Derivative Based Methods for Deciding SKA and SKAT

Sabine Broda Sílvia Cavadas Nelma Moreira
CMUP & DCC, Faculdade de Ciências da Universidade do Porto
Rua do Campo Alegre, 4169-007 Porto, Portugal

Technical Report Series: DCC-2014-10
Version 1.0 May 2014

Departamento de Ciência de Computadores
&
Faculdade de Ciências da Universidade do Porto
Rua do Campo Alegre, 1021/1055,
4169-007 PORTO,
PORTUGAL
Tel: 220 402 900 Fax: 220 402 950
http://www.dcc.fc.up.pt/Pubs/

Derivative Based Methods for Deciding SKA and SKAT

Sabine Broda Sílvia Cavadas Nelma Moreira
CMUP & DCC, Faculdade de Ciências da Universidade do Porto
Rua do Campo Alegre, 4169-007 Porto, Portugal

September 1, 2014

Abstract

Synchronous Kleene algebra (SKA) is a decidable framework that combines Kleene algebra (KA) with a synchrony model of concurrency. Elements of SKA can be seen as processes taking place within a fixed discrete time frame and that, at each time step, may execute one or more basic actions or then come to a halt. The extension synchronous Kleene algebra with tests (SKAT) combines SKA with a boolean algebra. Both algebras were introduced by C. Prisicariu, who proved the completeness of SKA axioms, and thus decidability, through a Kleene theorem based on the classical Thompson ε-NFA construction. Using the notion of partial derivatives, we present a new decision procedure for SKA terms equivalence. The results are extended for SKAT considering automata with transitions labeled by boolean expressions instead of atoms. This work extends previous one done for KA and KAT, where derivative based methods have been used in feasible algorithms for testing terms equivalence.

Keywords: Synchronous Kleene Algebra, Concurrency, Equivalence, Derivative

1 Introduction

Synchronous Kleene algebra (SKA) combines Kleene algebra (KA) with the synchrony model of concurrency of R. Milner's SCCS calculus [16]. Synchronous here means that two concurrent processes execute a single action simultaneously at each time instant of a unique global clock. Although this synchrony model seems to be a very weak model of concurrency when compared with asynchronous interleaving models, its equational framework is powerful and the SCCS calculus includes the CCS calculus as a sub-calculus. It also models the Esterel programming language [4], a tool used by the industry.

SKA was introduced by C. Prisicariu [20]. It consists of a KA to which a synchrony operator and a notion of basic action are added. In its standard model, a process is seen as a set of synchronous strings, each letter of a string being a set of basic actions executed in a single time step. Using a Kleene's style theorem, C. Prisicariu proved that the SKA axioms were complete and from that derived the decidability of the equational theory. He also generalized Kleene algebra with tests (KAT) [12], an equational system that extends Kleene algebra with Boolean algebra. KAT is specially suited to capture and verify properties of simple imperative programs and, in particular, subsumes propositional Hoare logic [13]. For the resulting algebra, called synchronous Kleene algebra with tests (SKAT), the models considered were sets of guarded synchronous strings and completeness was also proved using the so called automata on guarded synchronous strings. SKAT can be seen as an alternative to Hoare logic for reasoning about parallel programs with shared variables in a synchronous system.

Decision procedures for Kleene algebra terms equivalence has been a subject of intense research in recent years. This is partially motivated by the fact that regular expressions can be seen as a program logic that allows to express nondeterministic choice, sequence, and finite iteration of programs. Many proposed procedures decide equivalence based on the computation of a bisimulation (or a bisimulation up-to) between the two expressions [1, 18, 18, 6, 21]. Broda et al. studied the

average size of derivative based automata both for KA and KAT [7]. For KAT terms, a coalgebraic decision procedure was presented by D. Kozen [15]. There, derivatives are considered with respect to symbols vp where p is an action symbol but v corresponds to a valuation of the boolean tests. This induces an exponential blow-up on the number of states or transitions of the automata and an accentuated exponential complexity when testing the equivalence of two KAT expressions (as noted in [19, 2]). A. Silva [22] introduced a class of automata over guarded strings that avoids that blow-up. Broda et al. studied the average size of some automata of that class [7] and extended finite automata equivalence decision procedures to that class [8]. In this paper we continue this line of work and present new decision procedures for SKA and SKAT equivalence, based on the notion of partial derivatives. In particular, for SKAT we introduce a class of automata over guarded synchronous strings where transitions are labeled by boolean expressions instead of valuations.

2 Deciding Synchronous Kleene Algebra

First we review some concepts related with SKA. A *Kleene algebra* (KA) is an algebraic structure $(\mathcal{A}, +, \cdot, ^*, 0, 1)$, where $+$ and $\cdot$ are binary operations on $\mathcal{A}$, * is a unary operation on $\mathcal{A}$, and 0 and 1 belong to $\mathcal{A}$, such that $(\mathcal{A}, +, \cdot, 0, 1)$ is an idempotent semiring, i.e. satisfies axioms (1)-(9) below, and * satisfies axioms and * satisfies axioms (10)-(13) below. The natural order $\leq$ in $(\mathcal{A}, +, \cdot, 0, 1)$ is defined by $\alpha \leq \beta$ if and only if $\alpha + \beta = \beta$.

(1) $\alpha + (\beta + \gamma) = (\alpha + \beta) + \gamma$
(2) $\alpha + \beta = \beta + \alpha$
(3) $\alpha + 0 = 0 + \alpha = \alpha$
(4) $\alpha + \alpha = \alpha$
(5) $\alpha \cdot (\beta \cdot \gamma) = (\alpha \cdot \beta) \cdot \gamma$
(6) $\alpha \cdot 1 = 1 \cdot \alpha = \alpha$
(7) $\alpha \cdot (\beta + \gamma) = \alpha \cdot \beta + \alpha \cdot \gamma$
(8) $(\alpha + \beta) \cdot \gamma = \alpha \cdot \gamma + \beta \cdot \gamma$
(9) $0 \cdot \alpha = \alpha \cdot 0 = 0$
(10) $1 + \alpha \cdot \alpha^* \leq \alpha^*$
(11) $1 + \alpha^* \cdot \alpha \leq \alpha^*$
(12) $\alpha + \beta \cdot \gamma \leq \gamma \quad \rightarrow \quad \beta^* \cdot \alpha \leq \gamma$
(13) $\alpha + \gamma \cdot \beta \leq \gamma \quad \rightarrow \quad \alpha \cdot \beta^* \leq \gamma$

A *synchronous Kleene algebra* (SKA) over a finite set $\mathsf{A_B}$ is a structure $(\mathcal{A}, +, \cdot, \times, ^*, 0, 1, \mathsf{A_B})$, where $\mathsf{A_B} \subseteq \mathcal{A}$, $(\mathcal{A}, +, \cdot, ^*, 0, 1)$ is a Kleene algebra, and $\times$ is a binary operator that satisfies axioms (S1)-(S8) below. The set $\mathsf{A_B^\times}$ is the smallest subset of $\mathcal{A}$ that contains $\mathsf{A_B}$ and is closed for $\times$.

(S1) $\alpha \times (\beta \times \gamma) = (\alpha \times \beta) \times \gamma$
(S2) $\alpha \times \beta = \beta \times \alpha$
(S3) $\alpha \times 0 = 0 \times \alpha = 0$
(S4) $\alpha \times 1 = 1 \times \alpha = \alpha$
(S5) $a \times a = a \quad \forall a \in \mathsf{A_B}$
(S6) $\alpha \times (\beta + \gamma) = \alpha \times \beta + \alpha \times \gamma$
(S7) $(\alpha + \beta) \times \gamma = \alpha \times \gamma + \beta \times \gamma$
(S8) $(\alpha^\times \cdot \alpha) \times (\beta^\times \cdot \beta) = (\alpha^\times \times \beta^\times) \cdot (\alpha \times \beta) \quad \forall \alpha^\times, \beta^\times \in \mathsf{A_B^\times}$

As usual, we will omit the operator $\cdot$ whenever it does not give rise to any ambiguity and use the following precedence over the operators: $+ < \cdot < \times < ^*$.

Synchronous Kleene algebra was first presented by C. Prisacariu [20]. We think of the elements of SKA as processes taking place within a fixed discrete time frame and that, at each time step, may execute one or more basic actions in $\mathsf{A_B}$ or then come to a halt. Apart from the usual KA operators $+$, $\cdot$ and *, corresponding to choice, sequential execution and iteration, we also have an operator $\times$ that corresponds to a simultaneous execution where the actions executed in each time step are synchronized. In this context, the elements of $\mathsf{A_B^\times}$, called the synchronous actions, are the possible behaviours for one single time step. Algebraically, this synchronous behaviour is imposed by axiom (S8), which is therefore called the synchrony axiom.

The standard model of an SKA over $\mathsf{A_B}$ is the set of languages over the alphabet $\Sigma = \mathcal{P}(\mathsf{A_B}) \setminus \{\emptyset\}$, which we will call synchronous languages. Note that, modulo the natural identifications, $\mathsf{A_B} \subseteq \Sigma \subseteq \Sigma^* \subseteq \mathcal{P}(\Sigma^*)$. Each synchronous language represents a process described by its possible executions,

which are given by the words over Σ, each one a sequence of sets of basic actions executed in a single time step. As usual, the operators $+$, $\cdot$ and * denote the union, concatenation and Kleene star of languages. It is well known that these operations turn the set of languages into a Kleene algebra. The synchronous product of two words $x = \sigma_1 \cdots \sigma_m$ and $y = \tau_1 \cdots \tau_n$, with $n \geq m$, is defined by

$$x \times y = y \times x = (\sigma_1 \cup \tau_1 \cdots \sigma_m \cup \tau_m)\tau_{m+1} \cdots \tau_n.$$

In particular, the synchronous product of two letters in Σ is their union. The synchronous product of two languages L_1 and L_2 is defined by

$$L_1 \times L_2 = \{x \times y \,|\, x \in L_1, y \in L_2\}.$$

With this definition, the set of synchronous actions $\mathsf{A}_\mathsf{B}^\times$ is precisely Σ, and $\times$ verifies axioms (S1)-(S8). Having established that $\mathcal{P}(\Sigma^*)$ is an SKA over A_B, we can focus on the smallest subalgebra that contains A_B, which corresponds to the set of languages finitely generated from A_B, 0 and 1 and operators $+, \cdot, \times, ^*$, called the synchronous regular languages over A_B. It is clear that the synchronous regular languages over A_B contain the regular languages over Σ. As it will turn out later, when we construct an automaton accepting each synchronous regular language, they are exactly the same set: the regular languages over Σ are also closed for $\times$. In [20], the classical Thompson construction for regular languages is extended to build an automaton accepting the synchronous product of two languages given by their automata.

We now introduce the SKA analogue of the regular expressions. We denote by $\mathcal{T}_\mathsf{SKA}$ the set containing 0 plus all terms finitely generated from $\mathsf{A}_\mathsf{B} \cup \{1\}$ and operators $+, \cdot, \times, ^*$, that is, the terms generated by the grammar

$$\alpha \quad \to \quad 1 \mid a \mid \alpha + \alpha \mid \alpha \cdot \alpha \mid \alpha \times \alpha \mid \alpha^* \quad (a \in \mathsf{A}_\mathsf{B}).$$

Note that we do not include in $\mathcal{T}_\mathsf{SKA}$ compound expressions that have 0 as a subexpression. We do so because it will simplify matters later on and, since 0 is the identity for $+$ and an absorbing element for $\cdot$ and $\times$, it does not restrict the intended semantics. The elements of $\mathcal{T}_\mathsf{SKA}$, called SKA terms or SKA expressions, are a representation of the synchronous regular languages over A_B in the natural way. More precisely, given $\alpha \in \mathcal{T}_\mathsf{SKA}$, the language $\mathcal{L}(\alpha)$ denoted by α is inductively defined as follows,

$$\begin{array}{lcl} \mathcal{L}(a) & = & \{\{a\}\} \\ \mathcal{L}(0) & = & \emptyset \\ \mathcal{L}(1) & = & \{\varepsilon\} \\ \mathcal{L}(\alpha^*) & = & \mathcal{L}(\alpha)^* \end{array} \qquad \begin{array}{lcl} \mathcal{L}(\alpha + \beta) & = & \mathcal{L}(\alpha) \cup \mathcal{L}(\beta) \\ \mathcal{L}(\alpha\beta) & = & \mathcal{L}(\alpha)\mathcal{L}(\beta) \\ \mathcal{L}(\alpha \times \beta) & = & \mathcal{L}(\alpha) \times \mathcal{L}(\beta). \end{array}$$

Example 1. *Let* $\mathsf{A}_\mathsf{B} = \{a, b\}$*, hence* $\Sigma = \{\{a\}, \{b\}, \{a, b\}\}$*, and consider the SKA term* $\alpha = (a(b + a)^*) \times (a + bb)^*$ *over* A_B*. Then*

$$\begin{aligned} \mathcal{L}(\alpha) &= \{\{a\}, \{a\}\{a\}, \{a\}\{b\}, \ldots\} \times \{\varepsilon, \{a\}, \{a\}\{a\}, \{b\}\{b\}, \ldots\} \\ &= \{\{a\}, \{a\}\{a\}, \{a\}\{b\}, \{a\}\{a, b\}, \{a, b\}\{b\}, \{a, b\}\{a, b\}, \ldots\}. \end{aligned}$$

Given $\alpha, \beta \in \mathcal{T}_\mathsf{SKA}$, we write $\alpha = \beta$ if α and β are syntactically equal and $\alpha \sim \beta$ if they are *equivalent*, i.e., denote the same language. We also define $\varepsilon(\alpha) = \varepsilon(\mathcal{L}(\alpha))$, where, given a language L, $\varepsilon(L) = 1$ if $\varepsilon \in L$ and $\varepsilon(L) = 0$ otherwise. A recursive definition of $\varepsilon : \mathcal{T}_\mathsf{SKA} \longrightarrow \{0, 1\}$ is given by the following,

$$\begin{array}{lcl} \varepsilon(a) = \varepsilon(0) & = & 0 \\ \varepsilon(1) & = & 1 \\ \varepsilon(\alpha^*) & = & 1 \end{array} \qquad \begin{array}{lcl} \varepsilon(\alpha + \beta) & = & \varepsilon(\alpha) + \varepsilon(\beta) \\ \varepsilon(\alpha\beta) & = & \varepsilon(\alpha)\varepsilon(\beta) \\ \varepsilon(\alpha \times \beta) & = & \varepsilon(\alpha)\varepsilon(\beta). \end{array}$$

We generalize ε for sets $S \subseteq \mathcal{T}_\mathsf{SKA}$ by $\varepsilon(S) = \sum_{\alpha \in S} \varepsilon(\alpha)$.

2.1 Automata and Systems of Equations

We first recall the definition of a *nondeterministic finite automaton* (NFA). An NFA is a tuple $\mathcal{A} = \langle S, \Sigma, S_0, \delta, F\rangle$, where S is a finite set of states, Σ is a finite alphabet, $S_0 \subseteq S$ a set of initial states, $\delta : S \times \Sigma \longrightarrow \mathcal{P}(S)$ the transition function, and $F \subseteq S$ a set of final states. The extension $\hat{\delta} : \mathcal{P}(S) \times \Sigma^* \longrightarrow \mathcal{P}(S)$ of δ to sets of states and words is defined by $\hat{\delta}(X, \varepsilon) = X$ and $\hat{\delta}(X, \sigma x) = \hat{\delta}(\cup_{s\in X}\delta(s, \sigma), x)$. A word $x \in \Sigma^*$ is accepted by $\mathcal{A}$ if and only if $\hat{\delta}(S_0, x) \cap F \neq \emptyset$. The *language of* $\mathcal{A}$ is the set of words accepted by $\mathcal{A}$ and denoted by $\mathcal{L}(\mathcal{A})$. The *right language of a state* s, denoted by $\mathcal{L}_s$, is the language accepted by $\mathcal{A}$ if we take $S_0 = \{s\}$. The class of languages accepted by an NFA is precisely the set of regular languages. It is well known that there is a natural correspondence between each NFA over $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$ with n states and right languages $\mathcal{L}_1, \ldots, \mathcal{L}_n$ and each system of linear equations

$$\mathcal{L}_i \quad = \quad \sigma_1 a_{1i} + \cdots + \sigma_k a_{ki} + \varepsilon(\mathcal{L}_i), \quad i \in [1, n]$$

where each a_{ij} is a (possibly empty) sum of elements in $\{\mathcal{L}_1, \ldots, \mathcal{L}_n\}$.

In the context of SKA, we consider the alphabet $\Sigma = \mathcal{P}(\mathsf{A_B}) \setminus \{\emptyset\}$ and call the NFA a *nondeterministic automaton on synchronous strings*. Each of these automata accepts a synchronous language. We will see that for each synchronous regular language over $\mathsf{A_B}$ (represented by an SKA expression) there is an automaton that accepts it, so that the set of synchronous regular languages over $\mathsf{A_B}$ is precisely the set of regular languages over Σ. We now generalize the notion of support of a regular expression to SKA terms.

Definition 2. *Consider* $\Sigma = \mathcal{P}(\mathsf{A_B}) \setminus \{\emptyset\} = \{\sigma_1, \ldots, \sigma_k\}$ *and* $\alpha_0 \in \mathcal{T}_{\mathsf{SKA}}$. *A* support *of* α_0 *is a set* $\{\alpha_1, \ldots, \alpha_n\}$ *that satisfies a system of equations*

$$\alpha_i \quad \sim \quad \sigma_1\alpha_{1i} + \cdots + \sigma_k\alpha_{ki} + \varepsilon(\alpha_i), \quad i \in [0, n] \tag{1}$$

for some $\alpha_{i1}, \ldots, \alpha_{ik}$, *each one a (possibly empty) sum of elements in* $\{\alpha_1, \ldots, \alpha_n\}$. *In this case* $\{\alpha_0, \alpha_1, \ldots, \alpha_n\}$ *is called a* prebase *of* α_0.

Example 3. *Consider again the expression* α *from Example 1 and let* $\beta = (b+a)^*$ *and* $\gamma = (a+bb)^*$, *i.e.* $\alpha = (a\beta)\times\gamma$. *The set* $\{\beta\times\gamma, \beta\times(b\gamma), \beta, b\gamma, \gamma\}$ *is a support for* $\alpha_0 = \alpha$, *as shown by the following system of equations.*

$$\begin{array}{lcllllll}
(a\beta)\times\gamma & = & \alpha_0 & \sim & \{a\}(\alpha_1+\alpha_3) & & +\{a,b\}\alpha_2 & \\
\beta\times\gamma & = & \alpha_1 & \sim & \{a\}(\alpha_1+\alpha_3+\alpha_5) & +\{b\}(\alpha_2+\alpha_3+\alpha_4) & +\{a,b\}(\alpha_1+\alpha_2) & +1 \\
\beta\times(b\gamma) & = & \alpha_2 & \sim & & +\{b\}(\alpha_1+\alpha_5) & +\{a,b\}\alpha_1 & \\
\beta & = & \alpha_3 & \sim & \{a\}\alpha_3 & +\{b\}\alpha_3 & & +1 \\
b\gamma & = & \alpha_4 & \sim & & +\{b\}\alpha_5 & & \\
\gamma & = & \alpha_5 & \sim & \{a\}\alpha_5 & +\{b\}\alpha_4 & & +1
\end{array}$$

It is clear from what was said above that the existence of a support of α implies the existence of an NFA that accepts the language denoted by α. If $\{\alpha_1, \ldots, \alpha_n\}$ is a support of $\alpha = \alpha_0$, the system of equations (1) can be written in matrix form $\mathsf{A}_\alpha \sim \mathsf{C}\cdot\mathsf{M}_\alpha + \mathsf{E}_\alpha$, where M_α is the $k \times (n+1)$ matrix with entries α_{ij}, and A_α, C and E_α denote respectively the following three matrices,

$$\mathsf{A}_\alpha = \begin{bmatrix}\alpha_0 & \cdots & \alpha_n\end{bmatrix}, \quad \mathsf{C} = \begin{bmatrix}\sigma_1 & \cdots & \sigma_k\end{bmatrix}, \quad \text{and} \quad \mathsf{E}_\alpha = \begin{bmatrix}\varepsilon(\alpha_0) & \cdots & \varepsilon(\alpha_n)\end{bmatrix}.$$

In the above, $\mathsf{C}\cdot\mathsf{M}_\alpha$ denotes the matrix obtained from C and M_α applying the standard rules of matrix multiplication, but replacing the multiplication operator with concatenation. We now define a function $\pi : \mathcal{T}_{\mathsf{SKA}} \longrightarrow \mathcal{P}(\mathcal{T}_{\mathsf{SKA}})$ that recursively computes a support for an expression $\alpha \in \mathcal{T}_{\mathsf{SKA}}$. The proof of the correctness of this definition provides us simultaneously with the definition of a corresponding equation system as in (1).

We now define a certain form of concatenation and synchronous product between two sets of SKA expressions. The idea is that we make the formal product between each element of one set and each of the other, but with two exceptions: products by 0 are not considered, and elements multiplied by

1 remain the same. More precisely, given $\alpha, \beta \neq 0$ in $\mathcal{T}_{\mathsf{SKA}}$, define $\alpha \odot \beta = \alpha \cdot \beta$ if $\alpha, \beta \neq 1$ and $\alpha \odot 1 = 1 \odot \alpha = \alpha$. Similarly, define $\alpha \otimes \beta = \alpha \times \beta$ if $\alpha, \beta \neq 1$ and $\alpha \otimes 1 = 1 \otimes \alpha = \alpha$. Given sets $S, T \subseteq \mathcal{T}_{\mathsf{SKA}}$, let $S \odot T = \{\alpha \odot \beta \mid \alpha \in S \backslash \{0\}, \beta \in T \backslash \{0\}\}$ and $S \otimes T = \{\alpha \otimes \beta \mid \alpha \in S \backslash \{0\}, \beta \in T \backslash \{0\}\}$. Convention that $\alpha \odot S = \{\alpha\} \odot S$, and similarly for $S \odot \alpha$, $\alpha \otimes S$ and $S \otimes \alpha$. These definitions somehow embody the axiomatic role of 0 and 1, and serve the following definition.

Definition 4. *Given $\alpha \in \mathcal{T}_{\mathsf{SKA}}$, the set $\pi(\alpha)$ is inductively defined by,*

$$\begin{array}{rclcrcl} \pi(0) & = & \pi(1) = \emptyset & & \pi(\alpha + \beta) & = & \pi(\alpha) \cup \pi(\beta) \\ \pi(a) & = & \{1\} \quad (a \in \mathsf{A_B}) & & \pi(\alpha\beta) & = & \pi(\alpha) \odot \beta \cup \pi(\beta) \\ \pi(\alpha^*) & = & \pi(\alpha) \odot \alpha^* & & \pi(\alpha \times \beta) & = & \pi(\alpha) \otimes \pi(\beta) \cup \pi(\alpha) \cup \pi(\beta). \end{array}$$

It is clear from the definition that $\pi(\alpha)$ is finite.

Proposition 5. *If $\alpha \in \mathcal{T}_{\mathsf{SKA}}$, then the set $\pi(\alpha)$ is a support of α.*

Proof. We proceed by induction on the structure of α. Excluding the case where α is $\alpha_0 \times \beta_0$, the proof can be found in [17, 10]. We now describe how to obtain a system of equations corresponding to an expression $\alpha_0 \times \beta_0$ from systems for α_0 and β_0. Suppose that $\pi(\alpha_0) = \{\alpha_1, \ldots, \alpha_n\}$ is a support of α_0 and $\pi(\beta_0) = \{\beta_1, \ldots, \beta_m\}$ is a support of β_0. For α_0 and β_0 consider C, A_{α_0}, M_{α_0}, E_{α_0} and A_{β_0}, M_{β_0}, E_{β_0} as above. We wish to show that

$$\pi(\alpha_0 \times \beta_0) = \{\alpha_1 \otimes \beta_1, \ldots, \alpha_1 \otimes \beta_m, \ldots, \alpha_n \otimes \beta_1, \ldots, \alpha_n \otimes \beta_m\} \cup \pi(\alpha_0) \cup \pi(\beta_0)$$

is a support of $\alpha_0 \times \beta_0$. Define the $(n+1)(m+1)$-column matrices

$$\begin{array}{rcl} \mathsf{A}_{\alpha_0 \times \beta_0} & = & \begin{bmatrix} \alpha_0 \otimes \beta_0 & \alpha_1 \otimes \beta_1 & \cdots & \alpha_n \otimes \beta_m & \alpha_1 & \cdots & \alpha_n & \beta_1 & \cdots & \beta_m \end{bmatrix} \\ \mathsf{E}_{\alpha_0 \times \beta_0} & = & \begin{bmatrix} \varepsilon(\alpha_0 \otimes \beta_0) & \varepsilon(\alpha_1 \otimes \beta_1) & \cdots & \varepsilon(\alpha_n \otimes \beta_m) & \varepsilon(\alpha_1) & \cdots & \varepsilon(\alpha_n) & \varepsilon(\beta_1) & \cdots & \varepsilon(\beta_n) \end{bmatrix} \end{array}$$

and let $\mathsf{M}_{\alpha_0 \times \beta_0}$ be the $k \times (n+1)(m+1)$ matrix whose last $n+m$ columns are the n columns of M_{α_0} followed by the m columns of M_{β_0}, and whose remaining entries $\gamma_{l,(i,j)}$, for $l \in [1,k]$ and $(i,j) \in \{(0,0)\} \cup [1,n] \times [1,m]$, are defined by:

$$\gamma_{l,(i,j)} = \sum \left(\{\alpha_{l_1 i} \otimes \beta_{l_2 j} \mid \sigma_{l_1} \times \sigma_{l_2} = \sigma_l\} \cup \{\varepsilon(\alpha_i)\} \otimes \{\beta_{lj}\} \cup \{\varepsilon(\beta_j)\} \otimes \{\alpha_{li}\} \right)$$

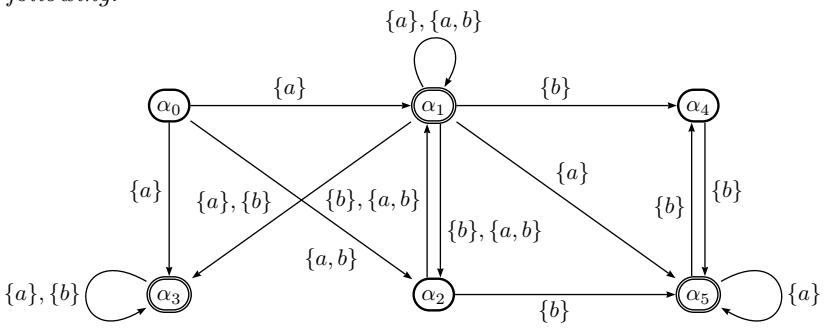
Note that, since by induction hypothesis each α_{li} is a sum of elements in $\pi(\alpha)$ and each β_{lj} is a sum of elements in $\pi(\beta)$, each element of $\mathsf{M}_{\alpha_0 \times \beta_0}$ is in fact a sum of elements in $\pi(\alpha_0 \times \beta_0)$. We will show that $\mathsf{A}_{\alpha_0 \times \beta_0} \sim \mathsf{C} \cdot \mathsf{M}_{\alpha_0 \times \beta_0} + \mathsf{E}_{\alpha_0 \times \beta_0}$. For the last $n+m$ entries of $\mathsf{A}_{\alpha_0 \times \beta_0}$, this is obvious. Consider now $\alpha_i \times \beta_j$ for some $(i,j) \in \{(0,0)\} \cup [1,n] \times [1,m]$. We have $\alpha_i \sim \sigma_1 \alpha_{1i} + \cdots + \sigma_k \alpha_{ki} + \varepsilon(\alpha_i)$ and $\beta_j \sim \sigma_1 \beta_{1j} + \cdots + \sigma_k \beta_{kj} + \varepsilon(\beta_j)$, thus, using properties of $\times$, namely the synchrony axiom,

$$\begin{array}{rcl} \alpha_i \times \beta_j & \sim & (\sigma_1 \alpha_{1i} + \cdots + \sigma_k \alpha_{ki} + \varepsilon(\alpha_i)) \times (\sigma_1 \beta_{1j} + \cdots + \sigma_k \beta_{kj} + \varepsilon(\beta_j)) \\ & \sim & \displaystyle\sum_{1 \leq l_1, l_2 \leq k} (\sigma_{l_1} \times \sigma_{l_2}).(\alpha_{l_1 i} \times \beta_{l_2 j}) \;+\; \varepsilon(\alpha_i) \times \sum_{1 \leq l \leq k} \sigma_l \beta_{jl} \;+ \\ & & \varepsilon(\beta_j) \times \displaystyle\sum_{1 \leq l \leq k} \sigma_l \alpha_{li} + \varepsilon(\alpha_i) \times \varepsilon(\beta_j) \\ & \sim & \sigma_1 \left(\displaystyle\sum_{\sigma_{l_1} \times \sigma_{l_2} = \sigma_1} \alpha_{l_1 i} \times \beta_{l_2 j} + \varepsilon(\alpha_i) \times \beta_{j1} + \varepsilon(\beta_j) \times \alpha_{1i} \right) \;+\; \cdots \;+ \\ & & \sigma_k \left(\displaystyle\sum_{\sigma_{l_1} \times \sigma_{l_2} = \sigma_k} \alpha_{l_1 i} \times \beta_{l_2 j} + \varepsilon(\alpha_i) \times \beta_{kj} + \varepsilon(\beta_j) \times \alpha_{ki} \right) + \varepsilon(\alpha_i) \times \varepsilon(\beta_j) \\ & \sim & \sigma_1 \gamma_{1,(i,j)} + \cdots + \sigma_k \gamma_{k,(i,j)} + \varepsilon(\alpha_i \otimes \beta_j). \end{array}$$

□

This shows that the language associated with each SKA expression is regular, and gives a way to construct a system of equations corresponding to an NFA that accepts it. This is done by recursively computing $\pi(\alpha)$ and the matrices A_α and E_α, obtaining the whole NFA in the final step.

Example 6. *For $\alpha = (a\beta) \times \gamma$ as before, we have $\pi(a\beta) = \{\beta\}$, $\pi(\gamma) = \{\gamma, b\gamma\}$, and $\pi(\alpha) = \pi(a\beta) \times \pi(\gamma) \cup \pi(a\beta) \cup \pi(\gamma) = \{\beta, \gamma, b\gamma, \beta \times \gamma, \beta \times (b\gamma)\}$. Thus, $\pi(\alpha)$ is exactly the support set given in Example 3. Furthermore, the automaton obtained from the system of equations in that example is the following.*



We note that the automata computed by this method can have non-useful states, i.e. states that are not reached from the initial states. Thus, frequently it is more useful being able to compute an accepting NFA on-the-fly, i.e., by computing the transitions from already computed states as they are necessary. In the next section we will show how to build an accepting NFA for each SKA expression using the theory of partial derivatives.

2.2 Partial Derivatives

As usual, the *left-quotient* of a synchronous language L w.r.t. a concurrent action σ is the set $\sigma^{-1}L = \{\, x \mid \sigma x \in L \,\}$. The left quotient of L w.r.t. a word $x \in \Sigma^*$ is inductively defined by $\varepsilon^{-1}L = L$ and $(x\sigma)^{-1}L = \sigma^{-1}(x^{-1}L)$. The left quotient w.r.t. a letter $\sigma \in \Sigma$ has the following properties:

$$\sigma^{-1}\emptyset = \sigma^{-1}\{\varepsilon\} = \emptyset \qquad \sigma^{-1}\{\{a\}\} = \begin{cases} \{\varepsilon\} \text{ if } \sigma = \{a\} \\ \emptyset \text{ otherwise} \end{cases}$$

$$\begin{aligned} \sigma^{-1}(L^*) &= (\sigma^{-1}L)L^* \\ \sigma^{-1}(L_1 + L_2) &= \sigma^{-1}L_1 \cup \sigma^{-1}L_2 \\ \sigma^{-1}(L_1L_2) &= (\sigma^{-1}L_1)L_2 \cup \varepsilon(L_1)\sigma^{-1}L_2 \end{aligned}$$

$$\sigma^{-1}(L_1 \times L_2) = \big(\textstyle\bigcup_{\sigma_1 \times \sigma_2 = \sigma} \sigma_1^{-1}L_1 \times \sigma_2^{-1}L_2\big) \cup \varepsilon(L_1)\sigma^{-1}L_2 \cup \varepsilon(L_2)\sigma^{-1}L_1.$$

With exception of the property concerning $\times$, which follows from the definition of synchronous product of two languages, the above properties are well known [9]. Antimirov [3] introduced the notion of partial derivatives which we now generalize to the set $\mathcal{T}_{\mathsf{SKA}}$.

Definition 7. *The set of partial derivatives (s.p.d.) of a term $\alpha \in \mathcal{T}_{SKA}$ w.r.t. the letter $\sigma \in \Sigma$, denoted by $\partial_\sigma(\alpha)$, is inductively defined by*

$$\partial_\sigma(0) = \partial_\sigma(1) = \emptyset \qquad \partial_\sigma(a) = \begin{cases} \{1\} \textit{ if } \sigma = \{a\} \\ \emptyset \textit{ otherwise} \end{cases}$$

$$\begin{aligned} \partial_\sigma(\alpha^*) &= \partial_\sigma(\alpha) \odot \alpha^* \\ \partial_\sigma(\alpha + \beta) &= \partial_\sigma(\alpha) \cup \partial_\sigma(\beta) \\ \partial_\sigma(\alpha\beta) &= \partial_\sigma(\alpha) \odot \beta \;\cup\; \varepsilon(\alpha) \odot \partial_\sigma(\beta) \end{aligned}$$

$$\partial_\sigma(\alpha \times \beta) = \big(\textstyle\bigcup_{\sigma_1 \times \sigma_2 = \sigma} \partial_{\sigma_1}(\alpha) \otimes \partial_{\sigma_2}(\beta)\big) \;\cup\; \varepsilon(\alpha) \otimes \partial_\sigma(\beta) \;\cup\; \varepsilon(\beta) \otimes \partial_\sigma(\alpha).$$

The s.p.d. of $\alpha \in \mathcal{T}_{SKA}$ w.r.t. a word $x \in \Sigma^*$ is inductively defined by $\partial_\varepsilon(\alpha) = \{\alpha\}$ and $\partial_{x\sigma}(\alpha) = \partial_\sigma(\partial_x(\alpha))$, where, given a set $S \subseteq \mathcal{T}_{SKA}$, $\partial_\sigma(S) = \bigcup_{\alpha \in S} \partial_\sigma(\alpha)$.

We denote by $\partial(\alpha)$ the set of all partial derivatives of α, $\partial(\alpha) = \bigcup_{x \in \Sigma^*} \partial_x(\alpha)$, and by $\partial^+(\alpha)$ the set of partial derivatives excluding the trivial derivative by ε, $\partial^+(\alpha) = \bigcup_{x \in \Sigma^+} \partial_x(\alpha)$. Given a set $S \subseteq \mathcal{T}_{\mathsf{SKA}}$, we define $\mathcal{L}(S) = \bigcup_{\alpha \in S} \mathcal{L}(\alpha)$.

Proposition 8. *For every SKA term α and word x, $\mathcal{L}(\partial_x(\alpha)) = x^{-1}\mathcal{L}(\alpha)$.*

Proof. Due to the correspondence between each case of the definition of the set of partial derivatives and the properties of the left quotient, one can check by induction on the structure of α that, for every $\sigma \in \Sigma$, $\mathcal{L}(\partial_\sigma(\alpha)) = \sigma^{-1}\mathcal{L}(\alpha)$. We now prove the result by induction on the length of the word x: we have $\mathcal{L}(\partial_\varepsilon(\alpha)) = \mathcal{L}(\{\alpha\}) = \varepsilon^{-1}L(\alpha)$ and, supposing the claim to be true for x, $L(\partial_{x\sigma}(\alpha)) = L(\partial_\sigma(\partial_x(\alpha))) = \sigma^{-1}L(\partial_x(\alpha)) = \sigma^{-1}(x^{-1}L(\alpha)) = (x\sigma)^{-1}L(\alpha)$. □

The following proposition shows that $\partial(\alpha)$ is finite.

Proposition 9. *Given $\alpha \in \mathcal{T}_{\mathsf{SKA}}$, $\partial^+(\alpha) \subseteq \pi(\alpha)$.*

Proof. The proof proceeds by induction on the structure of α. It is clear that $\partial^+(0) = \pi(0)$, $\partial^+(1) = \pi(1)$ and, for $a \in \mathsf{A_B}$, $\partial^+(a) = \pi(a)$. Now, suppose the claim is true for α and β. There are four induction cases to consider, in which we will make use of the fact that, for any SKA expression γ and letter $\sigma \in \Sigma$, the set $\partial^+(\gamma)$ is closed for taking derivatives w.r.t. σ, i.e., $\partial_\sigma(\partial^+(\gamma)) \subseteq \partial^+(\gamma)$.

i. One can check by induction on the length of x that, for $x \in \Sigma^+$, $\partial_x(\alpha + \beta) = \partial_x(\alpha) \cup \partial_x(\beta)$. Then $\partial^+(\alpha + \beta) = \partial^+(\alpha) \cup \partial^+(\beta) \subseteq \pi(\alpha) \cup \pi(\beta) = \pi(\alpha + \beta)$.

ii. We will prove by induction on the length of x that $\partial_x(\alpha\beta) \subseteq \partial^+(\alpha) \odot \beta \cup \partial^+(\beta)$ for every word $x \in \Sigma^+$. The claim is true for $\sigma \in \Sigma$ since $\partial_\sigma(\alpha\beta) = \partial_\sigma(\alpha) \odot \beta \cup \varepsilon(\alpha) \odot \partial_\sigma(\beta)$. Assuming it is true for x, $\partial_{x\sigma}(\alpha\beta) = \partial_\sigma(\partial_x(\alpha\beta)) \subseteq \partial_\sigma(\partial^+(\alpha) \odot \beta \cup \partial^+(\beta)) \subseteq \partial_\sigma(\partial^+(\alpha)) \odot \beta \cup \partial_\sigma(\beta) \cup \partial_\sigma(\partial^+(\beta)) \subseteq \partial^+(\alpha) \odot \beta \cup \partial^+(\beta)$. Hence $\partial^+(\alpha\beta) \subseteq \partial^+(\alpha) \odot \beta \cup \partial^+(\beta) \subseteq \pi(\alpha) \odot \beta \cup \pi(\beta) = \pi(\alpha\beta)$.

iii. We prove by induction on the length of x that, for every word $x \in \Sigma^+$, $\partial_x(\alpha \times \beta) \subseteq \partial^+(\alpha) \otimes \partial^+(\beta) \cup \partial^+(\alpha) \cup \partial^+(\beta)$. The claim is true for $\sigma \in \Sigma$ because $\partial_\sigma(\alpha \times \beta) = \bigcup_{\sigma_1 \times \sigma_2 = \sigma} \partial_{\sigma_1}(\alpha) \otimes \partial_{\sigma_2}(\beta) \cup \varepsilon(\alpha) \otimes \partial_\sigma(\beta) \cup \varepsilon(\beta) \otimes \partial_\sigma(\alpha)$; supposing it is true for x, $\partial_{x\sigma}(\alpha \times \beta) = \partial_\sigma(\partial_x(\alpha \times \beta)) \subseteq \partial_\sigma(\partial^+(\alpha) \otimes \partial^+(\beta) \cup \partial^+(\alpha) \cup \partial^+(\beta)) \subseteq (\bigcup_{\sigma_1 \times \sigma_2 = \sigma} \partial_{\sigma_1}(\partial^+(\alpha)) \otimes \partial_{\sigma_2}(\partial^+(\beta))) \cup \partial_\sigma(\partial^+(\alpha)) \cup \partial_\sigma(\partial^+(\beta)) \subseteq \partial^+(\alpha) \otimes \partial^+(\beta) \cup \partial^+(\alpha) \cup \partial^+(\beta)$. Hence $\partial^+(\alpha \times \beta) \subseteq \partial^+(\alpha) \otimes \partial^+(\beta) \cup \partial^+(\alpha) \cup \partial^+(\beta) \subseteq \pi(\alpha) \otimes \pi(\beta) \cup \pi(\alpha) \cup \pi(\beta) = \pi(\alpha \times \beta)$.

iv. We show by induction on the length of x that $\partial_x(\alpha^*) \subseteq \partial^+(\alpha) \odot \alpha^*$ for $x \in \Sigma^+$. It is true for $\sigma \in \Sigma$ because $\partial_\sigma(\alpha^*) = \partial_\sigma(\alpha) \odot \alpha^*$; supposing the claim true for x, $\partial_{\sigma x}(\alpha^*) = \partial_\sigma(\partial_x(\alpha^*)) \subseteq \partial_\sigma(\partial^+(\alpha) \odot \alpha^*) \subseteq \partial_\sigma(\partial^+(\alpha)) \odot \alpha^* \cup \partial_\sigma(\alpha^*) \subseteq \partial^+(\alpha) \odot \alpha^* \cup \partial_\sigma(\alpha) \odot \alpha^* \subseteq \partial^+(\alpha) \odot \alpha^*$. Hence $\partial^+(\alpha^*) \subseteq \partial^+(\alpha) \odot \alpha^* \subseteq \pi(\alpha) \odot \alpha^* = \pi(\alpha^*)$.

□

Remark 10. *When we do not consider $\times$, in the context of KA, it is in fact true that $\partial^+(\alpha) = \pi(\alpha)$. However, with the present extended definitions that equality is not true. For instance, $\pi(aa \times ab) = \{a \times b, b, a, 1\}$, but $\partial^+(aa \times ab) = \{a \times b, 1\}$. This happens because $\pi(\alpha \times \beta)$ includes all the non trivial derivatives of α and β, and these are not necessarily derivatives of $\alpha \times \beta$, since when deriving $\alpha \times \beta$ by a letter we "advance" in both α and β at the same time (unlike what happens with concatenation, there is no way to derive only one of the factors while keeping the other one fixed).*

As it is well known, the set of partial derivatives of a regular expression gives rise to an equivalent NFA, called the Antimirov automaton or partial derivative automaton. This remains valid in our extension of the partial derivatives to SKA terms. Given $\alpha \in \mathcal{T}_{\mathsf{SKA}}$, we define the partial derivative automaton associated to α by

$$\mathcal{A}(\alpha) = \langle \partial(\alpha), \Sigma, \{\alpha\}, \delta_\alpha, F_\alpha \rangle,$$

where $F_\alpha = \{\, \gamma \in \partial(\alpha) \mid \varepsilon(\gamma) = 1 \,\}$ and $\delta_\alpha(\gamma, \sigma) = \partial_\sigma(\gamma)$.

Proposition 11. *For every state $\gamma \in \partial(\alpha)$, the right language $\mathcal{L}_\gamma$ of γ in $\mathcal{A}(\alpha)$ is equal to $\mathcal{L}(\gamma)$, the language denoted by γ. In particular, the language accepted by $\mathcal{A}(\alpha)$ is precisely $\mathcal{L}(\alpha)$.*

Proof. Let $x \in \Sigma^*$. We will prove by induction on the length of x that x is accepted starting from state γ iff $x \in \mathcal{L}(\gamma)$. If $x = \varepsilon$, x is accepted starting from γ iff γ is an accepting state, i.e. $\varepsilon(\gamma) = 1 \Leftrightarrow \varepsilon \in L(\gamma)$. Suppose now that $x = \sigma y$ and the claim is true for y. Then, $x \in L(\gamma) \Leftrightarrow y \in \sigma^{-1}\mathcal{L}(\gamma) \Leftrightarrow y \in \mathcal{L}(\partial_\sigma(\gamma)) \Leftrightarrow y \in \mathcal{L}(\gamma')$ for some $\gamma' \in \partial_\sigma(\gamma) \Leftrightarrow y$ is accepted starting from state γ' for some $\gamma' \in \partial_\sigma(\gamma) \Leftrightarrow x = \sigma y$ is accepted starting from state γ, □

Example 12. *For $\alpha = \alpha_0$ from the previous examples, $\partial_{\{a\}}(\alpha_0) = \{\alpha_1, \alpha_3\}$, $\partial_{\{b\}}(\alpha_0) = \emptyset$, $\partial_{\{a,b\}}(\alpha_0) = \{\alpha_2\}$, $\partial_{\{a\}}(\alpha_1) = \{\alpha_1, \alpha_3, \alpha_5\}$, etc. Furthermore, the automaton obtained from these partial derivatives is precisely the one given in Example 6.*

2.3 Equivalence of SKA Expressions

We are interested in an algorithm that decides whether or not two SKA terms represent the same regular language. Since we already know how to construct an NFA that accepts a given SKA term, the problem is tantamount to deciding the language equivalence of two automata, for which there are already numerous solutions. One possible approach is to search for the existence of a bisimulation in the determinized NFAs (DFAs), as presented by Hopcroft and Karp [11]. This algorithm was extended to NFAs by Almeida et al. [1]. A presentation of this algorithm and an improved variant, together with proofs of correctness, can be found in Bonchi and Pous [5]. Below we just present the naive version of the algorithm adapted to the partial derivative automata of two SKA terms. We recall that, given an NFA $\mathcal{A} = \langle S, \Sigma, S_0, \delta, F\rangle$, the subset construction yields an equivalent DFA given by $\mathcal{A}' = \langle S', \Sigma, S_0, \delta', F'\rangle$ where $S' = \mathcal{P}(S)$, $\delta'(X, \sigma) = \bigcup_{s\in X} \delta(s, \sigma)$ and $F' = \{X \in S' \mid X \cap F \neq 0\}$.

Algorithm 1: Naive algorithm for deciding SKA expression equivalence.

```
1  def NAIVE(α, β):
2    R is empty; todo = {({α}, {β})}
3    while todo is not empty, do:
4          extract (X,Y) from todo
5          if (X,Y) ∈ R then skip
6          if ε(X) ≠ ε(Y) then return False
7          for all σ ∈ Σ, insert (⋃_{γ∈X} ∂_σ(γ), ⋃_{γ∈Y} ∂_σ(γ)) in todo
8          insert (X,Y) in R
9    return True
```

3 Deciding Synchronous Kleene Algebra with Tests

Synchronous Kleene algebra with tests (SKAT) was also introduced by C. Prisicariu as a natural extension of the Kleene algebra with tests to the synchronous setting. The SKA axiomatization was extended to SKAT, whose standard models are sets of guarded synchronous strings. For proving completeness the author defined automata over guarded synchronous strings that were based on the ones considered by Kozen for guarded strings [14]. In the synchronous case, automata were built in two layers: one that processed a synchronous string and another to represent the valuations of the boolean tests (called atoms, as defined below). Our main contribution in this section is to define a much simpler notion of automata and to show that the derivative based methods developed in the previous sections for SKA can be extended to SKAT. The automata considered here are standard finite automata where transitions are labeled both with action symbols and boolean tests (instead of atoms). Similar automata for KAT terms were presented by Silva [22] and Broda et al. [7, 8]. In the next section, we revise the notions of SKAT and guarded synchronous strings.

3.1 SKAT and Guarded Synchronous Strings

Formally, a SKAT is a structure $(\mathcal{A}, \mathcal{B}, +, \cdot, \times, ^*, \neg, 0, 1, \mathsf{A_B}, \mathsf{T})$, where $\mathsf{T} \subseteq \mathcal{B} \subseteq \mathcal{A}$ and $\mathsf{A_B}$ and T are disjoint finite sets, $(\mathcal{A}, +, \cdot, \times, ^*, 0, 1, \mathsf{A_B} \cup \mathsf{T})$ is an SKA, $(\mathcal{B}, +, \cdot, \times, \neg, 0, 1)$ is a Boolean algebra where $\cdot$ and $\times$ both correspond to conjunction, and $(\mathcal{B}, +, \cdot, \times, 0, 1)$ is a subalgebra of $(\mathcal{A}, +, \cdot, \times, 0, 1)$. The

axiomatization of SKAT coincides with the one of SKA with an extra axiom that corresponds to the axiom (S8) for the boolean tests $\mathcal{B}$.

Similar to what was done for SKA, we consider the set $\mathcal{B}_{\mathsf{SKAT}}$ of boolean expressions and the set $\mathcal{T}_{\mathsf{SKAT}}$ of SKAT expressions over $\mathsf{A_B} \cup \mathsf{T}$. $\mathcal{B}_{\mathsf{SKAT}}$ is the set of terms finitely generated from $\mathsf{T} \cup \{0,1\}$ and operators $+, \cdot, \times, \neg$, while $\mathcal{T}_{\mathsf{SKAT}}$ denotes the set of terms finitely generated from $\mathsf{A_B} \cup \mathcal{B}_{\mathsf{SKAT}}$ and operators $+, \cdot, \times, ^*$. Elements of $\mathcal{B}_{\mathsf{SKAT}}$ and $\mathcal{T}_{\mathsf{SKAT}}$ will be denoted by $b, b_1, \ldots$ and $\alpha, \beta, \alpha_1, \ldots$, respectively, and are generated by the following grammar

$$\begin{array}{rcll} b & \to & 0 \mid 1 \mid t \mid b + b \mid b \cdot b \mid b \times b \mid \neg b & (t \in \mathsf{T}), \\ \alpha & \to & a \mid b \mid \alpha + \alpha \mid \alpha \cdot \alpha \mid \alpha \times \alpha \mid \alpha^* & (a \in \mathsf{A_B}). \end{array}$$

The set At of *atoms* over $\mathsf{T} = \{t_0, \ldots, t_{l-1}\}$, with $l \geq 1$, is the set of all boolean assignments to all elements of T, i.e. $\mathsf{At} = \{x_0 \cdots x_{l-1} \mid x_i \in \{t_i, \bar{t}_i\},\ t_i \in \mathsf{T}\}$. We denote elements of At by $\mathtt{v}$, $\mathtt{v}_1$, etc. Note that each atom $\mathtt{v} \in \mathsf{At}$ has associated a binary word of l bits $(w_0 \cdots w_{l-1})$ where $w_i = 0$ if $\bar{t}_i \in \mathtt{v}$, and $w_i = 1$ if $t_i \in \mathtt{v}$. The standard model of SKAT consists of the sets of guarded synchronous strings. The set of guarded synchronous strings over $\mathsf{A_B} \cup \mathsf{T}$ is $\mathsf{GSS} = (\mathsf{At} \cdot \Sigma)^* \cdot \mathsf{At}$, where, as before, $\Sigma = \mathcal{P}(\mathsf{A_B}) \setminus \{\emptyset\}$. For $x = \mathtt{v}_0\sigma_1 \cdots \sigma_m\mathtt{v}_m$ and $y = \mathtt{v}'_0\sigma'_1 \cdots \sigma'_n\mathtt{v}'_n \in \mathsf{GSS}$, where $m, n \geq 0$, $\mathtt{v}_i, \mathtt{v}'_j \in \mathsf{At}$ and $\sigma_i, \sigma'_j \in \Sigma$, we define the *fusion product*

$$x \diamond y = \mathtt{v}_0\sigma_1 \cdots \sigma_m\mathtt{v}_m\sigma'_1 \cdots \sigma'_n\mathtt{v}'_n,$$

if $\mathtt{v}_m = \mathtt{v}'_0$, leaving it undefined otherwise. Similarly, for $m \leq n$ the product $x \times y = y \times x$ is defined only if $\mathtt{v}_0 = \mathtt{v}'_0, \ldots, \mathtt{v}_m = \mathtt{v}'_m$ by

$$x \times y = \mathtt{v}_0(\sigma_1 \cup \sigma'_1) \cdots (\sigma_m \cup \sigma'_m)\mathtt{v}_m\sigma'_{m+1} \cdots \sigma'_n\mathtt{v}'_n.$$

For sets $X, Y \subseteq \mathsf{GSS}$, $X \diamond Y = \{\ x \diamond y \mid x \in X,\ y \in Y,\ x \diamond y \text{ exists}\ \}$ and $X \times Y = \{\ x \times y \mid x \in X,\ y \in Y,\ x \times y \text{ exists}\ \}$. Finally, let $X^0 = \mathsf{At}$ and $X^{n+1} = X \diamond X^n$, for $n \geq 0$, and define $X^* = \bigcup_{n \geq 0} X^n$.

Given a SKAT expression α, we define $\mathsf{GSS}(\alpha) \subseteq \mathsf{GSS}$ inductively as follows,

$$\begin{array}{lcllcl} \mathsf{GSS}(a) & = & \{\ \mathtt{v}_1\{a\}\mathtt{v}_2 \mid \mathtt{v}_1, \mathtt{v}_2 \in \mathsf{At}\ \} & \mathsf{GSS}(\alpha \cdot \beta) & = & \mathsf{GSS}(\alpha) \diamond \mathsf{GSS}(\beta) \\ \mathsf{GSS}(b) & = & \{\ \mathtt{v} \mid \mathtt{v} \in \mathsf{At} \wedge \mathtt{v} \leq b\ \} & \mathsf{GSS}(\alpha \times \beta) & = & \mathsf{GSS}(\alpha) \times \mathsf{GSS}(\beta) \\ \mathsf{GSS}(\alpha + \beta) & = & \mathsf{GSS}(\alpha) \cup \mathsf{GSS}(\beta) & \mathsf{GSS}(\alpha^*) & = & \mathsf{GSS}(\alpha)^*, \end{array}$$

where $\mathtt{v} \leq b$ if $\mathtt{v} \to b$ is a propositional tautology. For $T \subseteq \mathcal{T}_{\mathsf{SKAT}}$, let $\mathsf{GSS}(T) = \bigcup_{\alpha \in T} \mathsf{GSS}(\alpha)$. Given two $\mathcal{T}_{\mathsf{SKAT}}$ expressions α and β, we say that they are *equivalent*, and write $\alpha \sim \beta$, if $\mathsf{GSS}(\alpha) = \mathsf{GSS}(\beta)$.

3.2 Automata for Guarded Synchronous Strings

We now introduce a new class of automata for guarded synchronous strings. Besides their simplicity when compared with the two-level automata of Prisicariu, their transitions are labeled with tests instead of atoms, avoiding in this way the inevitable exponential blow-up on the size of the automata induced by the number of valuations of tests.

A *(nondeterministic) automaton with tests* (NTA) over the alphabets Σ and T is a tuple $\mathcal{A} = \langle S, s_0, o, \delta \rangle$, where S is a finite set of states, $s_0 \in S$ is the initial state, $o : S \to \mathcal{B}_{\mathsf{SKAT}}$ is the output function, and $\delta \subseteq \mathcal{P}(S \times (\mathcal{B}_{\mathsf{SKAT}} \times \Sigma) \times S)$ is the transition relation.

A synchronous guarded string $\mathtt{v}_0\sigma_1 \ldots \sigma_n\mathtt{v}_n$, with $n \geq 0$, is accepted by the automaton $\mathcal{A}$ if and only if there is a sequence of states $s_0, s_1, \ldots, s_{n-1} \in S$, where s_0 is the initial state, and, for $i = 0, \ldots, n-1$, one has $\mathtt{v}_i \leq b_i$ for some $(s_i, (b_i, \sigma_{i+1}), s_{i+1}) \in \delta$, and $\mathtt{v}_n \leq o(s_n)$. The set of all guarded strings accepted by $\mathcal{A}$ is denoted by $\mathsf{GSS}(\mathcal{A})$. Formally, given an NTA $\mathcal{A} = \langle S, s_0, o, \delta \rangle$, one can naturally associate to the transition relation δ a function $\delta' : S \times (\mathsf{At} \cdot \Sigma) \longrightarrow \mathcal{P}(S)$, defined by $\delta'(s, \mathtt{v}\sigma) = \{\ s' \mid (s, (b, \sigma), s') \in \delta,\ \mathtt{v} \leq b\ \}$. Moreover, one can define a function $\hat{\delta} : S \times \mathsf{GSS} \longrightarrow \{0,1\}$ over pairs of states and guarded strings as follows

$$\hat{\delta}(s, \mathtt{v}) \ = \ \begin{cases} 1 & \text{if } \mathtt{v} \leq o(s), \\ 0 & \text{otherwise,} \end{cases} \qquad \hat{\delta}(s, \mathtt{v}\sigma x) \ = \ \sum_{s' \in \delta'(s, \mathtt{v}\sigma)} \hat{\delta}(s', x) \ .$$

Given a state s, $\mathsf{GSS}(s) = \{\ x \in \mathsf{GSS} \mid \hat{\delta}(s,x) = 1\ \}$ is the set of synchronous guarded strings accepted by s, and $\mathsf{GSS}(\mathcal{A}) = \mathsf{GSS}(s_0)$. We say that a SKAT expression $\alpha \in \mathcal{T}_{\mathsf{SKAT}}$ is *equivalent* to an automaton $\mathcal{A}$, and write $\alpha = \mathcal{A}$, if $\mathsf{GSS}(\mathcal{A}) = \mathsf{GSS}(\alpha)$.

3.3 Partial Derivatives for SKAT

In the following, we extend the notion of partial derivative, previously defined in [8] for KAT, to SKAT expressions.

Definition 13. *For $\alpha \in \mathcal{T}_{SKAT}$ and $\sigma \in \Sigma$, the set $\partial_\sigma(\alpha)$ of* partial derivatives *of α w.r.t. σ is a subset of $\mathcal{B}_{SKAT} \times \mathcal{T}_{SKAT}$ inductively defined as follows,*

$$\begin{array}{lcl@{\qquad}lcl}
\partial_\sigma(a) & = & \begin{cases} \{(1,1)\} & \text{if } \sigma = \{a\} \\ \emptyset & \text{otherwise} \end{cases} & \partial_\sigma(\alpha+\beta) & = & \partial_\sigma(\alpha) \cup \partial_\sigma(\beta) \\
\partial_\sigma(b) & = & \emptyset & \partial_\sigma(\alpha\beta) & = & \partial_\sigma(\alpha) \odot \beta \cup \mathsf{out}(\alpha) \odot \partial_\sigma(\beta) \\
\partial_\sigma(\alpha^*) & = & \partial_\sigma(\alpha) \odot \alpha^* & & & \\
\multicolumn{6}{c}{\partial_\sigma(\alpha \times \beta) \quad = \quad \big(\bigcup_{\sigma_1 \times \sigma_2 = \sigma} \partial_{\sigma_1}(\alpha) \otimes \partial_{\sigma_2}(\beta)\big) \cup \mathsf{out}(\alpha) \otimes \partial_\sigma(\beta) \cup \mathsf{out}(\beta) \otimes \partial_\sigma(\alpha),}
\end{array}$$

where $\mathsf{out} : \mathcal{T}_{SKAT} \longrightarrow \mathcal{B}_{SKAT}$ is defined by

$$\begin{array}{lcl@{\qquad}lcl}
\mathsf{out}(a) & = & 0 & \mathsf{out}(\alpha+\beta) & = & \mathsf{out}(\alpha) + \mathsf{out}(\beta) \\
\mathsf{out}(b) & = & b & \mathsf{out}(\alpha \cdot \beta) & = & \mathsf{out}(\alpha) \cdot \mathsf{out}(\beta) \\
\mathsf{out}(\alpha^*) & = & 1 & \mathsf{out}(\alpha \times \beta) & = & \mathsf{out}(\alpha) \times \mathsf{out}(\beta),
\end{array}$$

and for $S, T \subseteq \mathcal{B}_{SKAT} \times \mathcal{T}_{SKAT}$, $\alpha \neq 0$ in $\mathcal{T}_{SKAT}$, and $b \neq 0$ in $\mathcal{B}_{SKAT}$, $S \odot \alpha = \{\ (b', \alpha' \odot \alpha) \mid (b', \alpha') \in S,\ \alpha' \neq 0\ \}$, $b \odot S = \{\ (b \odot b', \alpha') \mid (b', \alpha') \in S, b' \neq 0\ \}$, $S \odot 0 = 0 \odot S = \emptyset$ and $S \otimes T = \{\ (b \otimes b', \alpha \otimes \alpha') \mid (b, \alpha) \in S, (b', \alpha') \in T, b, b', \alpha, \alpha' \neq 0\ \}$. Given $\alpha \in \mathcal{T}_{SKAT}$ and $\sigma \in \Sigma$ we define the set of expressions derived from α w.r.t. a letter σ by

$$\Delta_\sigma(\alpha) = \{\ \alpha' \mid (b, \alpha') \in \partial_\sigma(\alpha) \text{ for some } b\ \}.$$

The functions ∂_σ, out, and Δ_σ are naturally extended to sets of SKAT expressions. Then, we define the set of expressions derived from α w.r.t. a word $x \in \Sigma^*$ inductively by $\Delta_\varepsilon(\alpha) = \{\alpha\}$ and $\Delta_{x\sigma}(\alpha) = \Delta_\sigma(\Delta_x(\alpha))$.

We denote by $\Delta(\alpha)$ the set of all expressions derived from α, i.e. $\Delta(\alpha) = \bigcup_{x \in \Sigma^*} \Delta_x(\alpha)$. Given $\alpha \in \mathcal{T}_{\mathsf{SKAT}}$, we define the partial derivative automaton associated to α by $\mathcal{A}(\alpha) = \langle \Delta(\alpha), \alpha, \mathsf{out}, \delta_\alpha \rangle$, where

$$\delta_\alpha = \{\ (\gamma, (b, \sigma), \gamma') \mid \gamma \in \Delta(\alpha), (b, \gamma') \in \partial_\sigma(\gamma)\ \}.$$

In order to justify the correctness of the partial derivative automaton, i.e., to show that $\mathsf{GSS}(\mathcal{A}(\alpha)) = \mathsf{GSS}(\alpha)$, we first argue that for every SKAT expression α the set $\Delta(\alpha)$ is finite. For this one can use a similar argument as the one used for SKA in Section 2. First define a function π_{SKAT} with an (almost) identical definition as π, but with entry $\pi_{\mathsf{SKAT}}(b) = \emptyset$ instead of $\pi(0) = \pi(1) = \emptyset$. Finally, using an almost identical proof as for Proposition 9, show by induction on the structure of $\alpha \in \mathcal{T}_{\mathsf{SKAT}}$ that $\Delta^+(\alpha) \subseteq \pi_{\mathsf{SKA}}(\alpha)$, where again $\Delta^+(\alpha)$ is the set of expressions derived from α excluding the trivial derivation w.r.t. the empty word ε. Then, the finiteness of $\Delta(\alpha)$ follows from the finiteness of $\pi_{\mathsf{SKAT}}(\alpha)$. Finally, the correctness of the partial derivative automaton is guaranteed by the following result.

Proposition 14. *For every SKAT expression γ and $x \in (\mathsf{At} \times \Sigma)^* \cdot \mathsf{At}$ the following hold:*

i. if $x = \mathtt{v}$, then $x \in \mathsf{GSS}(\gamma)$ if and only if $\mathtt{v} \leq \mathsf{out}(\gamma)$;

ii. if $x = \mathtt{v}\sigma x'$, then $x \in \mathsf{GSS}(\gamma)$ if and only if there is some $(b, \gamma') \in \partial_\sigma(\gamma)$, such that $\mathtt{v} \leq b$ and $x' \in \mathsf{GSS}(\gamma')$.

Proof. For $i.$ the proof is by induction on the structure of γ. For $ii.$ we present the cases for $\gamma = \alpha\beta$, $\gamma = \alpha \times \beta$. Let $\gamma = \alpha\beta$ and $x = \mathtt{v}\sigma x'$. One has $x \in \mathsf{GSS}(\alpha\beta)$ iff $x \in \mathsf{GSS}(\alpha) \diamond \mathsf{GSS}(\beta)$. This means that either,

$$\begin{array}{ll} & \mathtt{v} \in \mathsf{GSS}(\alpha) \text{ and } x \in \mathsf{GSS}(\beta) \\ \Leftrightarrow & \mathtt{v} \leq \mathsf{out}(\alpha), \mathtt{v} \leq b \text{ and } x' \in \mathsf{GSS}(\gamma') \text{ for some } (b, \gamma') \in \partial_\sigma(\beta) \\ \Leftrightarrow & \mathtt{v} \leq \mathsf{out}(\alpha)b \text{ and } x' \in \mathsf{GSS}(\gamma') \text{ for some } (\mathsf{out}(\alpha)b, \gamma') \in \partial_\sigma(\alpha\beta), \end{array}$$

or $x' = x_1 \diamond x_2$, with $\mathtt{v}\sigma x_1 \in \mathsf{GSS}(\alpha)$ and $x_2 \in \mathsf{GSS}(\beta)$. Now,

$$\begin{array}{ll} & \mathtt{v}\sigma x_1 \in \mathsf{GSS}(\alpha) \text{ and } x_2 \in \mathsf{GSS}(\beta) \\ \Leftrightarrow & \text{for some } (b, \gamma') \in \partial_\sigma(\alpha), \mathtt{v} \leq b, x_1 \in \mathsf{GSS}(\gamma') \text{ and } x_2 \in \mathsf{GSS}(\beta) \\ \Leftrightarrow & \text{for some } (b, \gamma'\beta) \in \partial_\sigma(\alpha\beta), \mathtt{v} \leq b \\ & \text{and } x' = x_1 \diamond x_2 \in \mathsf{GSS}(\gamma') \diamond \mathsf{GSS}(\beta) = \mathsf{GSS}(\gamma'\beta). \end{array}$$

Consider $\gamma = \alpha \times \beta$ and $x = \mathtt{v}\sigma x'$. One has $x \in \mathsf{GSS}(\alpha \times \beta)$ iff $x \in \mathsf{GSS}(\alpha) \times \mathsf{GSS}(\beta)$. This means that either, $x = (\mathtt{v}\sigma_1 x_1) \times (\mathtt{v}\sigma_2 x_2)$ for some $\mathtt{v}\sigma_1 x_1 \in \mathsf{GSS}(\alpha)$, $\mathtt{v}\sigma_2 x_2 \in \mathsf{GSS}(\beta)$ such that $\sigma = \sigma_1 \cup \sigma_2$ and $x' = x_1 \times x_2$, or $\mathtt{v} \in \mathsf{GSS}(\alpha)$ and $x \in \mathsf{GSS}(\beta)$, $\mathtt{v} \in \mathsf{GSS}(\beta)$ and $x \in \mathsf{GSS}(\alpha)$. The proof for the two last cases are analogous to the first case for the concatenation. Furthermore, one has

$$\begin{array}{ll} & \mathtt{v}\sigma_1 x_1 \in \mathsf{GSS}(\alpha) \text{ and } \mathtt{v}\sigma_2 x_2 \in \mathsf{GSS}(\beta) \\ \Leftrightarrow & \text{for some } (b_1, \gamma_1') \in \partial_{\sigma_1}(\alpha) \text{ and } (b_2, \gamma_2') \in \partial_{\sigma_2}(\beta) \\ & \mathtt{v} \leq b_1, x_1 \in \mathsf{GSS}(\gamma_1'), \mathtt{v} \leq b_2 \text{ and } x_2 \in \mathsf{GSS}(\gamma_2') \\ \Leftrightarrow & \text{for some } (b_1 \times b_2, \gamma_1' \times \gamma_2') \in \partial_\sigma(\alpha \times \beta), \mathtt{v} \leq b_1 \times b_2 \\ & \text{and } x' = x_1 \times x_2 \in \mathsf{GSS}(\gamma_1') \times \mathsf{GSS}(\gamma_2') = \mathsf{GSS}(\gamma_1' \times \gamma_2'). \end{array}$$

□ □

3.4 Equivalence of SKAT Expressions

In this subsection we present an algorithm for testing equivalence of SKAT expressions, similar to the one presented for SKA in Subsection 2.3. The algorithm implicitly uses the definition of the partial derivative automaton associated to an SKAT expression, as well as a determinization algorithm for NTA's presented in [8]. In order to deal with boolean expressions in the transitions of an NTA, some notions relating boolean expressions and sets of valuations have first to be introduced. Consider the set of atoms $\mathsf{At} = \{\alpha_0, \ldots, \alpha_{2^l-1}\}$, where $l = |\mathsf{T}|$, with the natural order induced by their binary representation. We define the function

$$\begin{array}{rccl} \mathsf{V}: & \mathcal{B}_{\mathsf{SKAT}} & \longrightarrow & 2^{\{0,\ldots,2^l-1\}} \\ & b & \longmapsto & \mathsf{V}_b = \{\ i \mid \alpha_i \leq b, 0 \leq i \leq 2^l - 1\ \}. \end{array}$$

This representation of boolean expressions is such that $\mathsf{V}_b = \mathsf{V}_{b'}$ if and only if b and b' are logically equivalent expressions. We consider V_b as a canonical representation of b and write $\alpha_i \leq \mathsf{V}_b$ if and only if $i \in \mathsf{V}_b$. We then define $\mathsf{V}_{X,\sigma,s'} = \bigcup \{\ \mathsf{V}_b \mid (b, s') \in \partial_\sigma(X)\ \}$ for $s' \in \Delta_\sigma(X)$ and, for each $Y \subseteq \Delta_\sigma(X)$, $\mathsf{V}_{X,\sigma,Y} = \bigcap \big(\{\ \mathsf{V}_{X,\sigma,s'} \mid s' \in Y\ \} \cup \{\ \overline{\mathsf{V}}_{X,\sigma,s'} \mid s' \in \Delta_\sigma(X) \setminus Y\ \}\big)$. The adapted version of equivalence testing based on partial derivatives is presented below.

Algorithm 2: Naive algorithm for deciding SKAT expression equivalence.

```
1  def NAIVE(α, β):
2    R is empty; todo = {({α}, {β})}
3    while todo is not empty, do:
4          extract (X,Y) from todo
5          if (X,Y) ∈ R then skip
6          if out(X) ≠ out(Y) then return False
7          for all σ ∈ Σ,
8            B1 = ∂σ(X); B2 = ∂σ(Y)
```

```
9             if  ⋃_{(b1,-)∈B1} V_{b1} = ⋃_{(b2,-)∈B2} V_{b2}  then
10              for X' ⊆ Δσ(X):
11                for Y' ⊆ Δσ(Y):
12                  if V_{X,σ,X'} ∩ V_{Y,σ,Y'} ≠ ∅:
13                    if X' ≠ Y':
14                      insert (X', Y') in todo
15                      insert (X, Y) in R
16            else: return False
17    return True
```

4 Conclusion

In this paper we extended the notion of derivative to sets of (guarded) synchronous strings and showed that the methods based on derivatives lead to simple and elegant decision procedures for testing SKA and SKAT expressions equivalence. For practical usage, more efficient versions of the algorithms here proposed must be considered as already done for KA and KAT. Considering bisimulation equivalence instead of language equivalence for SKA and SKAT is also a topic for future work.

References

[1] Almeida, M., Moreira, N., Reis, R.: Testing regular languages equivalence. Journal of Automata, Languages and Combinatorics 15(1/2), 7–25 (2010)

[2] Almeida, R., Broda, S., Moreira, N.: Deciding KAT and Hoare logic with derivatives. In: Faella, M., Murano, A. (eds.) 3rd GANDALF. EPTCS, vol. 96, pp. 127–140 (2012)

[3] Antimirov, V.M.: Partial derivatives of regular expressions and finite automaton constructions. Theoret. Comput. Sci. 155(2), 291–319 (1996)

[4] Berry, G., Gonthier, G.: The esterel synchronous programming language: Design, semantics, implementation. Sci. Comput. Program. 19(2), 87–152 (1992)

[5] Bonchi, F., Pous, D.: Checking NFA equivalence with bisimulations up to congruence. In: Giacobazzi, R., Cousot, R. (eds.) POPL '13. pp. 457–468. ACM (2013)

[6] Braibant, T., Pous, D.: Deciding Kleene algebras in Coq. Logical Methods in Computer Science 8(1) (2012)

[7] Broda, S., Machiavelo, A., Moreira, N., Reis, R.: On the average size of Glushkov and equation automata for KAT expressions. In: FCT 2013. pp. 72–83. No. 8070 in LNCS, Springer (2013)

[8] Broda, S., Machiavelo, A., Moreira, N., Reis, R.: On the Equivalence of Automata for KAT-expressions. In: CiE 2014. LNCS, vol. 8493. Springer (2014), to appear

[9] Brzozowski, J.A.: Derivatives of regular expressions. JACM 11(4), 481–494 (1964)

[10] Champarnaud, J.M., Ziadi, D.: From Mirkin's prebases to Antimirov's word partial derivatives. Fundam. Inform. 45(3), 195–205 (2001)

[11] Hopcroft, J., Karp, R.M.: A linear algorithm for testing equivalence of finite automata. Tech. Rep. TR 71 -114, University of California, Berkeley, California (1971)

[12] Kozen, D.: Kleene algebra with tests. Trans. on Prog. Lang. and Systems 19(3), 427–443 (05 1997)

[13] Kozen, D.: On Hoare logic and Kleene algebra with tests. ACM Trans. Comput. Log. 1(1), 60–76 (2000)

[14] Kozen, D.: Automata on guarded strings and applications. Matématica Contemporânea 24, 117–139 (2003)

[15] Kozen, D.: On the coalgebraic theory of Kleene algebra with tests. Tech. Rep. `http://hdl.handle.net/1813/10173`, Cornell University (05 2008)

[16] Milner, R.: Communication and concurrency. PHI Series in computer science, Prentice Hall (1989)

[17] Mirkin, B.G.: An algorithm for constructing a base in a language of regular expressions. Engineering Cybernetics 5, 51—57 (1966)

[18] Nelma Moreira, D.P., de Sousa, S.M.: Deciding regular expressions (in-)equivalence in Coq. In: Griffin, T.G., Kahl, W. (eds.) 13th RAMiCS 2012. LNCS, vol. 7560, pp. 98–113. Springer (2012)

[19] Pereira, D.: Towards Certified Program Logics for the Verification of Imperative Programs. Ph.D. thesis, University of Porto (2013)

[20] Prisacariu, C.: Synchronous Kleene algebra. J. Log. Algebr. Program. 79(7), 608–635 (2010)

[21] Rot, J., Bonsangue, M.M., Rutten, J.J.M.M.: Coinductive proof techniques for language equivalence. In: Dediu, A.H., Martín-Vide, C., Truthe, B. (eds.) LATA 2013. LNCS, vol. 7810, pp. 480–492. Springer (2013)

[22] Silva, A.: Position automata for Kleene algebra with tests. Sci. Ann. Comp. Sci. 22(2), 367–394 (2012)