $w_1 \to_R w_2$ if and only if there exists a rule $u_1 \to u_2 \in R$ and $r, s \in \Sigma^*$ such that $w_i = r u_i s$, $i = 1, 2$. The *reduction relation* of R is the reflexive and transitive closure of $\to_R$ and it is denoted $\to_R^*$.

We define synchronization expressions using the extended definition from [Salomaa and Yu 98]. The operators $\to$, $\|$, $|$, $\&$ and $*$ are called, respectively, the *sequencing, join, selection, intersection,* and *repetition* operators. Later we give also the original more restricted definition that differs only in the use of the join operator.

The intuitive meaning of the operations will be apparent from the semantic interpretation given below. For easier readability we omit the outermost parentheses of an expression. All the four binary operations will be associative, so usually also other parentheses may be omitted.

Let Σ be an alphabet such that each symbol of Σ denotes a distinct process. In order to define the meaning of synchronization expressions, for each $a \in \Sigma$ we associate symbols a_s and a_t to denote, respectively, the *start* and *termination* of the process. Also, we denote

$$\Sigma_s = \{a_s \mid a \in \Sigma\}, \quad \Sigma_t = \{a_t \mid a \in \Sigma\}.$$

The synchronization languages satisfy the condition that the termination of an occurrence of a process always precedes the start of the next occurrence. The condition is defined formally as follows. For $a \in \Sigma$ let $p_a : (\Sigma_s \cup \Sigma_t)^* \longrightarrow \{a_s, a_t\}^*$ be the morphism determined by the conditions

$$p_a(x) = \begin{cases} x, & \text{if } x \in \{a_s, a_t\}, \\ \lambda, & \text{if } x \notin \{a_s, a_t\}. \end{cases}$$

A word $w \in (\Sigma_s \cup \Sigma_t)^*$ is said to satisfy the start-termination condition (or st-condition for short) if for all $a \in \Sigma$, $p_a(w) \in (a_s a_t)^*$. The st-condition means that autoconcurrency of processes is not permitted. If we would allow sequences of the form $\ldots a_s \ldots a_s \ldots a_t \ldots a_t \ldots$ (where the dots denote sequences not containing symbols a_s, a_t) we could not know which a_t occurrence corresponds to which a_s occurrence (without adding, for instance, pointers to the sequences). Naturally we can allow a finite amount of autoconcurrency simply by using distinct symbols for the distinct occurrences of a process.

If w satisfies the st-condition, then w belongs to the shuffle of some languages $((a_1)_s(a_1)_t)^*$, $\ldots$, $((a_k)_s(a_k)_t)^*$ where $a_i \neq a_j$, when $i \neq j$, $i, j = 1, \ldots k$. The set of all words over Σ satisfying the st-condition is denoted W_Σ^{st} and subsets of W_Σ^{st} are called *st-languages.*

The synchronization expressions over an alphabet Σ and the languages denoted by the expressions are defined inductively as follows.

Definition 1. The set of *synchronization expressions* over alphabet Σ, SE(Σ), is the smallest subset of $(\Sigma \cup \{\phi, \rightarrow, \&, |, \|, *, (,)\})^*$ defined inductively by the following rules. The synchronization language denoted by $\alpha \in \mathrm{SE}(\Sigma)$ is $L(\alpha)$.

(i) $\Sigma \cup \{\phi\} \subseteq \mathrm{SE}(\Sigma)$. $L(\phi) = \emptyset$ and $L(a) = \{a_s a_t\}$ when $a \in \Sigma$.
(ii) If $\alpha_1, \alpha_2 \in \mathrm{SE}(\Sigma)$ then $(\alpha_1 \rightarrow \alpha_2) \in \mathrm{SE}(\Sigma)$ and $L(\alpha_1 \rightarrow \alpha_2) = L(\alpha_1) \cdot L(\alpha_2)$.
(iii) If $\alpha_1, \alpha_2 \in \mathrm{SE}(\Sigma)$ then $(\alpha_1 \| \alpha_2) \in \mathrm{SE}(\Sigma)$ and
$L(\alpha_1 \| \alpha_2) = \omega(L(\alpha_1), L(\alpha_2)) \cap W_\Sigma^{\mathrm{st}}$.
(iv) If $\alpha_1, \alpha_2 \in \mathrm{SE}(\Sigma)$ then $(\alpha_1 \mid \alpha_2) \in \mathrm{SE}(\Sigma)$ and $L(\alpha_1 \mid \alpha_2) = L(\alpha_1) \cup L(\alpha_2)$.
(v) If $\alpha_1, \alpha_2 \in \mathrm{SE}(\Sigma)$ then $(\alpha_1 \& \alpha_2) \in \mathrm{SE}(\Sigma)$ and $L(\alpha_1 \& \alpha_2) = L(\alpha_1) \cap L(\alpha_2)$.
(vi) If $\alpha \in \mathrm{SE}(\Sigma)$ then $\alpha^* \in \mathrm{SE}(\Sigma)$ and $L(\alpha^*) = L(\alpha)^*$.

For concrete examples of synchronization expressions see the last section or [Guo et al. 94, Guo et al. 96]. When defining the interpretation of the parallelization operator in (iii) we use intersection with the set of st-words. The definition implies that $L(a \| a) = L(a \rightarrow a)$.

The original definition of synchronization expressions required that the expressions appearing as arguments of the parallelization operator must be over disjoint alphabets. The set of *restricted synchronization expressions,* $\mathrm{SE}_r(\Sigma)$, is defined as in Definition 1 by substituting $\mathrm{SE}_r(\Sigma)$ everywhere for SE(Σ) and modifying the condition (iii) as follows. We denote by alph(α) the set of symbols of Σ appearing in the expression α. (alph(α) can naturally be defined inductively following Definition 1.)

(iii)' If $\alpha_1, \alpha_2 \in \mathrm{SE}_r(\Sigma)$ and $\mathrm{alph}(\alpha_1) \cap \mathrm{alph}(\alpha_2) = \emptyset$ then $(\alpha_1 \| \alpha_2) \in \mathrm{SE}_r(\Sigma)$ and $L(\alpha_1 \| \alpha_2) = \omega(L(\alpha_1), L(\alpha_2))$.

In restricted synchronization expressions, always when $\alpha_1 \| \alpha_2$ is defined each word of $\omega(L(\alpha_1), L(\alpha_2))$ satisfies the st-condition so we do not need the intersection with the set W_Σ^{st}.

The family of *synchronization languages* $\mathcal{L}(\mathrm{SE})$ (respectively, *restricted synchoronization languages* $\mathcal{L}(\mathrm{SE}_r)$) consists of all languages $L(\alpha)$ where $\alpha \in \mathrm{SE}(\Sigma)$ (respectively, $\alpha \in \mathrm{SE}_r(\Sigma)$) for some alphabet Σ. Directly from the definition it follows that $\mathcal{L}(\mathrm{SE}_r) \subseteq \mathcal{L}(\mathrm{SE})$. Furthermore, the inclusion is strict since, for instance, the expression $(b \rightarrow a^*) \| (a^* \rightarrow c)$ denotes a language that is not in $\mathcal{L}(\mathrm{SE}_r)$ [Ryl et al. 97, Salomaa and Yu 98].

A regular expression α over an alphabet $\Sigma_s \cup \Sigma_t$ is said to be *well-formed* if always when β^* is a subexpression of α, the language denoted by β is an st-language. A language L over the alphabet $\Sigma_s \cup \Sigma_t$ is *well-formed* if L is denoted by some well-formed regular expression α. When trying to obtain a characterization of the synchronization languages in terms of closure under rewriting rules, it turns out that the problematic cases are languages that are not well-formed.

3 Rewriting Rules for Synchronization Languages

Synchronization languages are regular languages satisfying the st-condition and furthermore they are closed under certain types of semi-commutation rules and their extensions. It would be very useful if we could exactly characterize the synchronization languages in terms of closure under some simple rewriting rules. A characterization could yield more efficient algorithms for testing the equivalence of SEs since in many cases it would be easy to test whether a given finite automaton accepts a language satisfying the given semi-commutation properties. We can effectively decide whether a given regular language is closed under a given semi-commutation system [Clerbout et al. 95].

Definition 2. Let Σ be an alphabet. We define the rewriting system $R_\Sigma \subseteq (\Sigma_s \cup \Sigma_t)^* \times (\Sigma_s \cup \Sigma_t)^*$ to consist of the following rules, where $a, b \in \Sigma$, $a \neq b$:

$$\text{(R1) } a_s b_t \to b_t a_s, \ \text{(R2) } a_s b_s \to b_s a_s, \ \text{(R3) } a_t b_t \to b_t a_t.$$

The set S_Σ is defined to consist of all rules

$$\begin{aligned} &(a_1)_t \cdots (a_i)_t (a_1)_s \cdots (a_i)_s (b_1)_t \cdots (b_j)_t (b_1)_s \cdots (b_j)_s \\ &\to (b_1)_t \cdots (b_j)_t (b_1)_s \cdots (b_j)_s (a_1)_t \cdots (a_i)_t (a_1)_s \cdots (a_i)_s, \end{aligned} \tag{1}$$

where where $a_1, \ldots, a_i, b_1, \ldots, b_j$ are pairwise distinct elements of Σ, $i, j \geq 1$. Then we denote

$$R'_\Sigma = R_\Sigma \cup S_\Sigma.$$

The rules of R_Σ define a *semi-commutation relation* [Clerbout et al. 95] on Σ. By symmetry the rules (R2) and (R3) could be written also as two-directional rules.

If R is a rewriting system over Σ and $L \subseteq \Sigma^*$ we denote $\Delta^*_R(L) = \{u \in \Sigma^* \mid (\exists v \in L) v \to^*_R u\}$. We say that L is closed under R if $L = \Delta^*_R(L)$. The following results hold.

Theorem 3. *Let $L \subseteq \Sigma^*$.*

(a) [Salomaa and Yu 98] *If L is a synchronization language, then L is closed under R_Σ.*
(b) [Guo et al. 94, Guo et al. 96] *If L is a restricted synchronization language, then L is closed under R'_Σ.*

The intuitive explanation why restricted synchronization languages are closed under rules (1) is that if we have a subword of the form $\ldots a_t a_s b_t b_s \ldots$ (or the given more general form), then all the following pairs of the given occurrences of the symbols are parallel: (a_t, b_t), (a_s, b_s) and (a_s, b_t), i.e., they correspond to symbols occuring in different arguments of the parallelization operator. On the other hand, since the different arguments may not contain occurrences of the same symbol of Σ, the only possibility is that the symbols a_t, a_s correspond to occurrences in one of the arguments, and b_t, b_s in the other argument. The proof showing closure under the rules of R_Σ is similar but simpler.

It was conjectured [Guo et al. 96] that closure under the rewriting system R'_Σ together with the st-condition exactly characterizes the restricted synchronization languages. The conjecture was proved in [Clerbout et al. 99] for alphabets corresponding to two actions.

Theorem 4. [Clerbout et al. 99] *A language $L \subseteq \{a_s, a_t, b_s, b_t\}^*$ is a restricted synchronization language if and only if L is regular, satisfies the st-condition and is closed under R'_Σ.*

At the same time [Clerbout et al. 99, Ryl et al. 97] established that the conjecture does not hold in general. The counter-example given there is the language

$$L_1 = \Delta^*_{R'_\Sigma}(b_s(a_s a_t)^* c_s b_t (a_s a_t)^* c_t)$$

which is a regular st-language and it can be shown that L_1 is not a restricted synchronization language. The intuitive idea is that restricted synchronization expressions cannot differentiate in L_1 between instances of the action a that occur during b and during c, respectively. Formally the fact that $L_1 \notin \mathcal{L}(\mathrm{SE}_r)$ is proved by establishing a so called switching property for restricted synchronization languages [Clerbout et al. 99, Ryl et al. 97].

One can show that $\mathcal{L}(\mathrm{SE}_r)$ is closed under certain infinite extensions of R'_Σ that satisfy nice projection properties onto subalphabets. However, the hope of obtaining a rewriting characterization for this class was destroyed by the following result.

Theorem 5. [Ryl et al. 98, Clerbout et al. 98] *There does not exist any rewriting system Q such that each $L \in \mathcal{L}(\mathrm{SE}_r)$ is closed under Q and each regular st-language closed under Q is in $\mathcal{L}(\mathrm{SE}_r)$.*

It may be noted another undesirable property of the restricted synchronization languages is that the corresponding rewriting rules (1) do not, in general, preserve regularity of st-languages. Consider the language

$$L_2 = a_s b_s (a_t a_s b_t b_s)^* b_t a_t.$$

Using the rule $a_t a_s b_t b_s \to b_t b_s a_t a_s$ (and the semi-commutation rules of Definition 2) every word of L_2 can be rewritten to a word $(b_s b_t)^m (a_s a_t)^m$, and this observation easily implies that $\Delta^*_{R'_\Sigma}(L_2)$ is not regular.

Due to the negative result of Theorem 5 it is natural to consider the more general definition of SEs given in Definition 1. This allows us to drop the rules (1) from the rewriting system describing closure properties of the synchronization languages. Relying on general properties of semi-commutation systems [Clerbout et al. 95] we can then prove:

Theorem 6. [Salomaa and Yu 98] *If $L \subseteq (\Sigma_s \cup \Sigma_t)^*$ is a regular st-language, then $\Delta^*_{R_\Sigma}(L)$ is also regular.*

Using the more general definition, in the case of finite languages we can strengthen the result of Theorem 3(a) into an "if and only if" condition.

Theorem 7. [Salomaa and Yu 98] *Assume that L is a finite language over $\Sigma_s \cup \Sigma_t$. Then $L \in \mathcal{L}(\mathrm{SE})$ if and only if L is an st-language closed under R_Σ.*

We conjecture that the characterization result of Theorem 7 can be extended for the family of well-formed languages.
Conjecture. Assume that $L \subseteq (\Sigma_s \cup \Sigma_t)^*$ is a well-formed regular language. Then L is a synchronization language iff L is an st-language closed under R_Σ.

We do not know whether a similar characterization result holds for the general family of synchronization languages. Even for some very simple looking regular expressions it can be difficult to find a synchronization expression for the corresponding R_Σ-closure language. By Theorem 6 the language

$$L_3 = \Delta^*_{R_\Sigma}(a_s(b_s a_t a_s b_t)^* a_t)$$

is regular, and clearly L_3 is an st-language. The language is denoted by the SE $((a \rightarrow b)^* \rightarrow a) \parallel (a \parallel b)^*$. However, there seems to be little structural correspondence between the SE and the original regular expression $a_s(b_s a_t a_s b_t)^* a_t$ which is not well-formed. The paper [Ryl et al. 99] gives an explicit regular st-language closed under R_Σ for which no synchronization expression is known.

Interestingly it is shown in [Ryl et al. 98, Ryl et al. 99] that closure under R_Σ exactly characterizes the images of synchronization languages under functions called st-morphisms.

Definition 8. Let Σ and Ω be alphabets and $\varphi : \Sigma^* \longrightarrow \Omega^*$ a strictly alphabetical morphism (that is, $\varphi(a) \in \Omega$ for each $a \in \Sigma$). The morphism φ is extended in the natural way to a morphism $(\bar{\varphi}) : (\Sigma_s \cup \Sigma_t)^* \longrightarrow (\Omega_s \cup \Omega_t)^*$ by setting $\bar{\varphi}(a_s) = \varphi(a)_s$ and $\bar{\varphi}(a_t) = \varphi(a)_t$ for each $a \in \Sigma$.

With each strictly alphabetical morphism $\varphi : \Sigma^* \longrightarrow \Omega^*$, we associate a function $\hat{\varphi}$ called an *st-morphism:*

$$\hat{\varphi} = \{(u, \bar{\varphi}(u)) \mid u \in W^{\rm st}_\Sigma \text{ and } \bar{\varphi}(u) \in W^{\rm st}_\Omega\}.$$

The family of st-morphisms is denoted $\Phi_{\rm st}$.

Note that $\hat{\varphi}$ is equal to the relation $(\cap W^{\rm st}_\Omega) \circ \bar{\varphi} \circ (\cap W^{\rm st}_\Sigma)$ and an st-morphism is not, strictly speaking, a morphism. For instance, if $\varphi(a) = \varphi(b) = c$ then $\hat{\varphi}(a_s b_s a_t b_t)$ is not defined.

Theorem 9. [Ryl et al. 99] $\Phi_{\rm st}(\mathcal{L}(\text{SE}))$ *is equal to the family of regular languages* $L \subseteq (\Sigma_s \cup \Sigma_t)^*$ *that satisfy the st-condition and are closed under the rewriting system* R_Σ.

According to Theorem 9, the question whether synchronization languages are exactly the regular st-languages closed under R_Σ is equivalent to determining whether the equality $\Phi_{\rm st}(\mathcal{L}(\text{SE})) = \mathcal{L}(\text{SE})$ holds In [Ryl et al. 98] it is shown that $\Phi_{\rm st}(\mathcal{L}(\text{SE}_r)) = \Phi_{\rm st}(\mathcal{L}(\text{SE}))$ which implies that the restricted synchronization languages are not closed under st-morphisms.

4 Implementation of synchronization expressions

In this section, we explain how the ideas of SEs and SLs can be used and implemented in a programming language that is designed for a parallel or/and distributed computing environment. The implementation follows strictly our semantic interpretation of SEs using SLs. More specifically, for an SE, we construct a finite automaton that accepts the prefix language of the SL denoted by the SE at compile time. Then the constructed automaton is used to impose the synchronization constraints specified by the SE at runtime.

In the following, we will use examples written in the *ParC* concurrent programming language [Govindarajan et al. 91]. The examples will be explained in detail assuming that the reader knows the C programming language but not

ParC. In *ParC*, (simple or compound) statements are considered as the basic elements of synchronization. Statements that are involved in synchronization are represented in SEs by *statement tags*. In other words, the alphabet of an SE is a set of statement tags. This is a design decision specific to *ParC* and need not be true in general. Especially in an object-oriented concurrent programming language, it may be more appropriate that object operations rather than statements should be the basic elements of synchronization specified by SEs. However, the implementation principle described below would still be applicable as long as a symbol in an SE represents an entire execution of a process.

We provide in the following a simple example to show how a ParC program with synchronization expressions is translated into a Sequent C program with system calls [Osterhaug 89]. We first give the ParC program with a detailed explanation. Then we give the generated code in Sequent C with system calls.

```
main(){
    shared int w;
    int r;
    int f();
    tag a, b;
    restrict (a->b)*;

    pexec{
        {/* ...... */
         b:: r = w;
         /* ...... */
        }
        {/* ...... */
         a:: w = f();
         /* ...... */
        }
    }
}
```

We first look at the `pexec` statement, i.e., the *parallel execution* statement. This language construct denotes that all statements that are directly under the scope can be executed in parallel. In the above program, there are two compound statements directly under the `pexec` statement, which can be considered as two parallel processes. In the first process, the statement "r = w;" is tagged with `b`, and in the second process, "w = f();" is tagged with `a`. We assume that each of the two statements is in a loop. The `restrict` statement above the `pexec` statement specifies a synchronization expression `(a->b)*`, meaning that the `a` and `b` statements can be executed only in the following sequence: `a, b, a, b,` …, i.e., the ith instance of `b` cannot start its execution before the ith instance of `a` finishes and the $(i+1)$th instance of `a` cannot start its execution before the ith instance of `b` finishes, $i \geq 1$. Also in the declarations, `w` is declared as a shared integer variable, which means that the two appearances of `w` in the two parallel processes, respectively, are the same variable. The integer variable `i` is not shared. So, it has different copies in different parallel processes in the scope. And the statement `tag a, b;` declares that both `a` and `b` are statement tags.

A deterministic finite automaton (DFA) is built corresponding to the SE `(a->b)*`, which accepts the prefix language of the SL defined by the SE. The DFA is shown in Figure 1.

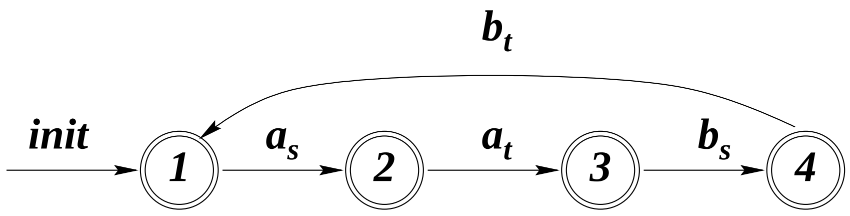

Figure 1: DFA for the SE `(a->b)*`

To ensure that the the state update can only be performed exclusively by one process, we build a macro called

$$\texttt{wait\_and\_set(int state; int } i_1, j_1\texttt{; ...; int } i_n, j_n\texttt{).}$$

for `n > 0`. The macro waits until the condition "`state ==` i_t", for some t, $1 \leq t \leq n$, becomes true and then sets "`state =` j_t". Instead of using a variable-length list `int` i_1, j_1`; ...; int` i_n, j_n, the actual implementation of `wait_and_set` may use a pointer to a table (an array) for the transitions. Here we use a list just to keep the notation simple. The macro is an indivisible (exclusive) operation and can be implemented using primitive synchronization mechanisms of the system.

Then the state transitions of the DFA correspond to the following Sequent C statements or macros:

- Initialization `_state = 1;`
- The a_s transition `wait_and_set(_state; 1, 2);`
- The a_t transition `wait_and_set(_state; 2, 3);`
- The b_s transition `wait_and_set(_state; 3, 4);`
- The b_t transition `wait_and_set(_state; 4, 1);`

The following is the code generated from the ParC program:

```
#include<parallel/microtask.h>
#include<parallel/parallel.h>
#include<sys/wait.h>
#include<sys/types.h>
#include<stdio.h>

main() {
shared int w;
int k, r;
shared int _state=1;
int f();
```

```
        {
        int _pw, _pid;

            if ((_pid=fork()) == 0){
                /* ...... */
                wait_and_set(_state; 3, 4);
                    r = w;
                wait_and_set(_state; 4, 1);
                /* ...... */
                exit(0);
            }
            else if (_pid < 0){
                printf("fork error\n");
                exit(1);
            }
            /* ...... */
            wait_and_set(_state; 1, 2);
            w = f(k);
            wait_and_set(_state; 2, 3);
            /* ...... */
        }
    }
```

For a slightly more complicated SE "`(a || b)->c`", a corresponding DFA is shown in Figure 2.

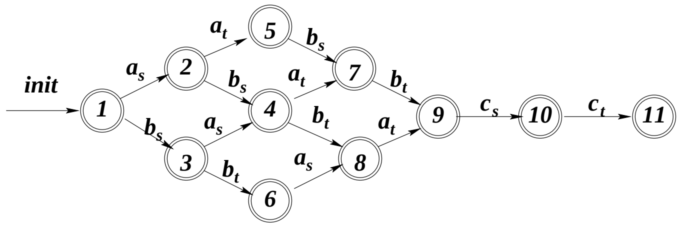

Figure 2: DFA for the SE `(a || b)->c`

In this case the code generated for the `a` statement would be

```
wait_and_set(_state; 1, 2; 3, 4; 6, 8);
............ /* the a statement itself */
wait_and_set(_state; 2, 5; 4, 7; 8, 9);
```

Similarly, the code generated for `b` statement would be

```
wait_and_set(_state; 1, 3; 2, 4; 5, 7);
............ /* the b statement itself */
wait_and_set(_state; 3, 6; 4, 8; 7, 9);
```

And the following is the code for c:

```
wait_and_set(_state; 9, 10);
............ /* the c statement itself */
wait_and_set(_state; 10, 11);
```

The above two simple examples illustrate the basic idea of implementing SEs using synchronization languages. There are many other issues concerning implementation. Below we briefly discuss a couple of them.

The first is the *size-explosion problem* of DFA. For many practical synchronization problems, the sizes of the DFA can be too large to implement efficiently. We suggest the use of alternating finite automata (AFA) instead of DFA in the implementation of SEs. The use of AFA can significantly increase the space efficiency of implementation [Salomaa et al. 98, Huerter et al. 99].

The checking of states may become the bottleneck of synchronization since it is done sequentially. However, this does not necessarily need to be the case. Note that SEs within different concurrent blocks, respectively, can be implemented by automata that actually run concurrently.

Conflicting synchronization constraints and deadlock conditions caused by the definitions of SEs in the same scope can be examined by checking the intersection of the synchronization languages defined by the SEs and the execution sequences defined by the program flow. The precise semantic definition of SEs makes the checking process conceptually clear and feasible to implement in practice.

Acknowledgement

This work has been supported by the Natural Sciences and Engineering Research Council of Canada Grants OGP0147224 and OGP0041630.

References

[Aalbersberg and Rozenberg 88] Aalbersberg, I.J., Rozenberg, G.: "Theory of traces"; *Theoret. Comput. Sci.* **60** (1988) 1–82.

[Book and Otto 95] Book, R.V., Otto, F.: *String-Rewriting Systems;* Texts and Monographs in Computer Science, Springer-Verlag, 1993.

[Ciobanu et al. 00] Ciobanu, G., Salomaa, K., Yu, S.: "Synchronization languages and ST semantics"; manuscript in preparation.

[Clerbout et al. 95] Clerbout, M., Latteux, M., Roos, Y.: "Semi-commutations"; in: *The Book of Traces.* (V. Diekert, G. Rozenberg, eds.) Chapter 12, pp. 487–552, World Scientific, Singapore, 1995.

[Clerbout et al. 98] Clerbout, M., Roos, Y., Ryl, I.: "Langages de synchronization et systèmes de réécriture"; Tech. Rep. IT-98-311, Université des Sciences et Technologies de Lille, 1998.

[Clerbout et al. 99] Clerbout, M., Roos, Y., Ryl, I.: "Synchronization languages"; *Theoret. Comput. Sci.* **215** (1999) 99–121.

[Diekert and Métivier 97] Diekert, V., Métivier, Y.: "Partial commutation and traces"; in: *Handbook of Formal Languages, Vol. III.* (G. Rozenberg, A. Salomaa, eds.) pp. 457–533, Springer-Verlag, 1997.

[Govindarajan et al. 91] Govindarajan, R., Guo, L., Yu, S., Wang, P.: "ParC Project: Practical constructs for parallel programming languages"; Proc. of the 15th Annual IEEE International Computer Software & Applications Conference, 1991, pp. 183–189.

[Guo et al. 94] Guo, L., Salomaa, K., Yu, S.: "Synchronization expressions and languages"; Proc. of the 6th IEEE Symposium on Parallel and Distributed Processing, (Dallas, Texas). IEEE Computer Society Press, 1994, pp. 257–264.

[Guo et al. 96] Guo, L., Salomaa, K., Yu, S.: "On synchronization languages"; *Fundamenta Inform.* **25** (1996) 423–436.

[Hennessy 89] Hennessy, M.: *Algebraic Theory of Processes;* The MIT Press, Cambridge, Mass., 1989.

[Huerter et al. 99] Huerter, S., Salomaa, K., Wu, X., Yu, S.: "Implementing Reversed Alternating Finite Automaton (r-AFA) Operations"; Proceedings of the Third International Workshop on Implementing Automata (WIA'98), Lect. Notes Comput. Sci. **1660**, Springer-Verlag, 1999, pp. 69–81.

[Osterhaug 89] Osterhaug, A. (ed.): *Guide to Parallel Programming — On Sequent Computer Systems;* Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[Ryl et al. 97] Ryl, I., Roos, Y., Clerbout, M.: "Partial characterization of synchronization languages"; Proc. of 22nd International Symposium on Mathematical Foundations of Computer Science (MFCS'97), Lect. Notes Comput. Sci. **1295** (1997) 209–218.

[Ryl et al. 98] Ryl, I., Roos, Y., Clerbout, M.: "About synchronization languages"; Proc. of 23rd International Symposium on Mathematical Foundations of Computer Science (MFCS'98), Lect. Notes Comput. Sci. **1450**, Springer-Verlag, 1998, pp. 533–542.

[Ryl et al. 99] Ryl, I., Roos, Y., Clerbout, M.: "Generalized synchronization languages"; Proc. of 12th International Symposium on Fundamentals of Computation Theory (FCT'99), Lect. Notes Comput. Sci. **1684**, Springer-Verlag, 1999, pp. 451–462.

[Salomaa 73] Salomaa, A.: *Formal Languages;* Academic Press, 1973.

[Salomaa et al. 98] Salomaa, K., Wu, X., Yu, S.: "An Efficient Implementation of Regular Languages Using r-AFA"; Proceedings of the Second International Workshop on Implementing Automata (WIA'97), Lect. Notes Comput. Sci. **1436**, Springer-Verlag, 1998, pp. 176–184.

[Salomaa and Yu 98] Salomaa, K., Yu, S.: "Synchronization expressions with extended join operation"; *Theoret. Comput. Sci.* **207** (1998) 73–88.

[Sassone et al. 96] Sassone, V., Nielsen, M., Winskel, G.: "Models of concurrency: Towards a classification"; *Theoret. Comput. Sci.* **170** (1996) 297–348.

[Yu 97] Yu, S.: "Regular languages"; in: *Handbook of Formal Languages, Vol. I.* (G. Rozenberg, A. Salomaa, eds.) pp. 41–110, Springer-Verlag, 1997.